Database Management System

John Ren G. Santos
ChristianJay B. Tantan

Bachelor of Science in Information Technology
College of Computing Studies

VISION

Laguna University shall be a socially responsive educational institution
of choice providing holistically developed individuals
in the Asia-Pacific Region

MISSION

Laguna University is committed to produce academically prepared
and technically skilled individuals who are socially
and morally upright

Table of Contents

List of Figures

List of Table

Course Code: PC 2207 Database Management System

Course Description: This course unit offers an introduction to the latest, cutting-edge research outcomes in the area of database management systems (DBMSs). It starts with a brief overview of the internal architecture of traditional DBMSs, and proceeds to cover a range of advanced systems that extend that architecture to different execution environments than the classical, centralized one. The viewpoint adopted throughout is systems- oriented and research-oriented. Focus falls on the impacts on classical query processing functionality (i.e., impacts on other DBMS-provided services such as storage, concurrency and transaction management are largely ignored) with the use of Adv. SQL.

This course discusses database system implementation techniques such as concurrency control, transaction processing, scheduling, and recovery. It also covers discussions on the current trends in database systems in different environments, distributed databases, object-oriented databases, knowledge-based database systems and emerging database applications and technologies.

Course Intended Learning Outcomes (CILO):

At the end of this program, graduates will have the ability to:

1. Perform normalization
2. Limit database redundancy
3. Perform database maintenance easily
4. Reduce the development time of a database
5. Perform database backup and restore
6. Exhibit the values of love, respect and humility as an individual inhis workplace and community.

Course Requirements:

GRADING SYSTEM

Class Standing		**60 %**
Attendance	10 %	
/Recitation	20 %	
Quizzes	10 %	
Assignments	20 %	
Activities		
Major Exams		**40 %**
Periodic Grade		**100 %**

Prelim Grade = 60% (Class standing) + 40% (Prelim exam)
Midterm Grade= 30% (Prelim Grade) + 70 % (Midterm Grade): [60% (Midterm Class standing) + 40% (Midterm exam)]
Final Grade = 30% (Midterm Grade) + 70 % (Final Grade): [60% (Final Class standing) + 40% (Final exam)]

MODULE 1
ACCESSING DATA IN MULTIPLE TABLES

Introduction

MySQL supports several methods that you can use to access multiple tables in a single SQL statement. The first of these is to create a join in the statement that defines the tables to be linked together. Another method that you can use is to embed a subquery in your statement so that you can use the data returned by the subquery in the main SQL statement. In addition, you can create a union that joins together two SELECT statements in order to produce a result set that contains data retrieved by both statements. In this chapter, you learn about all three methods for accessing data in multiple tables.

Learning Outcomes

At the end of this module, students should be able to:

1. Write a MySQL query to fetch results.
2. Use the WHERE clause to filter results.
3. Use operators in MySQL statements.

Lesson 1. Introduction to Joins

In a normalized database, groups of data are stored in individual tables, and relationships are established between those tables to link related data. As a result, often when creating SELECT, UPDATE, or DELETE statements, you want to be able to access data in different tables to carry out an operation affected by those relationships (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

To support the capability to access data in multiple tables, MySQL allows you to create joins in a statement that define how data is accessed in multiple tables. A join is a condition defined in a SELECT, UPDATE, or DELETE statement that links together two or more tables. In this section, you learn how to create joins in each of these types of statements. Although much of the discussion focuses on creating joins in a SELECT statement, which is where you will most commonly use joins, many of the elements used in a SELECT join are the same elements used for UPDATE and DELETE joins (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

As you're creating your SELECT statements for your applications, you may want to create statements that return data stored in different tables. The result set, though, cannot contain data that appears arbitrary in nature. In other words, the result set must be displayed in a way that suggests that the data could have been retrieved from one table. The data must be integrated and logical, despite the fact that it might come from different tables (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

To achieve this integration, MySQL allows you to add joins to your SELECT statements that link together two or more tables:

```
<select statement>::=
SELECT
[<select option> [<select option>...]]
{* | <select list>}
[
        FROM {<table reference> | <join definition>}
        [WHERE <expression> [{<operator> <expression>}...]]
        [GROUP BY <group by definition>]
        [HAVING <expression> [{<operator> <expression>}...]]
        [ORDER BY <order by definition>]
[LIMIT [<offset>,] <row count>]
]
```

```
<join definition>::=
{<table reference>, <table reference> [{, <table reference>}...]}
| {<table reference> [INNER | CROSS ] JOIN <table reference> [<join condition>]}
| {<table reference> STRAIGHT_JOIN <table reference>}
| {<table reference> LEFT [OUTER] JOIN <table reference> [<join condition>]}
| {<table reference> RIGHT [OUTER] JOIN <table reference> [<join condition>]}
| {<table reference> NATURAL [{LEFT | RIGHT} [OUTER]] JOIN <table reference>}
<table reference>::=
<table name> [[AS] <alias>]
[{USE | IGNORE | FORCE} INDEX <index name> [{, <index name>}...]]
<join condition>::=
ON <expression> [{<operator> <expression>}...]
| USING (<column> [{, <column>}...])
```

The syntax is not presented in its entirety and contains only those elements that are relevant to defining a join in a SELECT statement. In addition, the syntax includes elements that you have not seen before. These elements are based on the FROM clause, which has been modified from the original definition. As you can see, the FROM clause syntax is now as follows:

```
FROM {<table reference> | <join definition>}
```

When you originally learned about the SELECT statement syntax in Chapter 7, the clause was quite different:

```
FROM <table reference> [{, <table reference>}...]
```

Originally, the clause included only the <table reference> placeholders, and there was no mention of the **<join definition>** placeholder. This was done for the sake of brevity and to avoid presenting too much information at one time. As a result, the original syntax suggested that the FROM clause could include only one or more table references. Another way to describe the FROM clause is by also including the **<join definition>** placeholder, which defines how to add joins to a SELECT statement (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

As you can see from the updated syntax of the FROM clause, the clause can include either a table reference or a join definition. You've already seen numerous examples of SELECT statements that use the **<table reference>** format. These are the SELECT statements that include only one table name in the FROM clause. If you plan to reference multiple tables in your FROM clause, you are defining some type of join, in which case the <join definition> placeholder applies (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The **<join definition>** placeholder refers to a number of different types of joins, as the following syntax shows:

```
<join definition>::=
{<table reference>, <table reference> [{, <table reference>}...]}
| {<table reference> [INNER | CROSS ] JOIN <table reference> [<join condition>]}
| {<table reference> STRAIGHT_JOIN <table reference>}
| {<table reference> LEFT [OUTER] JOIN <table reference> [<join condition>]}
| {<table reference> RIGHT [OUTER] JOIN <table reference> [<join condition>]}
| {<table reference> NATURAL [{LEFT | RIGHT} [OUTER]] JOIN <table reference>}
```

A complete join that produces all possible row combinations is called a Cartesian product. For example, joining each row in a 100-row table to each row in a 200-row table produces a result containing 100 × 200, or 20,000 rows. With larger tables, or joins between more than two tables, the result set for a Cartesian product can easily become immense. Because of that, and because you rarely want all the combinations anyway, a join normally includes an **ON** or **USING** clause that specifies how to join rows between tables. (This requires that each table have one or more columns of common information that can be used to link them together logically.) You can also include a **WHERE** clause that restricts which of the joined rows to select. Each of these clauses narrows the focus of the query(Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The examples assume that you have an art collection and use the following two tables to record your acquisitions. artist lists those painters whose works you want to collect, and painting lists each painting that you've actually purchased(Ramakrishhan, Raghu and

```
CREATE TABLE artist
(
	a_id INT UNSIGNED NOT NULL AUTO_INCREMENT, # artist ID
	name VARCHAR(30) NOT NULL, # artist name
	PRIMARY KEY (a_id),
	UNIQUE (name)
);

CREATE TABLE painting
(
	a_id INT UNSIGNED NOT NULL, # artist ID
	p_id INT UNSIGNED NOT NULL AUTO_INCREMENT, # painting ID
	title VARCHAR(100) NOT NULL, # title of painting
	state VARCHAR(2) NOT NULL, # state where purchased
	price INT UNSIGNED, # purchase price (dollars)
	INDEX (a_id),
	PRIMARY KEY (p_id)
);
```

Gehrke,Johannes,2018):

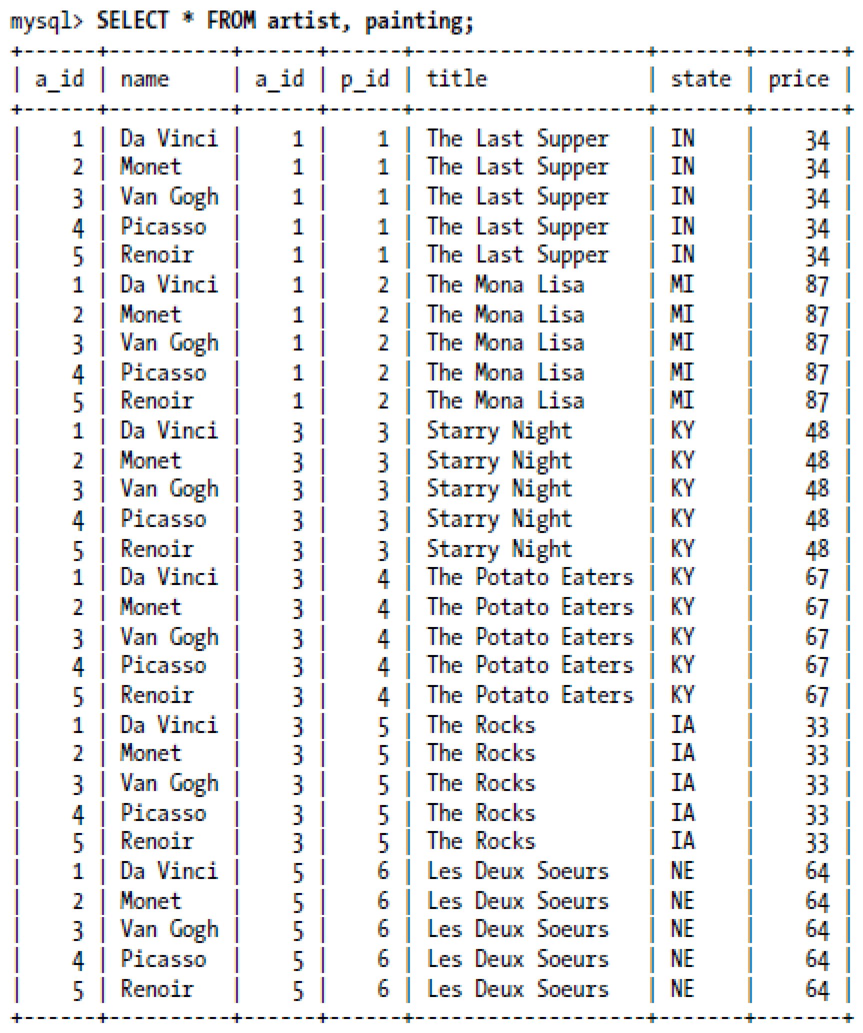
You've just begun the collection, so the tables contain only the following rows:

```
mysql> SELECT * FROM artist ORDER BY a_id;
+------+----------+
| a_id | name     |
+------+----------+
|    1 | Da Vinci |
|    2 | Monet    |
|    3 | Van Gogh |
|    4 | Picasso  |
|    5 | Renoir   |
+------+----------+
mysql> SELECT * FROM painting ORDER BY a_id, p_id;
+------+------+-------------------+-------+-------+
| a_id | p_id | title             | state | price |
+------+------+-------------------+-------+-------+
|    1 |    1 | The Last Supper   | IN    |    34 |
|    1 |    2 | The Mona Lisa     | MI    |    87 |
|    3 |    3 | Starry Night      | KY    |    48 |
|    3 |    4 | The Potato Eaters | KY    |    67 |
|    3 |    5 | The Rocks         | IA    |    33 |
|    5 |    6 | Les Deux Soeurs   | NE    |    64 |
+------+------+-------------------+-------+-------+
```

The low values in the price column of the painting table betray the fact that your collection actually contains only cheap facsimiles, not the originals. Well, that's all right: who can afford the originals?

Each table contains partial information about your collection. For example, the **artist** table doesn't tell you which paintings each artist produced, and the **painting** table lists artist IDs but not their names. To use the information in both tables, you can ask MySQL to show you various combinations of artists and paintings by writing a query that performs a join. A join names two or more tables after the **FROM** keyword. In the output column list, you can name columns from any or all the joined tables, or use expressions that are based on those columns, **tbl_name .*** to select all columns from a given table, or * to select all columns from all tables(Ramakrishhan, Raghu and Gehrke,Johannes,2018).

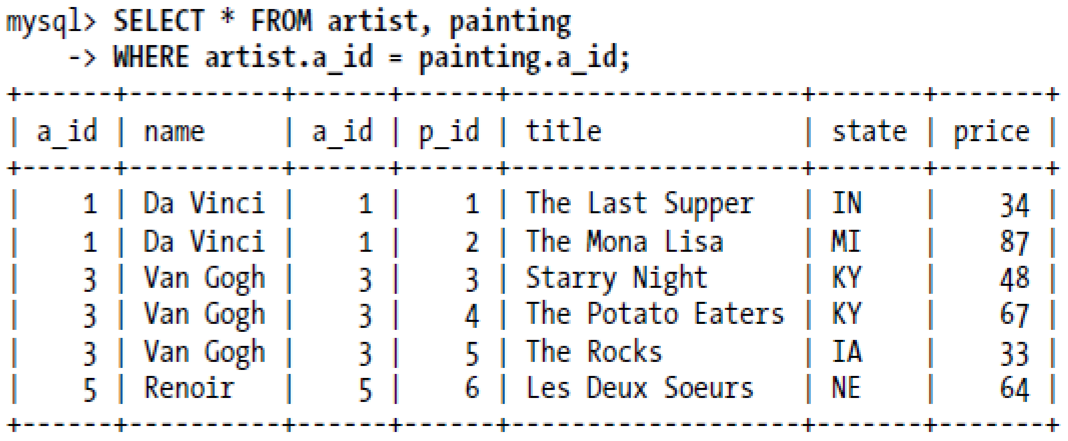
The simplest join involves two tables and selects all columns from each. With no restrictions, the join generates output for all combinations of rows (that is, the Cartesian product). The following complete join between the artist and painting tables shows this:



```
mysql> SELECT * FROM artist, painting;
+------+----------+------+------+-------------------+-------+-------+
| a_id | name     | a_id | p_id | title             | state | price |
+------+----------+------+------+-------------------+-------+-------+
|    1 | Da Vinci |    1 |    1 | The Last Supper   | IN    |    34 |
|    2 | Monet    |    1 |    1 | The Last Supper   | IN    |    34 |
|    3 | Van Gogh |    1 |    1 | The Last Supper   | IN    |    34 |
|    4 | Picasso  |    1 |    1 | The Last Supper   | IN    |    34 |
|    5 | Renoir   |    1 |    1 | The Last Supper   | IN    |    34 |
|    1 | Da Vinci |    1 |    2 | The Mona Lisa     | MI    |    87 |
|    2 | Monet    |    1 |    2 | The Mona Lisa     | MI    |    87 |
|    3 | Van Gogh |    1 |    2 | The Mona Lisa     | MI    |    87 |
|    4 | Picasso  |    1 |    2 | The Mona Lisa     | MI    |    87 |
|    5 | Renoir   |    1 |    2 | The Mona Lisa     | MI    |    87 |
|    1 | Da Vinci |    3 |    3 | Starry Night      | KY    |    48 |
|    2 | Monet    |    3 |    3 | Starry Night      | KY    |    48 |
|    3 | Van Gogh |    3 |    3 | Starry Night      | KY    |    48 |
|    4 | Picasso  |    3 |    3 | Starry Night      | KY    |    48 |
|    5 | Renoir   |    3 |    3 | Starry Night      | KY    |    48 |
|    1 | Da Vinci |    3 |    4 | The Potato Eaters | KY    |    67 |
|    2 | Monet    |    3 |    4 | The Potato Eaters | KY    |    67 |
|    3 | Van Gogh |    3 |    4 | The Potato Eaters | KY    |    67 |
|    4 | Picasso  |    3 |    4 | The Potato Eaters | KY    |    67 |
|    5 | Renoir   |    3 |    4 | The Potato Eaters | KY    |    67 |
|    1 | Da Vinci |    3 |    5 | The Rocks         | IA    |    33 |
|    2 | Monet    |    3 |    5 | The Rocks         | IA    |    33 |
|    3 | Van Gogh |    3 |    5 | The Rocks         | IA    |    33 |
|    4 | Picasso  |    3 |    5 | The Rocks         | IA    |    33 |
|    5 | Renoir   |    3 |    5 | The Rocks         | IA    |    33 |
|    1 | Da Vinci |    5 |    6 | Les Deux Soeurs   | NE    |    64 |
|    2 | Monet    |    5 |    6 | Les Deux Soeurs   | NE    |    64 |
|    3 | Van Gogh |    5 |    6 | Les Deux Soeurs   | NE    |    64 |
|    4 | Picasso  |    5 |    6 | Les Deux Soeurs   | NE    |    64 |
|    5 | Renoir   |    5 |    6 | Les Deux Soeurs   | NE    |    64 |
+------+----------+------+------+-------------------+-------+-------+
```

The statement output illustrates why a complete join generally is not useful: it produces a lot of output, and the result is not meaningful. Clearly, you're not maintaining these tables to match every artist with every painting, which is what the preceding statement does. An unrestricted join in this case produces nothing of value(Ramakrishhan, Raghu and Gehrke,Johannes,2018).

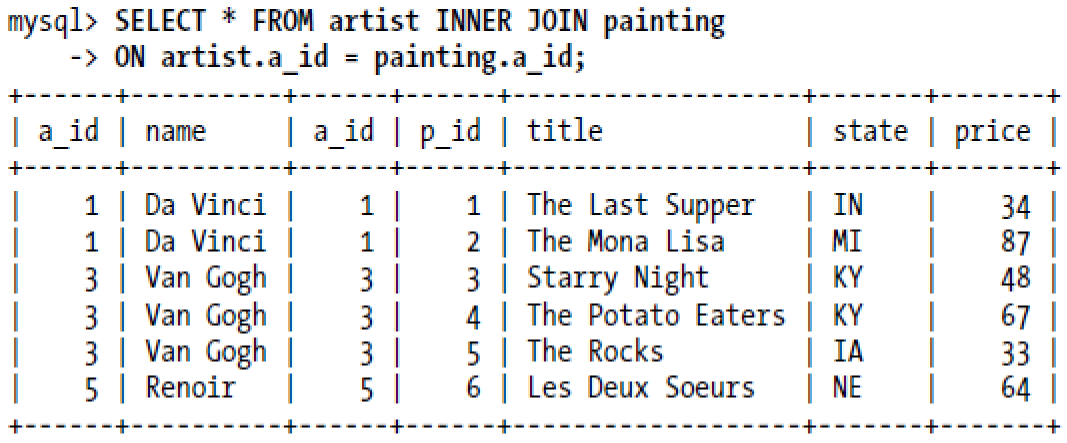
To answer questions meaningfully, you must combine the two tables in a way that produces only the relevant matches. Doing so is a matter of including appropriate join conditions. For example, to produce a list of paintings together with the artist names, you can associate rows from the two tables using a simple **WHERE** clause that matches up values in the artist ID column that is common to both tables and that serves as the link between them(Ramakrishhan, Raghu and Gehrke,Johannes,2018):



```
mysql> SELECT * FROM artist, painting
    -> WHERE artist.a_id = painting.a_id;
+------+----------+------+------+-------------------+-------+-------+
| a_id | name     | a_id | p_id | title             | state | price |
+------+----------+------+------+-------------------+-------+-------+
|    1 | Da Vinci |    1 |    1 | The Last Supper   | IN    |    34 |
|    1 | Da Vinci |    1 |    2 | The Mona Lisa     | MI    |    87 |
|    3 | Van Gogh |    3 |    3 | Starry Night      | KY    |    48 |
|    3 | Van Gogh |    3 |    4 | The Potato Eaters | KY    |    67 |
|    3 | Van Gogh |    3 |    5 | The Rocks         | IA    |    33 |
|    5 | Renoir   |    5 |    6 | Les Deux Soeurs   | NE    |    64 |
+------+----------+------+------+-------------------+-------+-------+
```

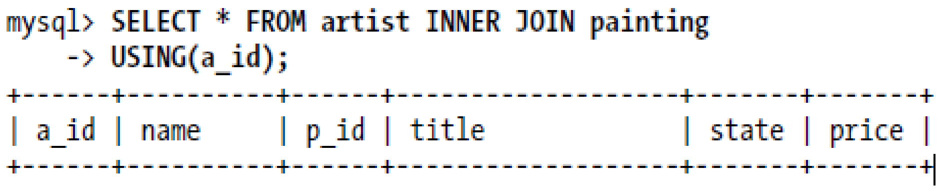
The column names in the **WHERE** clause include table qualifiers to make it clear which **a_id** values to compare. The output indicates who painted each painting, and, conversely, which paintings by each artist are in your collection(Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Another way to write the same join is to use **INNER JOIN** rather than the comma operator and indicate the matching conditions with an **ON** clause:

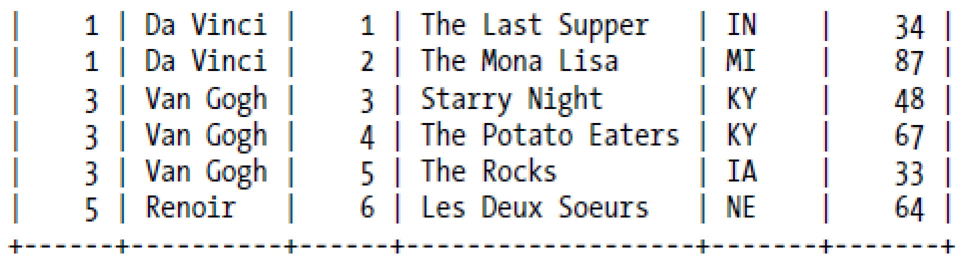


```
mysql> SELECT * FROM artist INNER JOIN painting
    -> ON artist.a_id = painting.a_id;
+------+----------+------+------+-------------------+-------+-------+
| a_id | name     | a_id | p_id | title             | state | price |
+------+----------+------+------+-------------------+-------+-------+
|    1 | Da Vinci |    1 |    1 | The Last Supper   | IN    |    34 |
|    1 | Da Vinci |    1 |    2 | The Mona Lisa     | MI    |    87 |
|    3 | Van Gogh |    3 |    3 | Starry Night      | KY    |    48 |
|    3 | Van Gogh |    3 |    4 | The Potato Eaters | KY    |    67 |
|    3 | Van Gogh |    3 |    5 | The Rocks         | IA    |    33 |
|    5 | Renoir   |    5 |    6 | Les Deux Soeurs   | NE    |    64 |
+------+----------+------+------+-------------------+-------+-------+
```

In the special case that the matched columns have the same name in both tables and are compared using the = operator, you can use an **INNER JOIN** with a **USING** clause instead. This requires no table qualifiers, and each join column is named only on



```
mysql> SELECT * FROM artist INNER JOIN painting
    -> USING(a_id);
+------+----------+------+-------------------+-------+-------+
| a_id | name     | p_id | title             | state | price |
+------+----------+------+-------------------+-------+-------+
```



```
|    1 | Da Vinci |    1 | The Last Supper   | IN    |    34 |
|    1 | Da Vinci |    2 | The Mona Lisa     | MI    |    87 |
|    3 | Van Gogh |    3 | Starry Night      | KY    |    48 |
|    3 | Van Gogh |    4 | The Potato Eaters | KY    |    67 |
|    3 | Van Gogh |    5 | The Rocks         | IA    |    33 |
|    5 | Renoir   |    6 | Les Deux Soeurs   | NE    |    64 |
+------+----------+------+-------------------+-------+-------+
```

Note that when you write a query with a **USING** clause, **SELECT *** returns only one instance of each join column **(a_id).**

Setting up sample tables

First, create two tables called members and committees:

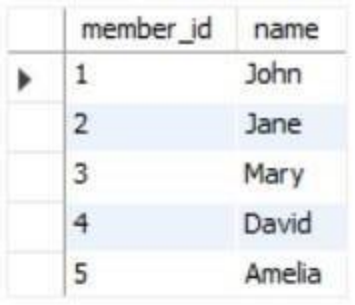
```
CREATE TABLE members (
		member_id INT AUTO_INCREMENT,
		name VARCHAR(100),
		PRIMARY KEY (member_id)
);

CREATE TABLE committees (
		committee_id INT AUTO_INCREMENT,
		name VARCHAR(100),
		PRIMARY KEY (committee_id)
);
```

Second, insert some rows into the tables members and committees :

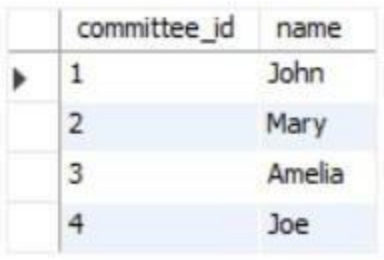
```
INSERT INTO members(name)
VALUES('John'),('Jane'),('Mary'),('David'),('Amelia');

INSERT INTO committees(name)
VALUES('John'),('Mary'),('Amelia'),('Joe');
```

Third, query data from the tables members and committees:

```
INSERT INTO members(name)
VALUES('John'),('Jane'),('Mary'),('David'),('Amelia');

INSERT INTO committees(name)
VALUES('John'),('Mary'),('Amelia'),('Joe');
```



member_id	name
1	John
2	Jane
3	Mary
4	David
5	Amelia



committee_id	name
1	John
2	Mary
3	Amelia
4	Joe

Some members are the committee members, and some are not. On the other hand, some committee members are in the members table, some are not.

Lesson 2. Inner Join

The INNER JOIN matches each row in one table with every row in other tables and allows you to query rows that contain columns from both tables.

The INNER JOIN is an optional clause of the SELECT statement. It appears immediately after the FROM clause. Here is the syntax of the INNER JOIN clause:

```
SELECT
   select_list
FROM t1
INNER JOIN t2 ON join_condition1
INNER JOIN t3 ON join_condition2
...;
```

In this syntax:

- First, specify the main table that appears in the FROM clause (t1).
- Second, specify the table that will be joined with the main table, which appears in the INNER JOIN clause (t2, t3,…).
- Third, specify a join condition after the ON keyword of the INNER JOIN clause. The join condition specifies the rule for matching rows between the main table and the table appeared in the INNER JOIN clause.

Assuming that you want to join two tables t1 and t2.

The following statement illustrates how to join two tables t1 and t2 using the INNER JOIN clause:

```
SELECT
    select_list
FROM
    t1
INNER JOIN t2 ON join_condition;
```

The INNER JOIN clause compares each row in the t1 table with every row in the t2 table based on the join condition.

If rows from both tables cause the join condition to evaluate to TRUE, the INNER JOIN creates a new row whose columns contain all columns of rows from the tables and includes this new row in the result set. Otherwise, the INNER JOIN justignores the rows.

In case no row between tables causes the join condition to evaluate to TRUE, the INNER JOIN returns an empty result set. This logic is also applied when you join more than 2 tables.

The following Venn diagram illustrates how the INNER JOIN clause works:

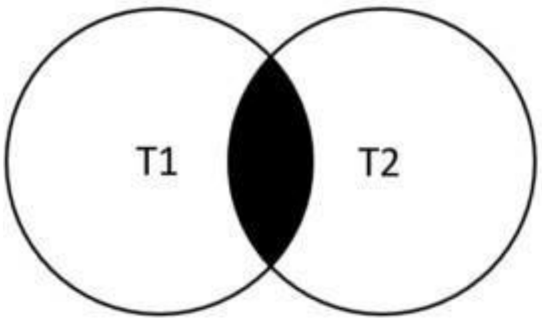


Figure 1.1 Inner Join Venn Diagram

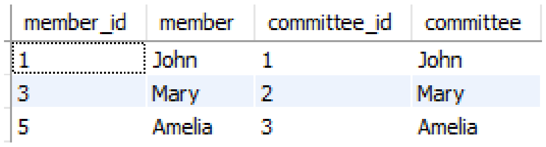
Source: https://www.mysqltutorial.org/mysql-inner-join.aspx

Example:

The following statement finds members who are also the committee members:

```
SELECT
   m.member_id,
   m.name as member,
   c.committee_id,
   c.name as committee
FROM
   members as m
INNER JOIN committees as c
         ON c.name = m.name;
```

In this example, the inner join clause used the values in the name columns in both tables members and committees to match.



member_id	member	committee_id	committee
1	John	1	John
3	Mary	2	Mary
5	Amelia	3	Amelia

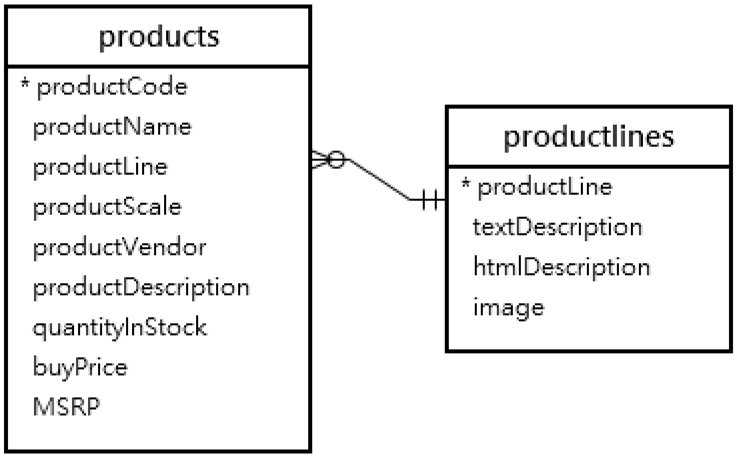
Because the name columns are the same in both tables, you can use the USING clause as shown in the following query:

```
SELECT
   m.member_id,
   m.name as member,
   c.committee_id,
   c.name as committee
FROM
   members as m
INNER JOIN committees as c USING(name);
```

Another Example

Let's look at the products and productlines tables in the sample database.

You could get the sample database from this link: **https://bit.ly/3fAa2nN**



In this diagram, the table products has the column productLine that references the column productline of the table productlines . The column productLine in the table products is called the foreign key column (*MySQL INNER JOIN By Practical Examples*, n.d.).

Typically, you join tables that have foreign key relationships like the productlines and products tables (*MySQL INNER JOIN By Practical Examples*, n.d.).

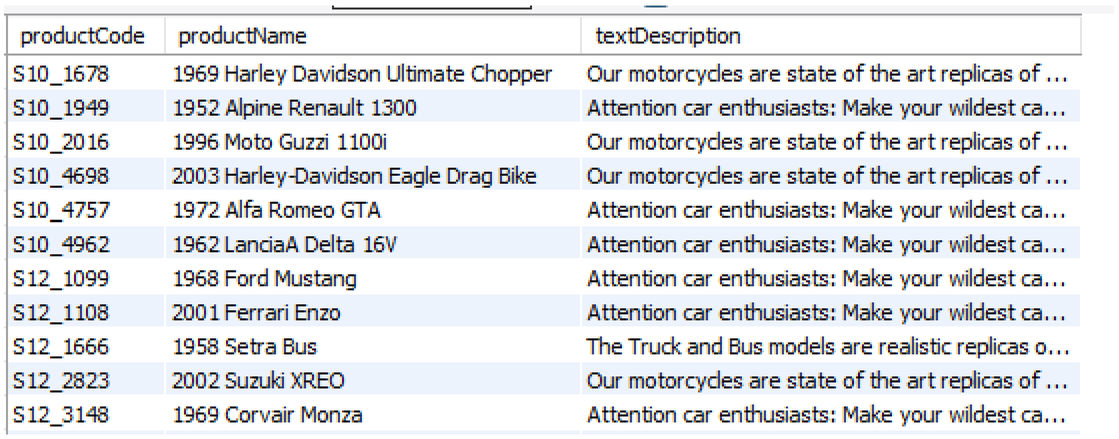
Suppose you want to get:

- The productCode and productName from the products table.
- The textDescription of product lines from the productlines table.

To do this, you need to select data from both tables by matching rows based on values in the productline column using the INNER JOIN clause as follows:

```
SELECT
   productCode,
   productName,
   textDescription
FROM
   products t1
INNER JOIN productlines t2
   ON t1.productline = t2.productline;
```

Output:



productCode	productName	textDescription
S10_1678	1969 Harley Davidson Ultimate Chopper	Our motorcycles are state of the art replicas of ...
S10_1949	1952 Alpine Renault 1300	Attention car enthusiasts: Make your wildest ca...
S10_2016	1996 Moto Guzzi 1100i	Our motorcycles are state of the art replicas of ...
S10_4698	2003 Harley-Davidson Eagle Drag Bike	Our motorcycles are state of the art replicas of ...
S10_4757	1972 Alfa Romeo GTA	Attention car enthusiasts: Make your wildest ca...
S10_4962	1962 LanciaA Delta 16V	Attention car enthusiasts: Make your wildest ca...
S12_1099	1968 Ford Mustang	Attention car enthusiasts: Make your wildest ca...
S12_1108	2001 Ferrari Enzo	Attention car enthusiasts: Make your wildest ca...
S12_1666	1958 Setra Bus	The Truck and Bus models are realistic replicas o...
S12_2823	2002 Suzuki XREO	Our motorcycles are state of the art replicas of ...
S12_3148	1969 Corvair Monza	Attention car enthusiasts: Make your wildest ca...

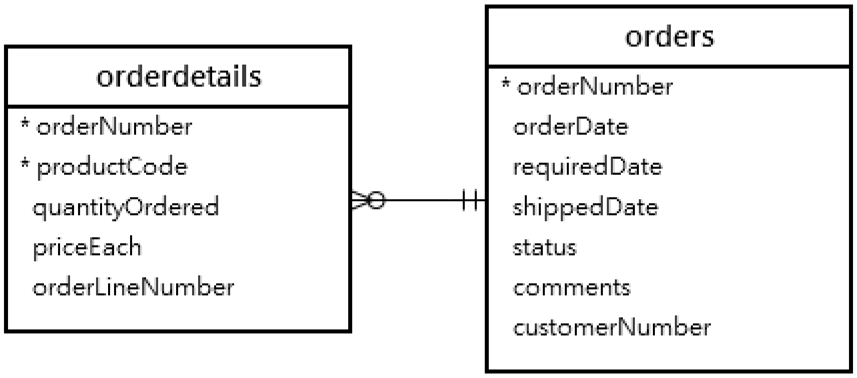
Because the joined columns of both tables have the same name productline, you can use the **USING** syntax:

```
SELECT
   productCode,
   productName,
   textDescription
FROM
   products
INNER JOIN productlines USING (productline);
```

The query returns the same result set. However, the USING syntax is much shorter and cleaner.

MySQL INNER JOIN with GROUP BY clause example

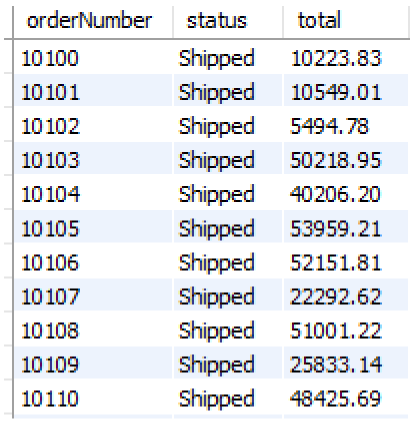
See the following orders and orderdetails tables:



This query returns order number, order status and total sales from the **orders** and **orderdetails** tables using the INNER JOIN clause with the GROUP BY clause:

```
SELECT
   t1.orderNumber,
   t1.status,
   SUM(quantityOrdered * priceEach) as total
FROM
   Orders as t1
INNER JOIN orderdetails as t2
   ON t1.orderNumber = t2.orderNumber
GROUP BY orderNumber;
```

Result:



orderNumber	status	total
10100	Shipped	10223.83
10101	Shipped	10549.01
10102	Shipped	5494.78
10103	Shipped	50218.95
10104	Shipped	40206.20
10105	Shipped	53959.21
10106	Shipped	52151.81
10107	Shipped	22292.62
10108	Shipped	51001.22
10109	Shipped	25833.14
10110	Shipped	48425.69

Lesson 3. Left Join

Similar to an inner join, a left join also requires a join-predicate. When joining two tables using a left join, the concepts of left and right tables are introduced. The left join selects data starting from the left table. For each row in the left table, the left join compares with every row in the right table. If the values in the two rows cause the join condition evaluates to true, the left join creates a new row whose columns contain all columns of the rows in both tables and includes this row in the result set (*MySQL Join Made Easy For Beginners*, n.d.).

If the values in the two rows are not matched, the left join clause still creates a new row whose columns contain columns of the row in the left table and NULL for columns of the row in the right table. In other words, the left join selects all data from the left table whether there are matching rows exist in the right table or not. In case there is no matching rows from the right table found, NULLs are used for columns of the row from the right table in the final result set (*MySQL Join Made Easy For Beginners*, n.d.).

Here is the basic syntax of a left join clause that joins two tables:

```
SELECT
   select_list
FROM
   t1
LEFT JOIN t2 ON
   join_condition;
```

In the above syntax, t1 is the left table and t2 is the right table.

The LEFT JOIN clause selects data starting from the left table (t1). It matches each row from the left table (t1) with every row from the right table(t2) based on the join_condition. If the rows from both tables cause the join condition evaluates to TRUE, the LEFT JOIN combine columns of rows from both tables to a new row and includes this new row in the result rows. In case the row from the left table (t1) does not match with any row from the right table(t2), the LEFT JOIN still combines columns of rows from both tables into a new row and include the new row in the result rows. However, it uses NULL for all the columns of the row from the right table (*Understanding MySQL LEFT JOIN Clause By Examples*, n.d.).

In other words, LEFT JOIN returns all rows from the left table regardless of whether a row from the left table has a matching row from the right table or not. If there is no match, the columns of the row from the right table will contain NULL (*Understanding MySQL LEFT JOIN Clause By Examples*, n.d.).

The following Venn diagram helps you visualize how the LEFT JOIN clause works:

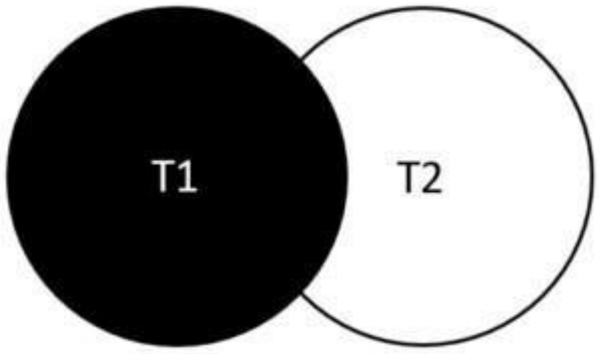
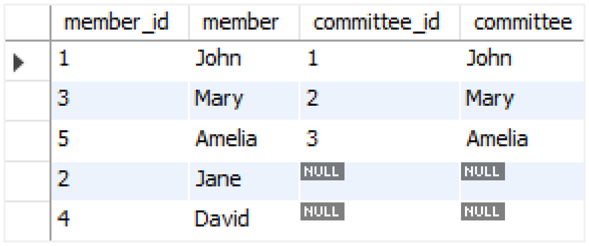


Figure 1.2 Left Join Venn Diagram

Source: https://www.mysqltutorial.org/mysql-left-join.aspx

```
SELECT
   m.member_id,
   m.name as member,
   c.committee_id,
   c.name as committee
FROM
   members as m
LEFT JOIN committees as c USING(name);
```

Result:

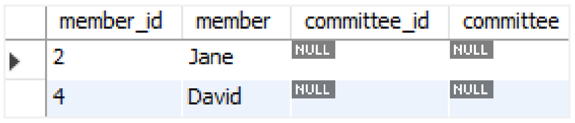


member_id	member	committee_id	committee
1	John	1	John
3	Mary	2	Mary
5	Amelia	3	Amelia
2	Jane	NULL	NULL
4	David	NULL	NULL

To find members who are not the committee members, you add a WHERE clause and IS NULL operator as follows:

```
SELECT
   m.member_id,
   m.name as member,
   c.committee_id,
   c.name as committee
FROM
   members as m
LEFT JOIN committees as c USING(name)
WHERE c.committee_id IS NULL;
```

Result:



member_id	member	committee_id	committee
2	Jane	NULL	NULL
4	David	NULL	NULL

Generally, this query pattern can find rows in the left table that does not have corresponding rows in the right table. This Venn diagram illustrates how to use the left join to select rows that only exist in the left table:

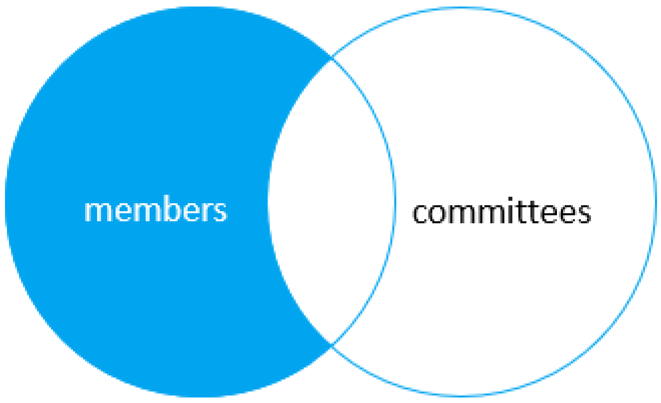
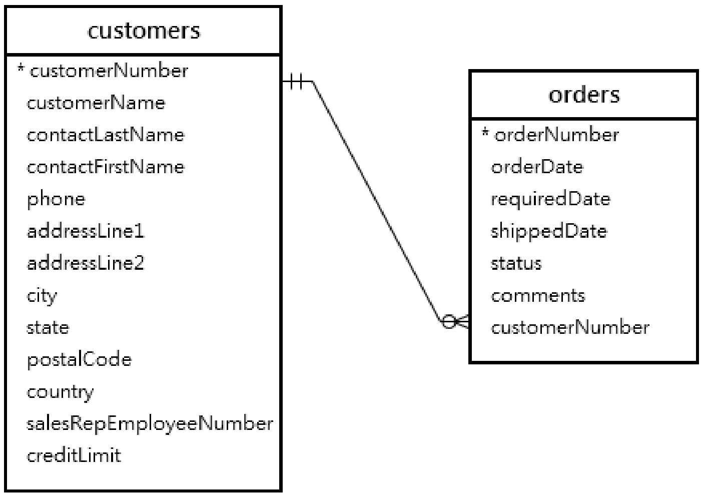


Figure 1.3 Left Join Result Venn Diagram

Source: https://www.mysqltutorial.org/mysql-join/

Other Examples:

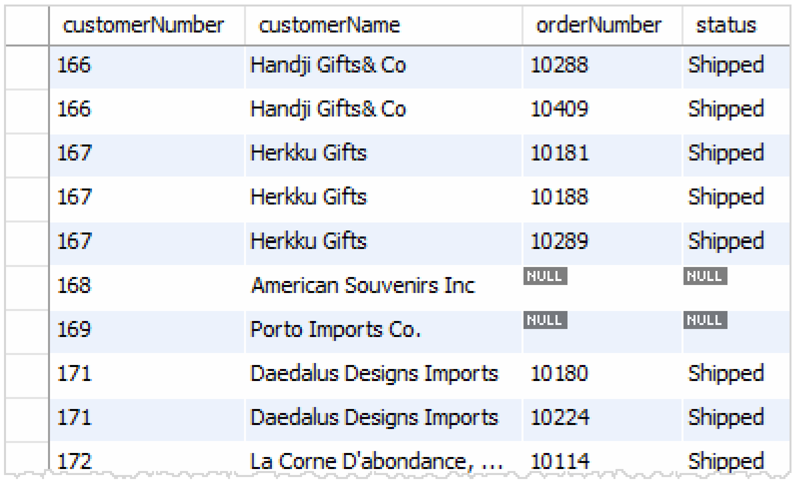
See the following tables customers and orders in the sample database. You could get the sample database from this link: **https://bit.ly/3fAa2nN**



Each customer can have zero or more orders while each order must belong to one customer. This query uses the LEFT JOIN clause to find all customers and their orders:

```
SELECT
    customers.customerNumber,
    customerName,
    orderNumber,
    status
FROM
    customers
LEFT JOIN orders ON
    orders.customerNumber = customers.customerNumber;
```

Result:



customerNumber	customerName	orderNumber	status
166	Handji Gifts& Co	10288	Shipped
166	Handji Gifts& Co	10409	Shipped
167	Herkku Gifts	10181	Shipped
167	Herkku Gifts	10188	Shipped
167	Herkku Gifts	10289	Shipped
168	American Souvenirs Inc	NULL	NULL
169	Porto Imports Co.	NULL	NULL
171	Daedalus Designs Imports	10180	Shipped
171	Daedalus Designs Imports	10224	Shipped
172	La Corne D'abondance, ...	10114	Shipped

In this example:

- The **customers** is the left table and orders is the right table.
- The **LEFT JOIN** clause returns all customers including the customers who have no order. If a customer has no order, the values in the column **orderNumber** and status are **NULL**.

Because both table customers and orders have the same column name (**customerNumber**) in the join condition with the equal operator, you can use the USING syntax as follows:

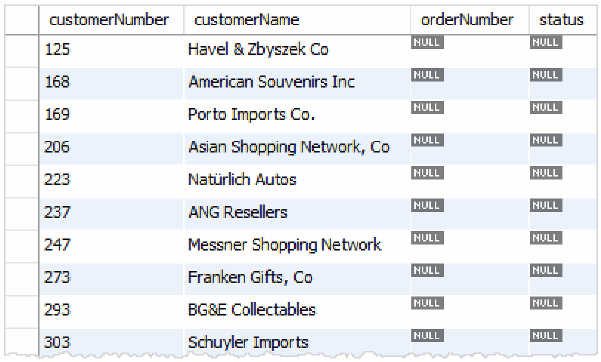
```
SELECT
            customerNumber,
            customerName,
            orderNumber,
            status
FROM
            customers
LEFT JOIN orders USING (customerNumber);
```

Other Example: Using MySQL LEFT JOIN clause to find unmatched rows.

The LEFT JOIN clause is very useful when you want to find rows in a table that doesn't have a matching row from another table. The following example uses the LEFT JOIN to find customers who have no order:

```
SELECT
   c.customerNumber,
   c.customerName,
   o.orderNumber,
   o.status
FROM
   customers as c
LEFT JOIN orders as o
   ON c.customerNumber = o.customerNumber
WHERE
   orderNumber IS NULL;
```

Result:



customerNumber	customerName	orderNumber	status
125	Havel & Zbyszek Co	NULL	NULL
168	American Souvenirs Inc	NULL	NULL
169	Porto Imports Co.	NULL	NULL
206	Asian Shopping Network, Co	NULL	NULL
223	Natürlich Autos	NULL	NULL
237	ANG Resellers	NULL	NULL
247	Messner Shopping Network	NULL	NULL
273	Franken Gifts, Co	NULL	NULL
293	BG&E Collectables	NULL	NULL
303	Schuyler Imports	NULL	NULL

Lesson 4. MySQL RIGHT JOIN clause

The right join clause is similar to the left join clause except that the treatment of tables is reversed. The right join starts selecting data from the right table instead of the left table. The right join clause selects all rows from the right table and matches rows in the left table. If a row from the right table does not have matching rows from the left table, the column of the left table will have NULL in the final result set (*MySQL Join Made Easy For Beginners*, n.d.).

Here is the syntax of the right join:

```
SELECT
   select_last
FROM t1
RIGHT JOIN t2 ON
   join_condition;
```

In this syntax:

- t1 is the left table and t2 is the right table
- join_condition specifies the rule for matching rows in both tables.

If the join_condition uses the equal operator (=) and the joined columns of both tables have the same name, you can use the USING syntax:

```
SELECT
   select_last
FROM t1
RIGHT JOIN t2 USING(column_name);
```

The RIGHT JOIN starts selecting data from the right table (t2). It matches each row from the right table with every row from the left table. If both rows cause the join condition to evaluate to TRUE, it combines columns into a new row and includes this new row in the result set. If a row from the right table does not have a matching row from the left table, it combines columns of rows from the right table with NULL values for all columns of the right table into a new row and includes this row in the result set (*MySQL RIGHT JOIN Explained By Practical Examples*, n.d.).

In other words, the RIGHT JOIN returns all rows from the right table regardless of having matching rows from the left table or not. It's important to emphasize that RIGHT JOIN and LEFT JOIN clauses are functionally equivalent and they can replace each other as long as the table order is reversed (*MySQL RIGHT JOIN Explained By Practical Examples*, n.d.).

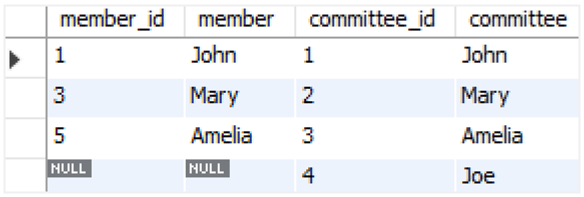
Note that the RIGHT OUTER JOIN is a synonym for RIGHT JOIN

Example:

This statement uses the right join to join the members and committees tables:

```
SELECT
   m.member_id,
   m.name as member,
   c.committee_id,
   c.name as committee
FROM
   members as m
RIGHT JOIN committees c on c.name = m.name;
```

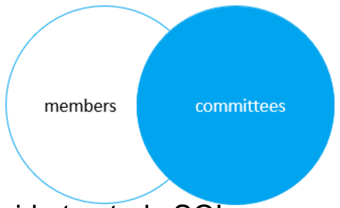
Result:



member_id	member	committee_id	committee
1	John	1	John
3	Mary	2	Mary
5	Amelia	3	Amelia
NULL	NULL	4	Joe

This Venn diagram illustrates the right join:

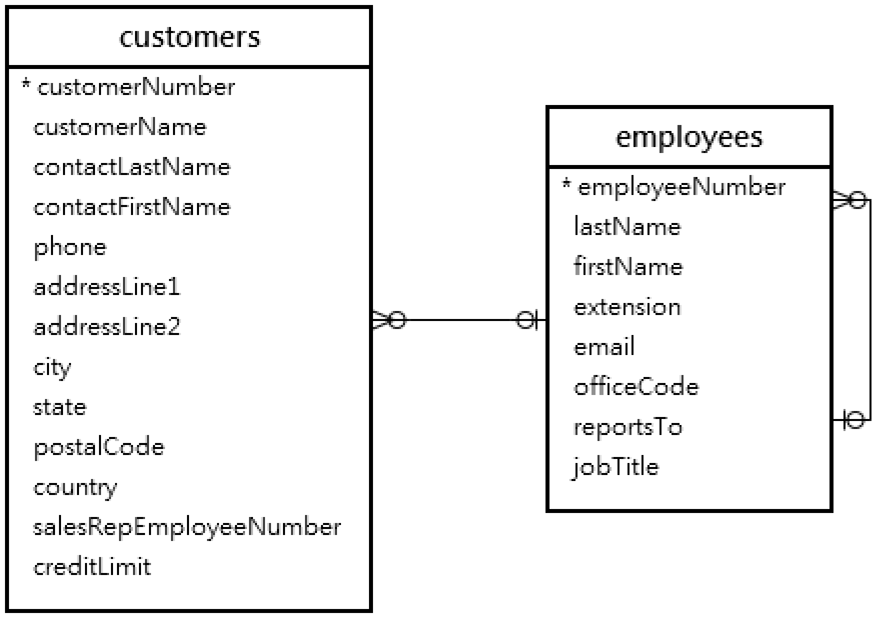
Figure 1.4 Right Join Result Diagram



Source :SQL for beginners: A guide to study SQL programming and database management systems

Other Examples:

We'll use the tables employees and customers from the sample database for the demonstration. You could get the sample database from this link: https://bit.ly/3fAa2nN

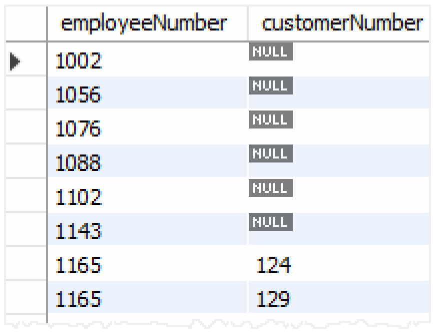


The column **salesRepEmployeeNumber** in the table **customers** links to the column **employeeNumber** in the employees table. A sales representative, or an employee, may in charge of zero or more customers. And each customer is taken care of by zero or one sales representative. If the value in the column **salesRepEmployeeNumber** is NULL, it means the customer does not have any sales representative.

This statement uses the RIGHT JOIN clause join the table customers with the table employees.

```
SELECT
   employeeNumber,
   customerNumber
FROM
   customers
RIGHT JOIN employees
   ON salesRepEmployeeNumber = employeeNumber
ORDER BY
```

```
employeeNumber;
```



employeeNumber	customerNumber
1002	NULL
1056	NULL
1076	NULL
1088	NULL
1102	NULL
1143	NULL
1165	124
1165	129

Result:

In this example:

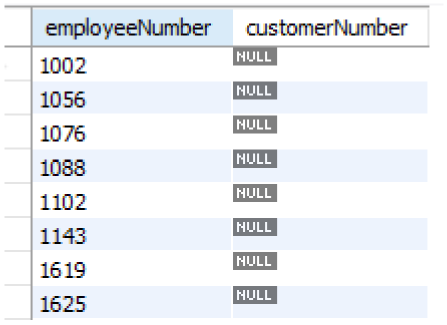
- The RIGHT JOIN returns all rows from the table employees whether rows in the table employees have matching values in the column salesRepEmployeeNumber of the table customers.
- If a row from the table employees has no matching row from the table customers, the RIGHT JOIN uses NULL for the customerNumber column

Another Example: Using MySQL RIGHT JOIN to find unmatching rows

The following statement uses the RIGHT JOIN clause to find employees who do not in charge of any customers:

```
SELECT
   employeeNumber,
   customerNumber
FROM
   customers
RIGHT JOIN employees ON
          salesRepEmployeeNumber = employeeNumber
WHERE customerNumber is NULL
ORDER BY employeeNumber;
```

Result:



employeeNumber	customerNumber
1002	NULL
1056	NULL
1076	NULL
1088	NULL
1102	NULL
1143	NULL
1619	NULL
1625	NULL

Assessment Task

For these assessments, you will use the “classicmodels” database. In case you don’t have it, download it from the link provided. https://bit.ly/3fAa2nN

1. Create a SELECT statement that will output the **ordernumber**, **productname** and **productvendor** from the **classicmodels** database. Display the result in descending order base from the **ordernumer**. (Note: The products table must be the MAIN TABLE / LEFT TABLE)

2. Create a SELECT statement that will list all the **productname** and the **total sales** it has from the classicmodels database. Display the result in highest to lowest in **sales**. (Note: The products table must be the MAIN TABLE / LEFT TABLE)

3. Create a SELECT statement that will list all the **productname** , **total sales,** and the **Total Amount** sold from the classicmodels database. Display the result in ascending order base from the **productname**. (Note: The products table must be the MAIN TABLE / LEFT TABLE)

4. Create a SELECT statement that will list the TOP 10 spender customers with their **Customer Number**, **Customer Name**, and their total **payment.** Display the result from highest to lowest amount **payment**. (Note: The customers table must be the MAIN TABLE / LEFT TABLE)

5. Create a SELECT statement that will list all the customers without any payment made with their **Customer Number**, **Customer Name**. Display the result from A-Z based from the Customer Name. (Note: The customers table must be the MAIN TABLE / LEFT TABLE)

Summary

The data you retrieve depends on how that data is related and on how you define the join in the SELECT statement. MySQL provides several methods for accessing multiple tables in your SQL statements. These include joins, subqueries, and unions. In this chapter, you learned how to use all three methods to access data in multiple tables.

- Create basic, inner, and straight joins that retrieve matched data from two or more tables
- Create left and right joins that retrieve matched data from two or more tables, plus additional data from one of the joined tables
- Create natural, full, and outer joins that automatically match data from one or more tables without having to specify a join condition

References

Cabral, S., & Murphy, K. (2009). *MySQL ® Administrator's Bible*. Wiley Publishing, Inc.

First Normal Form (1NF) of Database Normalization. (n.d.). Retrieved November 28, 2020, from https://www.studytonight.com/dbms/first-normal-form.php

Gupta, S. B., & Mittal, A. (2017). *Introdution to Database Management System* (2nd ed.). University Science Press. https://doi.org/10.1201/9781315372679-7

Lemahieu, W., Broucke, S. vanden, & Baesens, B. (2018). *Principles of Database Management*. Cambridge University Press. https://doi.org/10.16309/j.cnki.issn.1007-1776.2003.03.004

MySQL INNER JOIN By Practical Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-inner-join.aspx

MySQL Join Made Easy For Beginners. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-join/

MySQL RIGHT JOIN Explained By Practical Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-right-join/

Second Normal Form (2NF) of Database Normalization. (n.d.). Retrieved November 28, 2020, from https://www.studytonight.com/dbms/second-normal-form.php

Sheldon, R., & Moes, G. (2005). *Beginning MySQL* (1st ed.). Wrox.

Understanding MySQL LEFT JOIN Clause By Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-left-join.aspx

MODULE 2
DATABASE NORMALIZATION

Introduction

Normalization of a relational model is a process of analyzing the given relations to ensure they do not contain any redundant data. The goal of normalization is to ensure that no anomalies can occur during data insertion, deletion, or update. A step-by-step procedure needs to be followed to transform an unnormalized relational model to a normalized relational model. In what follows, we start by discussing the data anomalies that can occur when working with an unnormalized relational model. Next, we outline some informal normalization guidelines. This is followed by defining two concepts that are fundamental building blocks of a normalization procedure: functional dependencies and prime attribute types (Lemahieu et al., 2018).

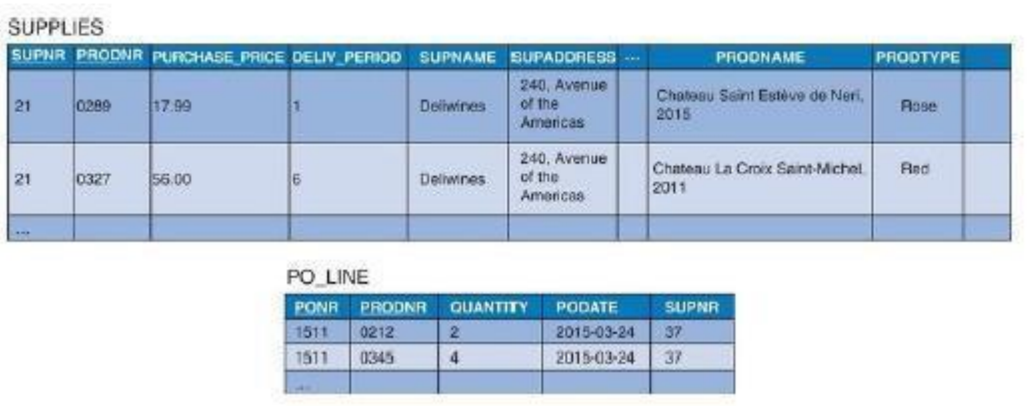
Learning Outcomes

At the end of this module, students should be able to:

1. Normalize a database.
2. Eliminate different types of database anomaly.
3. Design a normalized database.

Lesson 1. Anomalies in an Unnormalized Relational Model

Figure 2.1 shows an example of a relational data model in which we only have two relations with all the information. The SUPPLIES relation also includes all the attribute types for SUPPLIER, such as supplier name, supplier address, etc., and all the attribute types for PRODUCT, such as product number, product type, etc. You can also see that the PO_LINE relation now includes purchase order date and supplier number. Both relations contain duplicate information that may easily lead to inconsistencies. In the SUPPLIES table, for example, all supplier and product information is repeated for each tuple, which creates a lot of redundant information. Because of this redundant information, this relational model is called an unnormalized relational model. At least three types of anomaly may arise when working with an unnormalized relational model: an insertion anomaly, a deletion anomaly, and an update anomaly (Lemahieu et al., 2018).



SUPPLIES

SUPNR	PRODNR	PURCHASE_PRICE	DELIV_PERIOD	SUPNAME	SUPADDRESS	...	PRODNAME	PRODTYPE	
21	0289	17.99	1	Deliwines	240, Avenue of the Americas		Chateau Saint Estève de Neri, 2015	Rose	
21	0327	56.00	6	Deliwines	240, Avenue of the Americas		Chateau La Croix Saint-Michel, 2011	Red	
...									

PO_LINE

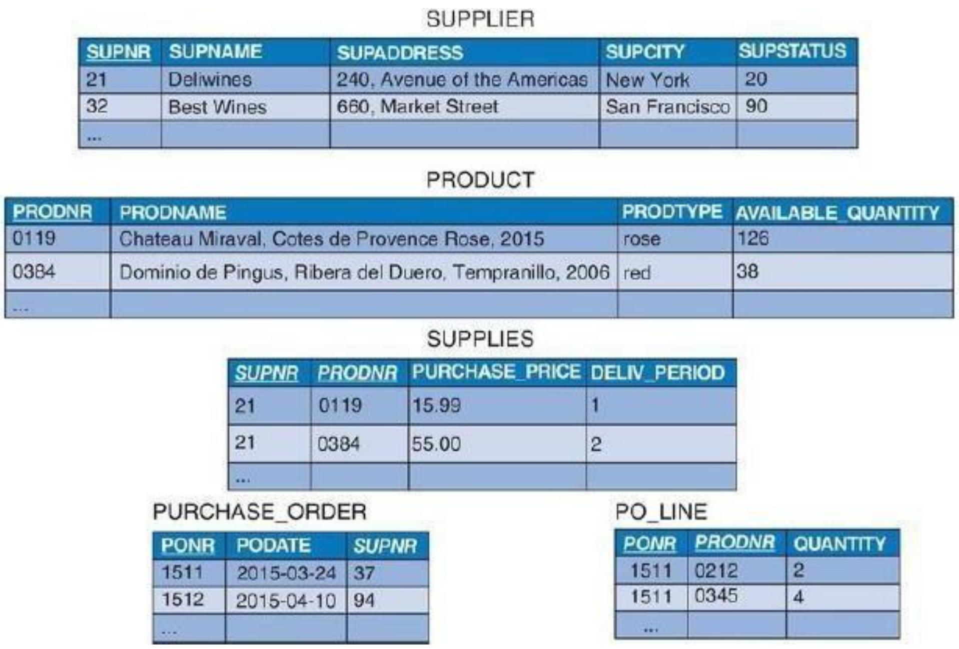
PONR	PRODNR	QUANTITY	PODATE	SUPNR
1511	0212	2	2015-03-24	37
1511	0345	4	2015-03-24	37
...				

Figure 2.1 Unnormalized relational data model

Source: Lemahieu et al., (2018)

An insertion anomaly can occur when we wish to insert a new tuple in the SUPPLIES relation. We must then be sure to each time include the correct supplier (e.g., SUPNR, SUPNAME, SUPADDRESS, etc.) and product (e.g., PRODNR, PRODNAME, PRODTYPE, etc.) information. Furthermore, in this unnormalized relational model, it is difficult to insert a new product for which there are no suppliers yet, or a new supplier who does not supply anything yet since the primary key is a combination of SUPNR and PRODNR, which can thus both not be NULL (entity integrity constraint). A deletion anomaly can occur if we were to delete a particular supplier from the SUPPLIES relation. Consequently, all corresponding product data may get lost as well, which is not desirable. An update anomaly can occur when we wish to update the supplier address in the SUPPLIES relation. This would necessitate multiple updates with the risk of inconsistency (Lemahieu et al., 2018).

Figure 2.2 shows another example relational data model and state for the purchase order administration. Let’s see how our insertion, deletion, and update operations work out here. Inserting a new tuple in the SUPPLIES relation can be easily done since the supplier name, address, etc. and the product name, product type, etc. are only stored once in the relations SUPPLIER and PRODUCT. Inserting a new product for which there are no supplies yet, or a new supplier who does not supply anything yet, can be accomplished by adding new tuples to the PRODUCT and SUPPLIER relation. Deleting a tuple from the SUPPLIER relation will not affect any product tuples in the PRODUCT relation. Finally, if we wish to update the supplier address, we only need to do one single update in the SUPPLIER table. As there are no inconsistencies or duplicate information in this relational model, it is also called a normalized relational model (Lemahieu et al., 2018).



SUPPLIER

SUPNR	SUPNAME	SUPADDRESS	SUPCITY	SUPSTATUS
21	Deliwines	240, Avenue of the Americas	New York	20
32	Best Wines	660, Market Street	San Francisco	90
...				

PRODUCT

PRODNR	PRODNAME	PRODTYPE	AVAILABLE_QUANTITY
0119	Chateau Miraval, Cotes de Provence Rose, 2015	rose	126
0384	Dominio de Pingus, Ribera del Duero, Tempranillo, 2006	red	38
...			

SUPPLIES

SUPNR	PRODNR	PURCHASE_PRICE	DELIV_PERIOD
21	0119	15.99	1
21	0384	55.00	2
...			

PURCHASE_ORDER

PONR	PODATE	SUPNR
1511	2015-03-24	37
1512	2015-04-10	94
...		

PO_LINE

PONR	PRODNR	QUANTITY
1511	0212	2
1511	0345	4
...		

Figure 2.2 Normalized relational data model

Source: Lemahieu et al., (2018)

To have a good relational data model, all relations in the model should be normalized. A formal normalization procedure can be applied to transform an unnormalized relational model into a normalized form (Lemahieu et al., 2018)

Lesson 2. Functional Dependencies and Prime Attribute Types

Before we can start discussing various normalization steps, we need to introduce two important concepts: functional dependency and prime attribute type. A functional dependency X → Y between two sets of attribute types X and Y implies that a value of X uniquely determines a value of Y. We also say that there is a functional dependency from X to Y or that Y is functionally dependent on X. As an example, the employee name is functionally dependent upon the social security number (Lemahieu et al., 2018):

SSN → ENAME

In other words, a social security number uniquely determines an employee name. The other way around does not necessarily apply, since multiple employees can share the same name, hence one employee name may correspond to multiple social security numbers. A project number uniquely determines a project name and a project location (Lemahieu et al., 2018):

PNUMBER → {PNAME, PLOCATION}

Project name and project location are thus functionally dependent upon project number. The number of hours an employee worked on a project is functionally dependent upon both the social security number and the project number (Lemahieu et al., 2018):

{SSN, PNUMBER} → HOURS

Note that if X is a candidate key of a relation R, this implies that Y is functionally dependent on X for any subset of attribute types Y of R (Lemahieu et al., 2018).

A prime attribute type is another important concept that is needed in the normalization process. A prime attribute type is an attribute type that is part of a candidate key. Consider the following relation (Lemahieu et al., 2018):

R1(SSN, PNUMBER, PNAME, HOURS)

The key of the relation is a combination of SSN and PNUMBER. Both SSN and PNUMBER are prime attribute types, whereas PNAME and HOURS are non-prime attribute types (Lemahieu et al., 2018).

Armstrong's Axioms

If F is a set of functional dependencies then the closure of F, denoted as F+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are a set of rules that, when applied repeatedly, generates a closure of functional dependencies.

- **Reflexive rule**: If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.
- **Augmentation rule**: If a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
- **Transitivity rule**: Same as transitive rule in algebra, if a → b holds and b → c holds, then

  a → c also holds. a → b is called as a functionally that determines b.

Trivial Functional Dependency

- **Trivial**: If a functional dependency (FD) X → Y holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.
- **Non-trivial:** If an FD X → Y holds, where Y is not a subset of X, then it is called a non-trivial FD.
- **Completely non-trivial**: If an FD X → Y holds, where x intersect Y = Φ, it is said to be a completely non-trivial FD.

Lesson 3. First Normal Form (1 NF)

The first normal form (1 NF) states that every attribute type of a relation must be atomic and single-valued. Hence, no composite or multi-valued attribute types are tolerated. This is the same as the domain constraint we introduced earlier. Consider the following example (Lemahieu et al., 2018):

SUPPLIER(SUPNR, NAME(FIRST NAME, LAST NAME), SUPSTATUS)

This relation is not in 1 NF as it contains a composite attribute type NAME that consists of the attribute types FIRST NAME and LAST NAME. We can bring it in 1 NF as follows (Lemahieu et al., 2018):

SUPPLIER(SUPNR, FIRST NAME, LAST NAME, SUPSTATUS)

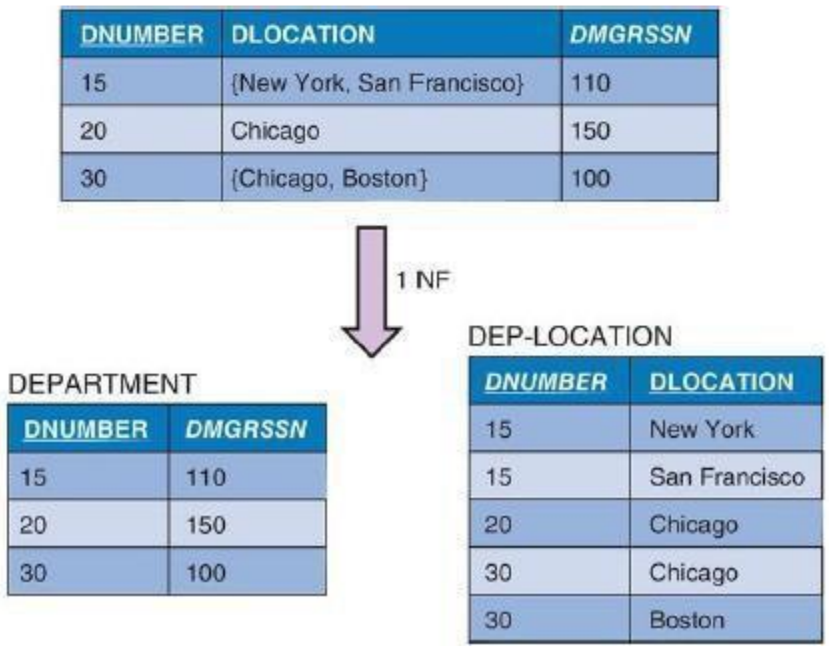
In other words, composite attribute types need to be decomposed in their parts to bring the relation in 1 NF. Suppose we have a relation DEPARTMENT. It has a department number, a department location and a foreign key referring to the social security number of the employee who manages the department (Lemahieu et al., 2018):

DEPARTMENT(DNUMBER, DLOCATION, DMGRSSN)

Assume now that a department can have multiple locations and that multiple departments are possible at a given location. The relation is not in 1 NF since DLOCATION is a multi-valued attribute type. We can bring it in 1 NF by removing DLOCATION from department and putting it into a new relation DEP-LOCATION together with DNUMBER as a foreign key (Lemahieu et al., 2018):

DEPARTMENT(DNUMBER, DMGRSSN)
DEP-LOCATION(DNUMBER, DLOCATION)

The primary key of this new relation is then the combination of both, since a department can have multiple locations and multiple departments can share a location. Figure 2.3 illustrates some example tuples. You can see that department number 15 has two locations: New York and San Francisco. Department number 30 also has two locations: Chicago and Boston. The two lower tables bring it in the 1 NF since every attribute type of both relations is now atomic and single-valued. To summarize, multi-valued attribute types (e.g., DLOCATION) should be removed and put in a separate relation (e.g., DEPLOCATION) along with the primary key (e.g., DNUMBER) of the original relation (e.g., DEPARTMENT) as a foreign key. The primary key of the new relation is then the combination of the multi-valued attribute type and the primary key of the original relation (e.g., DNUMBER and DLOCATION) (Lemahieu et al., 2018).



DNUMBER	DLOCATION	DMGRSSN
15	{New York, San Francisco}	110
20	Chicago	150
30	{Chicago, Boston}	100

DEPARTMENT

DNUMBER	DMGRSSN
15	110
20	150
30	100

DEP-LOCATION

DNUMBER	DLOCATION
15	New York
15	San Francisco
20	Chicago
30	Chicago
30	Boston

Figure 2.3 First normal form

Source: Lemahieu et al., (2018).

The unnormalized relation (above) is decomposed into two relations (below) by ensuring there are no composite or multi-valued attribute types (Lemahieu et al., 2018).

Let's give another example. Say we have a relation R1 with employee information such as SSN, ENAME, DNUMBER, DNAME, and PROJECT, which is a composite attribute type consisting of PNUMBER, PNAME and HOURS (Lemahieu et al., 2018):

R1(SSN, ENAME, DNUMBER, DNAME, PROJECT(PNUMBER, PNAME, HOURS))

We assume an employee can work on multiple projects and multiple employees can work on the same project. Hence, we have a multi-valued composite attribute type PROJECT in our relation R1. In other words, both conditions of the first normal form are clearly violated. To bring it in first normal form, we create two relations R11 and R12 where the latter includes the project attribute types together with SSN as a foreign key (Lemahieu et al., 2018):

R11(SSN, ENAME, DNUMBER, DNAME)

R12(SSN, PNUMBER, PNAME, HOURS)

The primary key of R12 is then the combination of SSN and PNUMBER, since an employee can work on multiple projects and multiple employees can work on a project (Lemahieu et al., 2018).

Rules for First Normal Form

The first normal form expects you to follow a few simple rules while designing your database, and they are:

Rule 1: Single Valued Attributes

Each column of your table should be single valued which means they should not contain multiple values. We will explain this with help of an example later, let's see the other rules for now (*First Normal Form (1NF) of Database Normalization*, n.d.).

Rule 2: Attribute Domain should not change

This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type.

For example: If you have a column dob to save date of births of a set of people, then you cannot or you must not save 'names' of some of them in that column along with 'date of birth' of others in that column. It should hold only 'date of birth' for all the records/rows (*First Normal Form(1NF)ofDatabaseNormalization*, n.d.).

Rule 3: Unique name for Attributes/Columns

This rule expects that each column in a table should have a unique name. This is to avoid confusion at the time of retrieving data or performing any other operation on the stored data. If one or more columns have same name, then the DBMS system will be left confused (*First Normal Form (1NF) of Database Normalization*, n.d.).

Rule 4: Order doesn't matters

This rule says that the order in which you store the data in your table doesn't matter (*First Normal Form (1NF) of Database Normalization*, n.d.).

Lesson 4. Second Normal Form (2 NF)

Before we can start discussing the second normal form (2 NF), we need to introduce the concepts of full and partial functional dependency. A functional dependency X → Y is a full functional dependency if removal of any attribute type A from X means that the dependency does not hold anymore. For example, HOURS is fully functionally dependent upon both SSN and PNUMBER (Lemahieu et al., 2018):

SSN, PNUMBER → HOURS

More specifically, to know the number of hours an employee worked on a project, we need to know both the SSN of the employee and the project number. Likewise, project name is fully functionally dependent upon project number (Lemahieu et al., 2018):

PNUMBER → PNAME

A functional dependency X → Y is a partial dependency if an attribute type A from X can be removed from X and the dependency still holds. As an example, PNAME is partially functionally dependent upon SSN and PNUMBER (Lemahieu et al., 2018):

SSN, PNUMBER → PNAME

It only depends upon PNUMBER, not on SSN.

A relation R is in the 2 NF if it satisfies 1 NF and every non-prime attribute type A in R is fully functionally dependent on any key of R. In case the relation is not in second normal form, we must decompose it and set up a new relation for each partial key together with its dependent attribute types. Also, it is important to keep a relation with the original primary key and any attribute types that are fully functionally dependent on it. Let’s illustrate this with an example. Say we have a relation R1 with attribute types SSN, PNUMBER, PNAME, HOURS (Lemahieu et al., 2018):

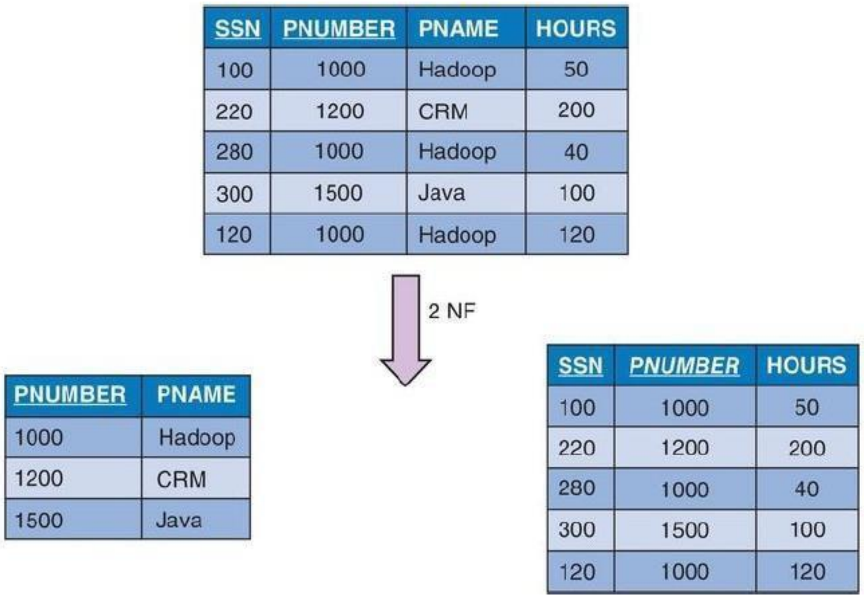
R1(SSN, PNUMBER, PNAME, HOURS)

It contains both project information and information about which employee worked on what project for how many hours. The assumptions are as follows: an employee can work on multiple projects; multiple employees can work on the same project; and a project has a unique name. The relation R1 is in 1 NF since there are no multi-valued or composite attribute types. However, it is not in 2 NF. The primary key of the relation R1 is a combination of SSN and PNUMBER. The attribute type PNAME is not fully functionally dependent on the primary key, it only depends on PNUMBER. HOURS, however, is fully functionally dependent upon both SSN and PNUMBER. Hence, we need to remove the attribute type PNAME and put it in a new relation R12, together with PNUMBER (Lemahieu et al., 2018):

R11(SSN, PNUMBER, HOURS)
R12(PNUMBER, PNAME)

The relation R11 can then be called WORKS-ON(SSN, PNUMBER, HOURS) and the relation R12 can be referred to as PROJECT(PNUMBER, PNAME). (Lemahieu et al., 2018).

Figure 2.4 illustrates how to bring a relation into 2 NF with some example tuples. Note the redundancy in the original relation. The name "Hadoop" is repeated multiple times, which is not desirable from a storage perspective. Also, if we would like to update it (e.g., from "Hadoop" to "Big Data"), then multiple changes need to take place. This is not the case for the two normalized relations at the bottom, where the update should only be done once in the PROJECT relation.



SSN	PNUMBER	PNAME	HOURS
100	1000	Hadoop	50
220	1200	CRM	200
280	1000	Hadoop	40
300	1500	Java	100
120	1000	Hadoop	120

PNUMBER	PNAME
1000	Hadoop
1200	CRM
1500	Java

SSN	PNUMBER	HOURS
100	1000	50
220	1200	200
280	1000	40
300	1500	100
120	1000	120

Figure 2.4 Second normal form

Source: Lemahieu et al., (2018).

The unnormalized relation (above) is decomposed into two relations (below) by ensuring every non-prime attribute type is fully functionally dependent on the primary key (Lemahieu et al., 2018).

For a table to be in the Second Normal Form, it must satisfy two conditions (*Second Normal Form (2NF) of Database Normalization*, n.d.):

- The table should be in the First Normal Form.
- There should be no Partial Dependency

For additional information and materials: https://www.studytonight.com/dbms/second-normal-form.php

Lesson 5. Third Normal Form (3 NF)

Third Normal Form is an upgrade to Second Normal Form. When a table is in the Second Normal Form and has no transitive dependency, then it is in the Third Normal Form

To discuss the third normal form (3 NF), we need to introduce the concept of transitive dependency. A functional dependency $X \rightarrow Y$ in a relation R is a transitive dependency if there is a set of attribute types Z that is neither a candidate key nor a subset of any key of R, and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. A relation is in the 3 NF if it satisfies 2 NF and no non-prime attribute type of R is transitively dependent on the primary key. If this is not the case, we need to decompose the relation R and set up a relation that includes the non-key attribute types that functionally determine the other non-key attribute types. Let's work out an example to illustrate this. The relation R1 contains information about employees and departments as follows (Lemahieu et al., 2018):

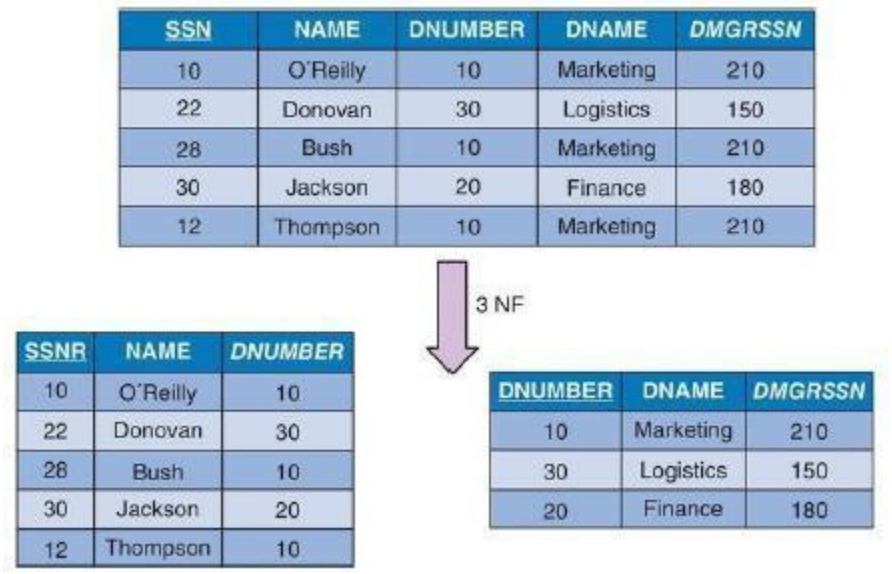
R1(SSN, ENAME, DNUMBER, DNAME, DMGRSSN)

The SSN attribute type is the primary key of the relation. The assumptions are as follows: an employee works in one department; a department can have multiple employees; and a department has one manager. Given these assumptions, we have two transitive dependencies in R. DNAME is transitively dependent on SSN via DNUMBER. In other words, DNUMBER is functionally dependent on SSN, and DNAME is functionally dependent on DNUMBER. Likewise, DMGRSSN is transitively dependent on SSN via DNUMBER. In other words, DNUMBER is functionally dependent on SSN and DMGRSSN is functionally dependent on DNUMBER. DNUMBER is not a candidate key nor a subset of a key. Hence, the relation is not in 3 NF. To bring it in 3 NF, we remove the attribute types DNAME and DMGRSSN and put them in a new relation R12 together with DNUMBER as its primary key (Lemahieu et al., 2018):

R11(SSN, ENAME, DNUMBER)
R12(DNUMBER, DNAME, DMGRSSN)

The relation R11 can be called EMPLOYEE(SSN, ENAME, DNUMBER) and the relation R12 can be referred to as DEPARTMENT(DNUMBER, DNAME, DMGRSSN)

Figure 2.5 shows some example tuples for both the unnormalized and normalized relations. Note the redundancy in the unnormalized case, where the values “marketing” for DNAME and “210” for DMGRSSN are repeated multiple times. This is not the case for the normalized relations where these values are only stored once.



SSN	NAME	DNUMBER	DNAME	DMGRSSN
10	O'Reilly	10	Marketing	210
22	Donovan	30	Logistics	150
28	Bush	10	Marketing	210
30	Jackson	20	Finance	180
12	Thompson	10	Marketing	210

SSNR	NAME	DNUMBER
10	O'Reilly	10
22	Donovan	30
28	Bush	10
30	Jackson	20
12	Thompson	10

DNUMBER	DNAME	DMGRSSN
10	Marketing	210
30	Logistics	150
20	Finance	180

Figure 2.5 Third normal form

Source: Lemahieu et al., (2018).

The unnormalized relation (above) is decomposed into two relations (below) by ensuring no non-prime attribute types are transitively dependent on the primary key (Lemahieu et al., 2018).

For a table to be in the third normal form:

- It should be in the Second Normal form.
- And it should not have Transitive Dependency.

For additional information and materials: https://www.studytonight.com/dbms/third-normal-form.php

Lesson 6. Boyce–Codd Normal Form (BCNF)

We can now discuss the Boyce–Codd normal form (BCNF), also referred to as the 3.5 normal form (3.5 NF). Let's first introduce another concept. A functional dependency X → Y is called a trivial functional dependency if Y is a subset of X. An example of a trivial functional dependency is between SSN and NAME, and SSN (Lemahieu et al., 2018):

SSN, NAME → SSN

A relation R is in the BCNF provided that for each of its non-trivial functional dependencies X → Y, X is a superkey – that is, X is either a candidate key or a superset thereof. It can be shown that the BCNF is stricter than the 3 NF. Hence, every relation in BCNF is also in 3 NF. However, a relation in 3 NF is not necessarily in BCNF. Let's give an example. Suppose we have a relation R1 with attribute types SUPNR, SUPNAME, PRODNR, and QUANTITY (Lemahieu et al., 2018):

R1(SUPNR, SUPNAME, PRODNR, QUANTITY)

It models information about which supplier can supply what products in what quantities. The assumptions are as follows: a supplier can supply multiple products; a product can be supplied by multiple suppliers; and a supplier has a unique name. Therefore, SUPNR and PRODNR are a superkey of the relation. Further, we have a non-trivial functional dependency between SUPNR and SUPNAME. The relation is thus not in BCNF. To bring it in BCNF we remove SUPNAME from R1 and put it in a new relation R12 together with SUPNR as the primary key (Lemahieu et al., 2018):

R11(SUPNR, PRODNR, QUANTITY)
R12(SUPNR, SUPNAME)

The relation R11 can be called SUPPLIES and the relation R12 can be referred to as SUPPLIER (Lemahieu et al., 2018).

For additional information and materials: https://www.studytonight.com/dbms/boyce-codd-normal-form.php

Assessment Task

Let us assume we have the corresponding table named Table 1.

Customer	Product	Shipping_Address	Newsletter	Supplier	Supplier_Phone	Price
James	Redmi Note 8	Santa Cruz	Mi News	Xiaomi	CALL-MI	13,000
Maybelle Cruz	Realme 5 Pro	Bay	Be Real	Realme	BE-REAL	15,000
Kimberly Doe	MI 10T	Bay	Mi News	Xiaomi	CALL-MI	19,999
Juan dela Cruz	Realme 8	Pila	Be Real	Realme	BE-REAL	12,000
Kimberly Lopez	Galaxy Note 5	Duhat	Premium News	Samsung	CALL-SAM	25,000
Jane De Guzman	MI 10T	Pila	Mi News	Xiaomi	CALL-MI	19,999
Juan dela Cruz	Redmi NOTE 9S	Pila	Mi News	Xiaomi	CALL-MI	10,000
Maybelle Cruz	Realme 7	Bay	Be Real	Realme	BE-REAL	9,000
Maybelle Cruz	Realme 7i	Santa Cruz	Be Real	Realme	BE-REAL	11,000
Jane De Guzman	Redmi Note 8	Pila	Mi News	Xiaomi	CALL-MI	13,000

1. Normalize the table 1 into 1st Normal Form. (Hint: 1 table, 8 columns)
2. Normalize the table 1 into 2nd Normal Form (Hint: 3 tables)
3. Normalize the table 1 into 3rd Normal Form (Hint: 4 tables)

Summary

- Update anomalies: If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.

- Deletion anomalies: We tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else.
- Insert anomalies: We tried to insert data in a record that does not exist atall.

- A relation (table) is in 1NF if (and only if) the domain of each attribute contains only atomic (indivisible) values, and the value of each attribute contains only a single value from that domain.
- A relation is in 2NF if it is in 1NF and every non-prime attribute of the relation is dependent on the whole of every candidate key.
- A relation is in 3NF if it is in 2NF and every non-prime attribute of the relation is non-transitively dependent on every key of the relation.

References

(Ramakrishhan, Raghu and Gehrke,Johannes,2018):

Source :SQL for beginners: A guide to study SQL programming and database management systems

First Normal Form (1NF) of Database Normalization. (n.d.). Retrieved November 28, 2020, from https://www.studytonight.com/dbms/first-normal-form.php

Gupta, S. B., & Mittal, A. (2017). Introdution to Database Management System (2nd ed.). University Science Press. https://doi.org/10.1201/9781315372679-7

Lemahieu, W., Broucke, S. vanden, & Baesens, B. (2018). Principles of Database Management. Cambridge University Press. https://doi.org/10.16309/j.cnki.issn.1007-1776.2003.03.004

MySQL INNER JOIN By Practical Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-inner-join.aspx

MySQL Join Made Easy For Beginners. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-join/

MySQL RIGHT JOIN Explained By Practical Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-right-join/

Second Normal Form (2NF) of Database Normalization. (n.d.). Retrieved November 28, 2020, from https://www.studytonight.com/dbms/second-normal-form.php

Understanding MySQL LEFT JOIN Clause By Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-left-join.aspx

MODULE 3
TRANSACTIONS AND CONCURRENCY CONTROL

Introduction

Transaction is a logical unit of work that represents the real-world events. A transaction is also defined as any one execution of a user program in a Database Management System (DBMS). The transaction management is the ability of a database management system to manage the different transactions that occur within it. A DBMS has to interleave the actions of many transactions due to performance reasons. The interleaving is done in such a way that the result of the concurrent execution is equivalent to some serial execution of the same set of transactions (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Learning Outcomes

At the end of this module, students should be able to:

1. Differentiate different concurrency techniques
2. Find solutions to deadlocks encountered in a database
3. Identify transaction problems

Lesson 1. Transaction

A transaction can be defined as a unit or part of any program at the time of its execution. During transactions data items can be read or updated or both. The database system is required to maintain the following properties of transactions to ensure the integrity of the data (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Atomicity: A transaction must be atomic. Atomic transaction means either all operations wit
transaction are completed or none.

Consistency: A transaction must be executed in isolation. It means variables used by a transaction cannot be changed by any other transaction concurrently.

Isolation: During concurrent transaction each transaction must be unaware of other transactions. For any transaction T_i, it appears to T_i that any other transaction T_k is either finished before starting of T_i or started after T_i finished.

Durability: Changes are made permanent to database after successful completion of transaction even in the case of system failure or crash.

A transaction must be in one of the following states as shown in Figure 3.1.

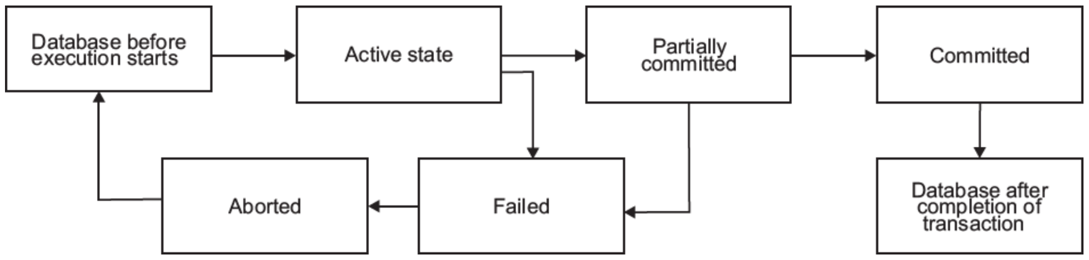

Database before execution starts
Active state
Partially committed
Committed
Aborted
Failed
Database after completion of transaction

Figure 3.1 States of the transaction

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

1. Active state: It is the initial state of transaction. During execution of statements, a transaction is in active state.
2. Partially committed: A transaction is in partially committed state, when all the statements within transaction are executed but transaction is not committed.
3. Failed: In any case, if transaction cannot be proceeded further then transaction is in failed state.
4. Committed: After successful completion of transaction, it is in committed state.

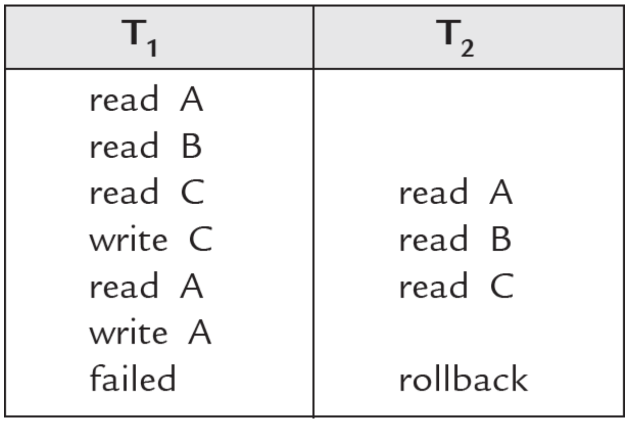
Some Definitions

Serializability - Consider a set of transactions (T_1, T_2, ..., T_i). S_1 is the state of database after they are concurrently executed and successfully completed and S_2 is the state of database after they are executed in any serial manner (one-by-one) and successfully completed. If S_1 and S_2 are same then the database maintains serializability (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Concurrent Execution - If more than one transaction is executed at the same time then they are said to be executed concurrently (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Recoverability - To maintain atomicity of database, undo effects of any transaction have to be performed in case of failure of that transaction. If undo effects successfully then that database maintains recoverability. This process is known as Rollback (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Cascading Rollback - If any transaction T_i is dependent upon T_j and T_j is failed due to any reason then rollback T_i. This is known as cascading rollback. Consider Figure 3.2. Here T_2 is dependent upon T_1 because T_2 reads value of C which is updated by T_1. If T_1 fails then rollback T_1 and also T_2 (Ramakrishhan, Raghu and Gehrke,Johannes,2018).



T_1	T_2
read A	
read B	
read C	read A
write C	read B
read A	read C
write A	
failed	rollback

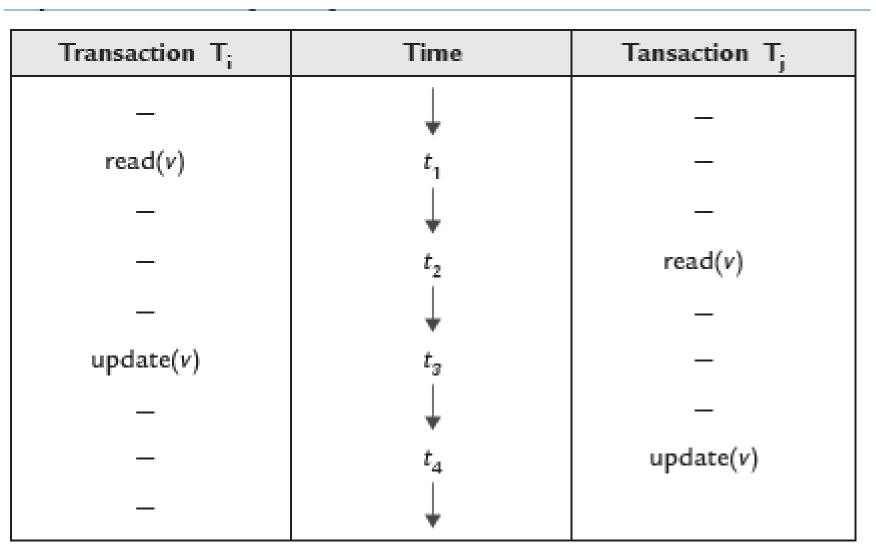
Figure 3.2 Cascading rollback.

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

Lesson 2. Why Concurrency Control is Needed?

To make system efficient and save time, it is required to execute more than one transaction at the same time. But concurrency leads several problems. The three problems associated with concurrency are as follows (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The Lost Update Problem - If any transaction Tj updates any variable v at time t without knowing the value of v at time t then this may lead to lost update problem. Consider thetransactions shown in Figure 3.3



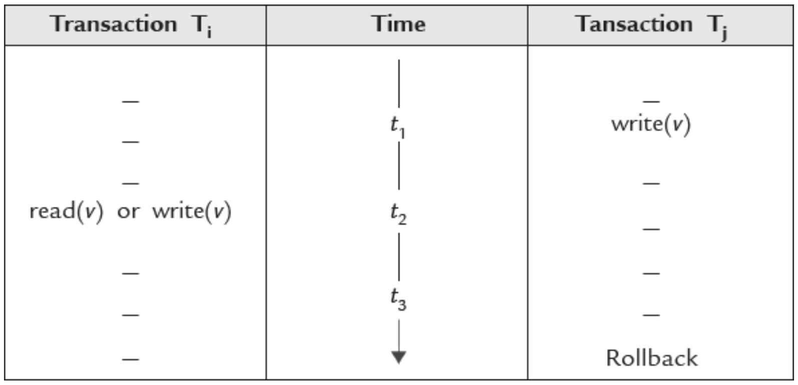
Transaction T_i	Time	Tansaction T_j
—	↓	—
read(*v*)	t_1	—
—	↓	—
—	t_2	read(*v*)
—	↓	—
update(*v*)	t_3	—
—	↓	—
—	t_4	update(*v*)
—	↓	

Figure 3.3 The lost update problem

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

At time t1 and t2, transactions ti and tj reads variable v respectively and get some valueof *v*. At time t3, ti updates *v* but, at t4, tj also updates *v* (old value of *v*) without looking the new value of *v* (updated by ti). So, updation made by ti at t3 is lost at t4 because Tj overwrites it (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The Uncommitted Dependency Problem - This problem arises if any transaction Tj updates any variable *v* and allows retrieval or updation of *v* by any other transaction but Tj rolled back due to failure. Consider the transactions shown in Figure 3.4 (Ramakrishhan, Raghu and Gehrke,Johannes,2018).



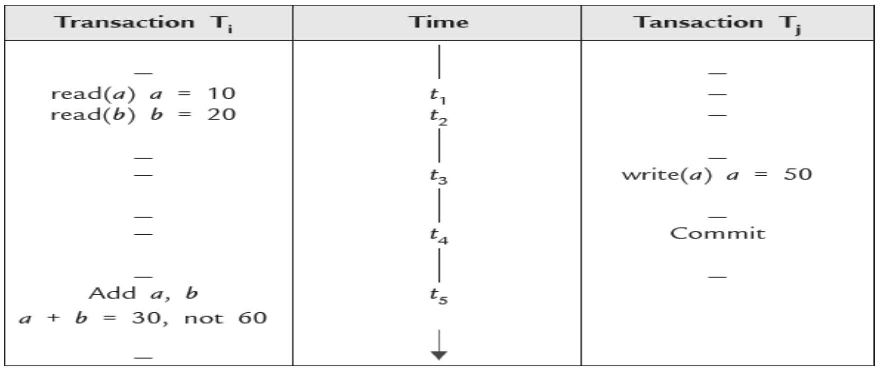
Transaction T_i	Time	Tansaction T_j
—		—
—	t_1	write(v)
—		—
read(v) or write(v)	t_2	—
—		—
—	t_3	—
—	↓	Rollback

Figure 3.4 The uncommitted dependencyproblem)

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

At time t_1, transaction T_j, updates variable v which is read or updated by T_i at time t_2. Suppose at time t_3, T_j is rollbacked due to any reason then result produced by T_i is wrong because it is based on false assumption (Ramakrishhan,Raghuand Gehrke,Johannes,2018).

The Inconsistent Analysis Problem - Consider the transactions shown in Figure 3.5.



Transaction T_i	Time	Tansaction T_j
—		—
read(a) a = 10	t_1	—
read(b) b = 20	t_2	—
—		—
—	t_3	write(a) a = 50
—		—
—	t_4	Commit
—		—
Add a, b	t_5	
a + b = 30, not 60		
—	↓	

Figure 3.5 The inconsistent analysis problem

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

The transaction T_i reads variable a and b at time t_1 and t_2 respectively. But at time t_3, T_j updates value of a from 10 to 50 and commits at t_4 that makes changes permanent. So, addition of a and b at time t_5 gives wrong result. This leads inconsistency in database because of inconsistent analysis by T_i (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Lesson 3. Concurrency Control Techniques

To avoid concurrency related problems and to maintain consistency, some rules (protocols) need to be made so that system can be protected from such situations and increase the efficiency (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Lock-Based Protocols

To maintain consistency, restrict modification of the data item by any other transaction which is presently accessed by some transaction. These restrictions can be applied by applying locks on data items. To access any data item, transaction have to obtain lock on it (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

1. **Granularity of Locks -** The level and type of information that the lock protects is called locking granularity. In other words, the size of data items that the data manager locks is called the locking granularity. The granularity of locks in a database refers to how much of the data is locked at one time. In theory, a database server can lock as much as the entire database or as little as one column of data. The level of locking used depends on the typical access needs of transactions as well as consideration of performance tradeoffs. Locking granularity affects performance since more work is necessary to maintain and coordinate the increased number of locks. To achieve optimum performance, a locking scheme must balance the needs of concurrency and overhead. Locking can occur on the following levels:
   a. **Coarse Granularity (Table or File or Database Locks):** The data manager could lock at a coarse granularity such as tables or files. If the locks are at coarse granularity, the data manager doesn't have to set many locks, because each lock

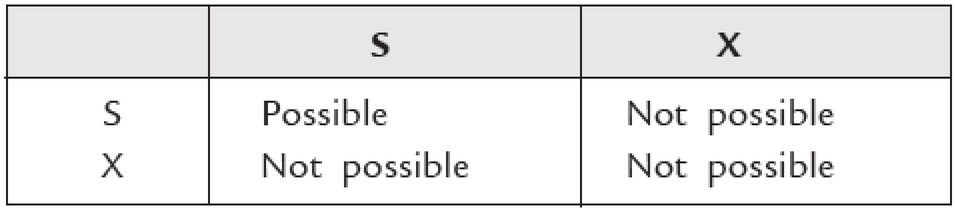
covers a great amount of data. Thus, the overhead of setting and releasing locks is low. However, by locking large chunks of data, the data manager is usually locking more data than a transaction need. For example, even if a transaction T accesses only a few records of a table, a data manager that locks at the granularity of tables will lock the whole table, thereby preventing other transactions from locking any other records of the table, most of which are not needed by transaction T. This reduces the number of transactions that can run concurrently, which both reduces the throughput and increases the response time of transactions (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

b. **Fine Granularity (Records or Fields Locks**): The data manager could lock at a fine granularity, such as records or fields. If the locks are at fine granularity, the data manager only locks the specific data actually accessed by a transaction. These locks do not artificially interfere with other transaction, as coarse grain locks do. However, the data manager must now lock every piece of data accessed by a transaction, which can generate much locking overhead. For example, if a transaction issues an SQL query that accesses tens of thousands of records, a data manager that do record granularity locking would set tens of thousands of locks, which can be quite costly. In addition to the record locks, locks on associated indexes are also needed, which compounds the problem. There is a fundamental tradeoff between amount of concurrency and locking overhead, depending on the granularity of locking. Coarsegrained locking has low overhead but low concurrency. Fine-grained locking has high concurrency but high overhead (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

c. **Intermediate Granularity (Pages Locks):** The data manager could lock at an intermediate granularity, such as pages. A disk-page or page is the equivalent of a disk-block, which may be described as a section of a disk. A page has a fixed size, such as 4K, 8K, 16K, etc. A table may span several pages, and a page may contain several rows of one or more tables. Page-level locks are the most frequently used of the multi-user DBMS locking. This gives a moderate degree of concurrency with a moderate amount of locking overhead. It works well in systems that do not need to run at high transaction rates, and hence are unaffected by the reduced concurrency, or ones where transactions frequently access many records per page so that page locks are not artificially locking more data than transactions actually access. It also simplifies the recovery algorithms for Commit and Abort. However, for high performance Transaction Processing, record locking is needed, because there are too many cases where concurrent transactions need to lock different records on the same page (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

2. **Concurrency Control Manager (CCM) -** The concurrency control manager synchronizes the concurrent access of database transactions to shored objects in the database. It is responsible for maintaining the guarantees regarding the effects of concurrent access to the shored database. It will make sure that the result of transactions running in parallel is identical to the result of some serial execution of the same transactions. The Concurrency Control Manager grant locks to different transactions. Concurrency Control manager is basically a program (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

3. **Types of Lock -** There are two types of locks that can be implemented on a transaction.
   a. **Shared lock** : Shared lock is Read-Only lock. If a transaction T_i has obtained shared lock on data item A then T_i can read A but cannot modify A. It is denoted by S and T_i is said to be in Shared lock mode.
   b. **Exclusive lock** : Exclusive lock is Read-Write lock. If a transaction T_i has obtained exclusive lock on data item A then T_i can both read and modify
4. **Compatible Lock Modes –** Suppose transaction Ti has obtained a lock on data item Vin mode A. If transaction Tj can also obtain lock on V in mode B then, A is compatible with B or these two lock modes are compatible. Matrix of compatibility of shared and Exclusive lock modes are shown in Figure 3.6 (Ramakrishhan, Raghu and Gehrke,Johannes,2018)



	S	X
S	Possible	Not possible
X	Not possible	Not possible

Figure 3.6 Compatibility of shared and exclusive lock mode

Source: Gupta & Mittal (2017)

The table shows that if a transaction T_i has shared lock on data item V than any other transaction T_j can obtain only shared lock on V at the same time. All other possible combinations are incompatible (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

S - lock(A) — Shared lock on data item A

X - lock(A) — Exclusive lock on data item A
read(A) — Read operation on A
write(A) — Write operation on A
unlock(A) — A is free now.

A transaction T_i can hold only those items which are used in it. If any data item V is not free then T_i is made to wait until V is released. It is not desirable to unlock data item immediately after its final access (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The advantages of locking are

1. In maintains serializability.
2. It solves all the concurrency problems.

The disadvantages of locking are:

1. It leads to a new problem that is Deadlock.

5. **Two-phase Locking Protocol -** Two-phase locking protocol guarantees serializability.In this protocol, every transaction issue lock and unlock requests in two phases (Ramakrishhan, Raghu and Gehrke,Johannes,2018)
   a. **Growing phase**: In this phase, transaction can obtain new locks but cannot release any lock.
   b. **Shrinking or contracting phase**: In this phase, transaction can release locks, but cannot obtain new locks.

Lock Point: In growing phase, the point at which transaction has obtained its final lock is called Lock Point.

Basic Two-phase Locking Protocol - The technique described above is known as basic two phase locking Protocol (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Advantages: The advantages of Basic two-phase locking protocol are:

1. It ensures serializability.

Disadvantages: The disadvantages of basic two-phase locking protocol are:

1. It may cause deadlock.
2. It may cause cascading rollback

Strict Two-phase Locking Protocol - In strict two-phase locking protocol, all exclusive locks are kept until the end of transaction or until the transaction commits (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Advantages: The advantages of strict 2PL protocol are:

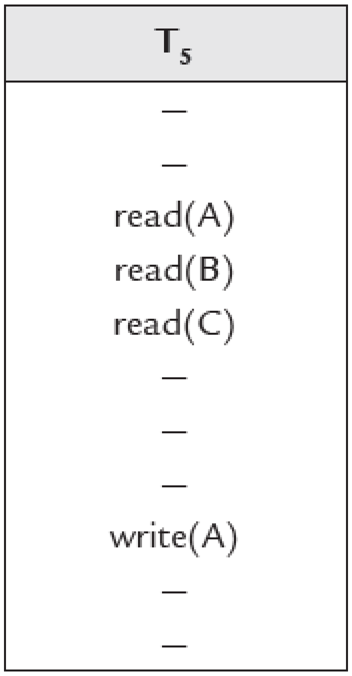
1. There is no cascading rollback in strict two-phase locking protocol.

Rigorous Two-phase Locking Protocol - In rigorous two-phase locking protocol, all locks are kept until the transaction commits. It means both shared and exclusive locks cannot be released till the end of transaction

Lock Conversion - To improve efficiency of two-phase locking protocol, modes of lock can be converted.

Upgrade: Conversion of shared mode to exclusive mode is known as upgrade.

Downgrade: Conversion of exclusive mode to shared mode is known as downgrade. Consider transaction T5 as shown in Figure 3.7.



T_5
—
—
read(A)
read(B)
read(C)
—
—
—
write(A)
—
—

Figure 3.7 Lock conversion (Upgrade)

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

In graph-based protocols, an additional information is needed for each transaction to know, how they will access data items of database. These protocols are used where two-phase protocols are not applicable but they required additional information that is not required in two-phase protocols. In graph-based protocols partial ordering is used (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Consider, a set of data items D = *d*1, *d*2, ..., *di*, *dj*, ..., *dz*}. If *di* → *dj* then, if any transaction T want to access both *di* and *dj* then T have to access *di* first then *dj*. Here, tree protocol is used in which D is viewed as a directed acyclic graph known as database graph (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Rules for tree protocol:

1. Any transaction T can lock any data item at first time.
2. After first lock if T wants to lock any other data item *v* then, first, T have to lock its parent.
3. T cannot release all exclusive locks before commit or abort.
4. Suppose T has obtained lock on data item *v* and then release it after sometime. Then T cannot obtain any lock on *v* again.

All the schedules under graph-based protocols maintain serializability. Cascadelessness and recoverability are maintained by rule (3) but this reduces concurrency and hence reduce efficiency (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Alternate approach: Here rule (3) is modified. Modified version is (3) T can release any lockat any time.

But in alternate approach there may be cascading rollback.

For each data item, if any transaction performed the last write to some data item and still uncommitted then record that transaction.

Commit dependency: A transaction T*i* has commit dependency upon T*j* if T*i* wants to read a uncommitted data item *v* and T*j* is the most recent transaction which performed write operation on *v.*

If any of these transactions aborts, T*i* must also be aborted. Consider the tree structured database graph as shown in Figure 3.8.

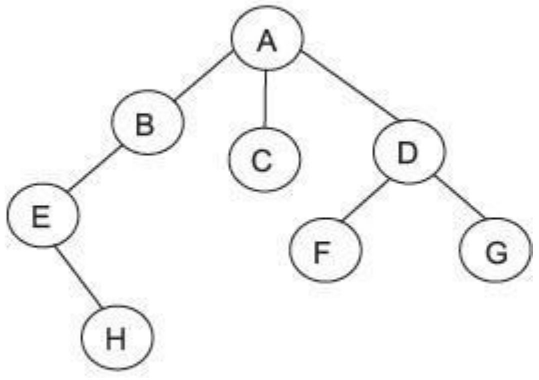


Figure 3.8 Tree structured database graph.

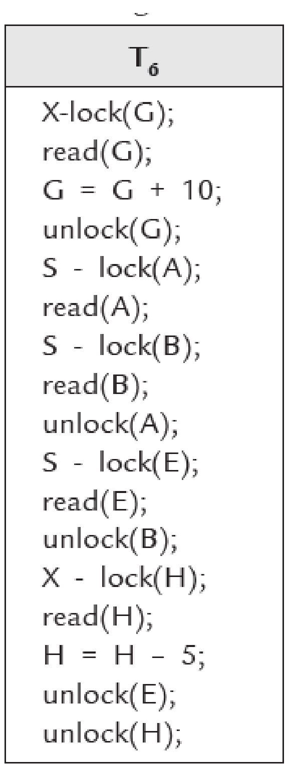
Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

Suppose we want to operate

G = G + 10 and

H = H – 5

then possible transaction is as shown in Figure 3.9.



T_6
X-lock(G);
read(G);
G = G + 10;
unlock(G);
S - lock(A);
read(A);
S - lock(B);
read(B);
unlock(A);
S - lock(E);
read(E);
unlock(B);
X - lock(H);
read(H);
H = H – 5;
unlock(E);
unlock(H);

Figure 3.9 Transaction using graph based protocols

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

Advantages: The advantages of graph-based protocols are:

1. There is no deadlock.
2. Unlike in two-phase, data items can be unlocked at any time, that reduces waiting time. 3. More concurrency.

Disadvantages: Those items have to be locked that have no use in transactions. This results in increased locking overhead. (locking of A, B and E to lock H in T_6 of Figure 3.9).

Validation Based Protocols

Validation Based Protocols are used where most of the transactions are read-only and conflicts are low. In these protocols, every transaction T_i have to go through the following phases depending upon its (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Read Phase: In read phase T_i reads all the data items and store them in temporary variables (local variables of T_i). After reading, all the write operations are made on the temporary variables (Ramakrishhan, Raghu and Gehrke,Johannes,2018).
Validation Phase: In validation phase, a validation test is performed to determine whether changes in actual database can be made (Ramakrishhan, Raghu and Gehrke,Johannes,2018).
Write Phase: If T_i cleared the validation test then actual database is changed according to temporary variables. Here actual changes are made in database. Validation test ensures violation free execution of transactions. To determine the time to perform validation test, use timestamps. Each transaction T_i is associated with three timestamps (Ramakrishhan, Raghu and Gehrke,Johannes,2018)
These are:

1. Start(T_i) : It gives time when Ti starts execution.
2. Validation(T_i) : It gives time whenTi finished its read phase and starts its validation phase.
3. Finish(T_i) : It gives time when Ti finished execution or write phase.

If any transaction T_i failed in validation test then it is aborted and rollback. To clear the validation test by transaction T_j, for all transactions such that $T_S(T_i) < T_j$, T_j must satisfy one of the following conditions:

1. If Finish(T_j) < Start(T_j) then there is no conflict because they are in serial order.
2. If Start(T_j) < Finish(T_i) < Validation(T_j). This condition ensures that actual writes to database for T_i and T_j does not overlap

Example. Consider the schedule as shown in Figure 3.10.

T_9	T_{10}
read(A)	
	read(B)
	B = B * 5;
read(B)	
	read(A)
	A = A + 10;
(Validate)	
display(A + B)	
	(Validate)
	Write(B)
	Write(A)

Figure 3.10 Schedule for transaction T_9 and T_{10} using validation-based protocols.

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

Suppose, initially A = 10 and B = 5 then T_9 display A+B = 15 because T_{10} first writenew values of A and B to local variables and validate after T_9 finished its execution.

Advantages: The advantages of validation-based protocols are:

1. It maintains serializability.
2. It is free from cascading rollback.
3. Less overhead than other protocols.

Disadvantages: The disadvantages of validation-based protocols are:

1. Starvation of long transactions due to conflicting short transactions.

Lesson 4. Deadlocks

A system is said to be in deadlock state if there exist a set of transactions {T_0, T_1,..., T_n} such that each transaction in set is waiting for releasing any resource by any other transaction in that set. In this case, all the transactions are in waiting state (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Necessary Conditions for Deadlock

A system is in deadlock state if it satisfies all of the following conditions.

1. **Hold and wait**: Whenever a transaction holding at least one resource and waiting for other resources that are currently being held by other processes.
2. **Mutual Exclusion**: When any transaction has obtained a non-sharable lock on any data item and any other transaction requests that item.
3. **No Preemption**: When a transaction holds any resource even after its completion.
4. **Circular Wait**: Let T be the set of waiting processes T = {T_0, T_1, ..., T_i} such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, T_i is waiting for a resource that is held by T_0.

There are Two methods for handling deadlocks. These are:

1. **Deadlock Prevention -** It is better to prevent the system from deadlock condition then to detect it and then recover it. Different approaches for preventing deadlocks are:

   a. **Advance Locking:** It is the simplest technique in which each transaction locks all the required data items at the beginning of its execution in atomic manner. It means all items are locked in one step or none is locked (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

      Disadvantages: The disadvantages of advance locking are:

      1. It is very hard to predict all data items required by transaction at starting.
      2. Underutilization is high.
      3. Reduces concurrency.

b. **Ordering Data Items**: In this approach, all the data items are assigned in order say 1, 2, 3, ... etc. Any transaction T_i can acquire locks in a particular order, such as in ascending order or descending order (Ramakrishhan, Raghu and Gehrke,Johannes,2018).
   Disadvantages: It reduces concurrency but less than advance locking.

c. **Ordering Data Items with Two-phase Locking**: In this approach, two-phase locking with ordering data items is used to increase concurrency (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

d. **Techniques based on Timestamps**: There are Two different techniques to remove deadlock based on time stamps.
   a. **Wait-die** : In wait-die technique if any transaction T_i needs any resource that is presently held by T_j then T_i have to wait only if it has timestamp smaller than T_j otherwise T_i is rolled back (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

```
If TS(Ti)<TS(Tj)
        Ti waits;
else
        Ti is rolled back;
```

   It is a nonpreemptive technique.
   b. **Wound-wait** : In wound-wait technique, if any transaction T_i needs any resource that is presently held by T_j then T_i have to wait only if it has timestamp larger then T_j otherwise T_j is rolled back.If $TS(T_i)>TS(T_j)T_i$waits;

elseIt is a preemptive technique. T_j is rolled back;

Disadvantage : There are unnecessary roll backs in both techniques that leads to starvation.

Deadlock Detection and Rec

In this approach, allow system to enter in deadlock state then detect it and recover it. To employ this approach, system must have (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

1. Mechanism to maintain information of current allocation of data items to transactions and the order in which they are allocated.
2. An algorithm which is invoked periodically by system to determine deadlocks in system.
3. An algorithm to recover from deadlock state.

Deadlock Detection: To detect deadlock, maintain wait-for graph.

Wait-for Graph : It consists of a pair G = (V, E), where V is the set of vertices and E is the set of edges. Vertices show transactions and edges show dependency of transactions. If transaction T_i request a data item which is currently being held by T_j then, an edge $T_i \rightarrow T_j$ is inserted in graph. When T_i has obtained the required item from T_j remove the edge. Deadlock occurs when there is cycle in wait-for graph (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Algorithm to Construct Wait-for Graph

1. For each transaction T_i active at the time of deadlock detection, create a node labelled T_i in the wait-for graph.
2. For each case, if there is a transaction T_i, waiting for a data-item that is currently allocated and held by transaction T_j, then there is a directed arc from the node for transaction T_i, to the node for transaction T_j.
3. For each case, if there is a directed arc from the node for transaction T_i, to the node for transaction T_j and transaction T_j released the lock, then remove the arc between them.

4. There exists no deadlock if the wait-for graph is acyclic

Consider wait-for graph as shown in Figure 3.11.

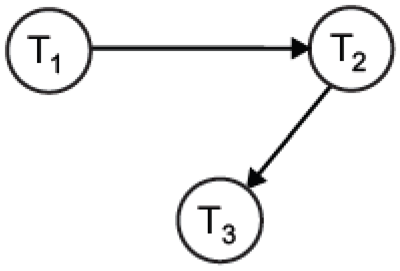


Figure 3.11 Wait-for graph without deadlock situation

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

There are three transactions T_1, T_2 and T_3. The transaction T_1 is waiting for any data item held by T_2 and T_2 is waiting for any data item held by T_3. But there is no deadlock because there is no cycle in wait-for graph (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Consider the graph as shown in Figure 3.12, where T_3 is also waiting for any data item held by T_1.

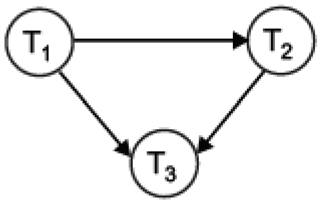


Figure 3.12 Wait-for graph with deadlock

Source :Ramakrishhan, Raghu and Gehrke,Johannes,2018

Thus, there exists a cycle in wait-for graph.

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$$

which results system in deadlock state and T_1, T_2 and T_3 are all blocked.

Recovery from Deadlock - When deadlock detection algorithm detects any deadlock in system, system must take necessary actions to recover from deadlock (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Steps of recovery algorithm are:

1. **Select a victim :** To recover from deadlock, break the cycle in wait-for graph. To break this cycle, rollback any of the deadlocked transaction. The transaction which is rolled back is known as **victim** (Ramakrishhan, Raghu and Gehrke,Johannes,2018).
   Various factors to determine victim are:
   - Cost of transaction.
   - How many more data items it required to complete its task?
   - How many more transactions affected by its rollback?
   - How much the transaction is completed and left?
2. **Rollback**: After selection of victim, it is rollbacked.
   There are Two types of rollback to break the cycle:
   a. **Total rollback**: This is simple technique. In total rollback, the victim is rolled back completely (Ramakrishhan, Raghu and Gehrke,Johannes,2018)
   b. **Partial rollback**: In this approach, rollback the transaction up to that point whichis necessary to break deadlock. Here transaction is partially rolled back. It is an effective technique but system have to maintain additional information aboutthe state of all running transactions (Ramakrishhan, Raghu and Gehrke,Johannes,2018).
3. **Prevent Starvation**: Sometimes, a particular transaction is rolledback several times and never completes which causes starvation of that transaction. To prevent starvation, set the limit of rollbacks or the maximum number, the same transaction chooses as victim. Also, the number of rollbacks can be added in cost factor to reduce starvation.

Timeout-based Schemes

Timeout-based scheme exists in between deadlock prevention and deadlock detection and recovery. In this scheme, the transaction waits for at most a fixed time for any resource. If transaction is not able to get resource in that fixed time then it is said to be timeout and transaction is rollbacked and restarted after sometime (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

This approach is preferred where transactions are short and long waits cause deadlocks. It is typical to choose appropriate time for timeout. If it is too long then it results unnecessary delays and if it is too short then it causes unnecessary rollbacks (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Example: Justify the following statement: Concurrent execution of transactions is more important when data must be fetched from (slow) disk or when transactions are long, and is less important when data is in memory and transactions are very short.

Solution: If a transaction is very long or when it fetches data from a slow disk, it takes a long time to complete. In absence of concurrency, other transactions will have to wait for longer period of time. Average response time will increase. Also, when the transaction is reading data from disk, CPU is idle. So, resources are not properly utilized. Hence concurrent execution becomes important in this case. However, when the transactions are short or the data is available in memory, these problems do not occur

Example: Consider the following actions taken by transaction T1 on databases X and Y:

R(X), W(X),

R(Y), W(Y)

1. Give an example of another transaction T2 that, if run concurrently to transaction T1 without some form of concurrency control, could interfere with T1.

2. Explain how the use of Strict 2PL would prevent interference between the two transactions.

3. Strict 2PL is used in many database systems. Give two reasons for its popularity.

Solution:

1. If the transaction T2 performed W(Y) before T1 performed R(Y), and then T2 aborted, the value read by T1 would be invalid and the abort would be cascaded to T1 (i.e. T1 would also have to abort.).

2. Strict 2PL would require T2 to obtain an exclusive lock on Y before writing to it. This lock would have to be held until T2 committed or aborted; this would block T1 from reading Y until T2 was finished, but there would be no interference.

3. Strict 2PL is popular for many reasons. One reason is that it ensures only ‘safe’ interleaving of transactions so that transactions are recoverable, avoid cascading aborts, etc. Another reason is that strict 2PL is very simple and easy to implement. The lock manager only needs to provide a lookup for exclusive locks and an atomic locking mechanism (such as with a semaphore).

For additional information and materials:

- https://www.youtube.com/watch?v=s1YEg1L0sH0
- https://www.youtube.com/watch?v=Thm0xW9oTow
- https://www.youtube.com/watch?v=B-QWfuIIBfk
- https://www.youtube.com/watch?v=QzvVQ8vRDuM
- https://www.youtube.com/watch?v=WO_VNMUuGoU

Assessment Task

Fill in the blanks:

1. A(n)_____________transaction is one for which all committed changes are permanent.

2. If a(n)______________is issued before the termination of a transaction, the DBMS will restore the database only for that particular transaction, rather than for all transactions.

3. A(n)________occurs when one transaction reads a changed record that has not been committed to the database.

4. The objective of_____________control is to ensure the serializability of transactions.

5. __________control is the simultaneous execution of transactions over a shared database.

6. Locks placed by a command issued to the DBMS from the application program are called ________locks.

7. The______interleaves the execution of database operations to ensure serializability.

8. To determine the appropriate order of the operations, the scheduler uses concurrency control algorithms, such as____________or time stamping methods.

9. ______________can take place at any of the following levels: database, table, page, row, or field.

10. ____________-level locks are less restrictive than database-level locks, but can create traffic jams when many transactions are waiting to access the same table.

11. A_________must unlock the object after its termination.

12. The size of a lock is referred to as the lock_________.

13. Only one transaction at a time can own an exclusive lock on the same object as per the

__________rule states.

14. ______________ensures that time stamp values always increase.

15. The DBMS keeps several copies of__________to ensure that a disk physical failure will

not harm the DBMS's ability to recover data.

Multiple choice

1. Which manager is responsible for assigning and policing the locks used by the transactions?

(a) Transaction (b) Database (c) Lock (d) Schedule

2. Which of the following Lock indicates the level of lock use?

(a) Granularity (b) Shrinking (c) Growing (d) Serializability

3. A disk page, or page, is the equivalent of a

(a) database table (b) disk sector

(c) database schema (d) diskblock

4. Which condition occurs when two or more transactions wait for each other to unlock data?

(a) Deadlock (b) Shared lock (c) Exclusive lock (d) Binary lock

5. Which of the following rules apply to the two-phase locking protocol?

(a) Two transactions cannot have conflicting locks.

(b) No unlock operation can precede a lock operation in a different transaction.

(c) No data is affected until all locks are released.

(d) No data is affected until the transaction is in its locked position.

6. Which phase in a two-phase lock is when a transaction releases all locks and cannot obtain any new lock?

(a) Growing (b) Shrinking (c) Locking (d) Unlocking

7. Which approach to scheduling concurrent transactions assigns a global unique stamp to each

transaction?

(a) Scheduled (b) Table-locking (c) Unique (d) Time stamping

8. In which phase, changes are permanently applied to the database?

(a) Read (b) Validation (c) Write (d) Shared

9. Which of the following is not true about two-phased locking?

(a) Can make transactions serializable

(b) Uses only shared locks

(c) Has a shrinking phase

(d) Cannot obtain a new lock once a lock has been released

0. Which type of lock prevents all types of access to the locked resource?

(a) Exclusive lock (b) Shared lock

(c) Two-phased lock (d) Explicit lock

Summary

Concurrency control is the activity of coordinating the actions of transactions that operate, simultaneously or in parallel to access the shared data. How the DBMS handles concurrent executions is an important aspect of transaction management. Other related issues are how the DBMS handles partial transactions, or transactions that are interrupted before successful completion. The DBMS makes it sure that the modifications done by such partial transactions are not seen by other transactions. Thus, transaction processing and concurrency control form important activities of any Database Management System (DBMS)

References

First Normal Form (1NF) of Database Normalization. (n.d.). Retrieved November 28, 2020, from https://www.studytonight.com/dbms/first-normal-form.php

University Science Press. https://doi.org/10.1201/9781315372679-7

Lemahieu, W., Broucke, S. vanden, & Baesens, B. (2018). Principles of Database Management. Cambridge University Press. https://doi.org/10.16309/ j.cnki.issn.1007- 1776.2003.03.004

MySQL INNER JOIN By Practical Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-inner-join.aspx

MySQL Join Made Easy For Beginners. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-join/

MySQL RIGHT JOIN Explained By Practical Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-right-join/

Second Normal Form (2NF) of Database Normalization. (n.d.). Retrieved November 28, 2020, fromhttps://www.studytonight.com/dbms/second-normal-form.php

Sheldon, R., & Moes, G. (2005). *Beginning MySQL* (1st ed.). Wrox.

Understanding MySQL LEFT JOIN Clause By Examples. (n.d.). Retrieved November 26, 2020, from https://www.mysqltutorial.org/mysql-left-join.aspx

(Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Source :SQL for beginners: A guide to study SQL programming and database management systems

Database management systemsGupta, G. K.,2018

MODULE 4
MYSQLVIEWS

Introduction

A normalized schema is the best starting point for any application. OLTP (on-line transaction processing) and OLAP (on-line analytics processing) have often used separate schemas. Batch imports and the Extract-Transform-Load (ETL) process are commonly used to migrate data from a transactional schema to an analytical one. However, this takes time and effort, and introduces a delay in data getting to a reporting server (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The need for more real-time analysis is growing stronger. In addition, replication in MySQL is widespread and easy. Views are tools that assist in denormalizing data, such as for analysis, without changing the underlying system for transaction processing (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Learning Outcomes

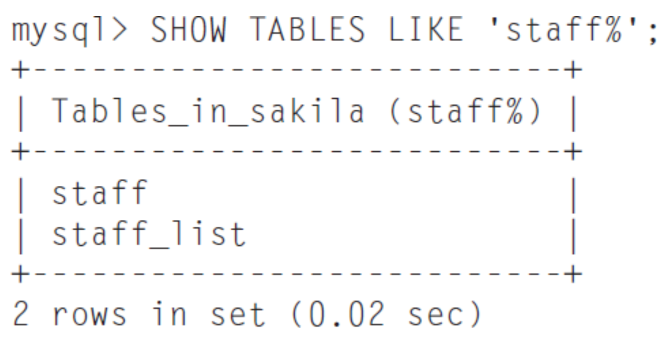
At the end of this module, students should be able to:

1. Understand views in MySQL
2. Define views in MySQL
3. Use views for security purposes

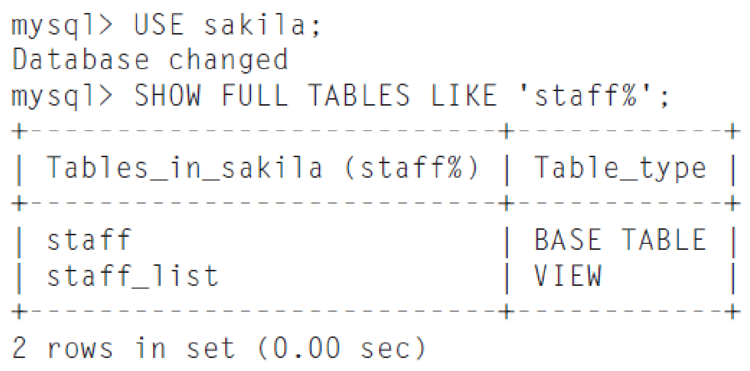
Lesson 1. Defining Views

A view is a way to define a dynamic virtual table based on an SQL statement. It can be queried like a regular table can, and views appear in the TABLES table of the INFORMATION_SCHEMA database. A view is an object associated with a database, similar to a regular table. In fact, the SHOW TABLES and SHOW TABLE STATUS commands return tables and views (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

You can get the SAKILA sample database from the following link:
https://dev.mysql.com/doc/sakila/en/sakila-installation.html



```
mysql> SHOW TABLES LIKE 'staff%';
+--------------------------+
| Tables_in_sakila (staff%) |
+--------------------------+
| staff                    |
| staff_list               |
+--------------------------+
2 rows in set (0.02 sec)
```



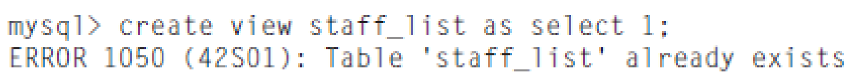
```
mysql> USE sakila;
Database changed
mysql> SHOW FULL TABLES LIKE 'staff%';
+---------------------------+------------+
| Tables_in_sakila (staff%) | Table_type |
+---------------------------+------------+
| staff                     | BASE TABLE |
| staff_list                | VIEW       |
+---------------------------+------------+
2 rows in set (0.00 sec)
```

SHOW TABLE STATUS can help you determine if an object is a table or a view. In the sakila sample database, staff is a table and staff_list is a view:

The simplest CREATE VIEW statement has the following syntax:

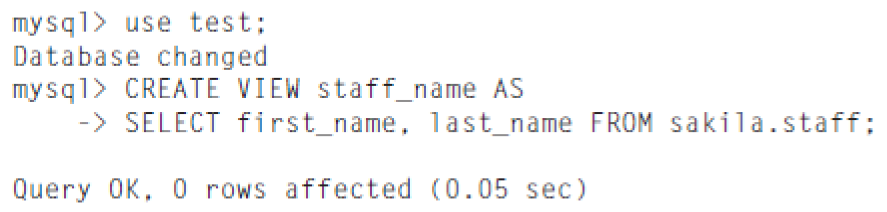
CREATE [OR REPLACE] VIEW viewname AS {SELECT statement}

CREATE VIEW will attempt to create a VIEW. If the VIEW exists, an error will be thrown:

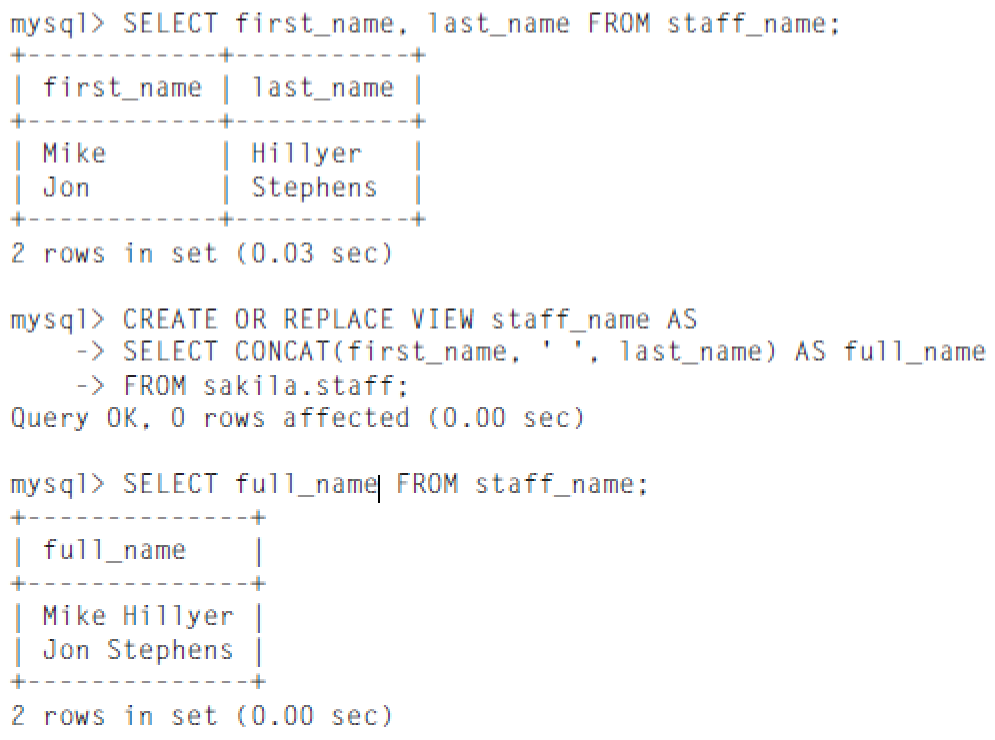


```
mysql> create view staff_list as select 1;
ERROR 1050 (42S01): Table 'staff_list' already exists
```

CREATE OR REPLACE VIEW will change the VIEW if it exists:

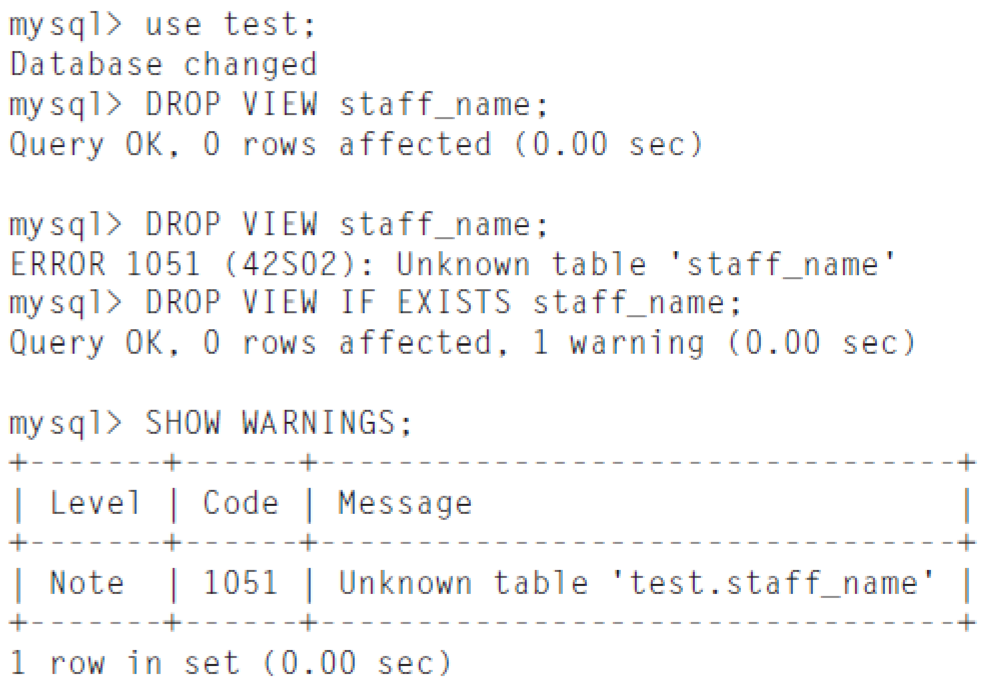


```
mysql> use test;
Database changed
mysql> CREATE VIEW staff_name AS
    -> SELECT first_name, last_name FROM sakila.staff;

Query OK, 0 rows affected (0.05 sec)
```



```
mysql> SELECT first_name, last_name FROM staff_name;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Mike       | Hillyer   |
| Jon        | Stephens  |
+------------+-----------+
2 rows in set (0.03 sec)

mysql> CREATE OR REPLACE VIEW staff_name AS
    -> SELECT CONCAT(first_name, ' ', last_name) AS full_name
    -> FROM sakila.staff;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT full_name FROM staff_name;
+--------------+
| full_name    |
+--------------+
| Mike Hillyer |
| Jon Stephens |
+--------------+
2 rows in set (0.00 sec)
```

The SELECT statement in the view definition can be almost any SELECT query, including unions and multi-table joins. The DROP VIEW statement works like other DROP statements. The DROP VIEW IF EXISTS statement will DROP the view if it exists, and if it does not exist will issue a warning (Ramakrishhan, Raghu and Gehrke,Johannes,2018).



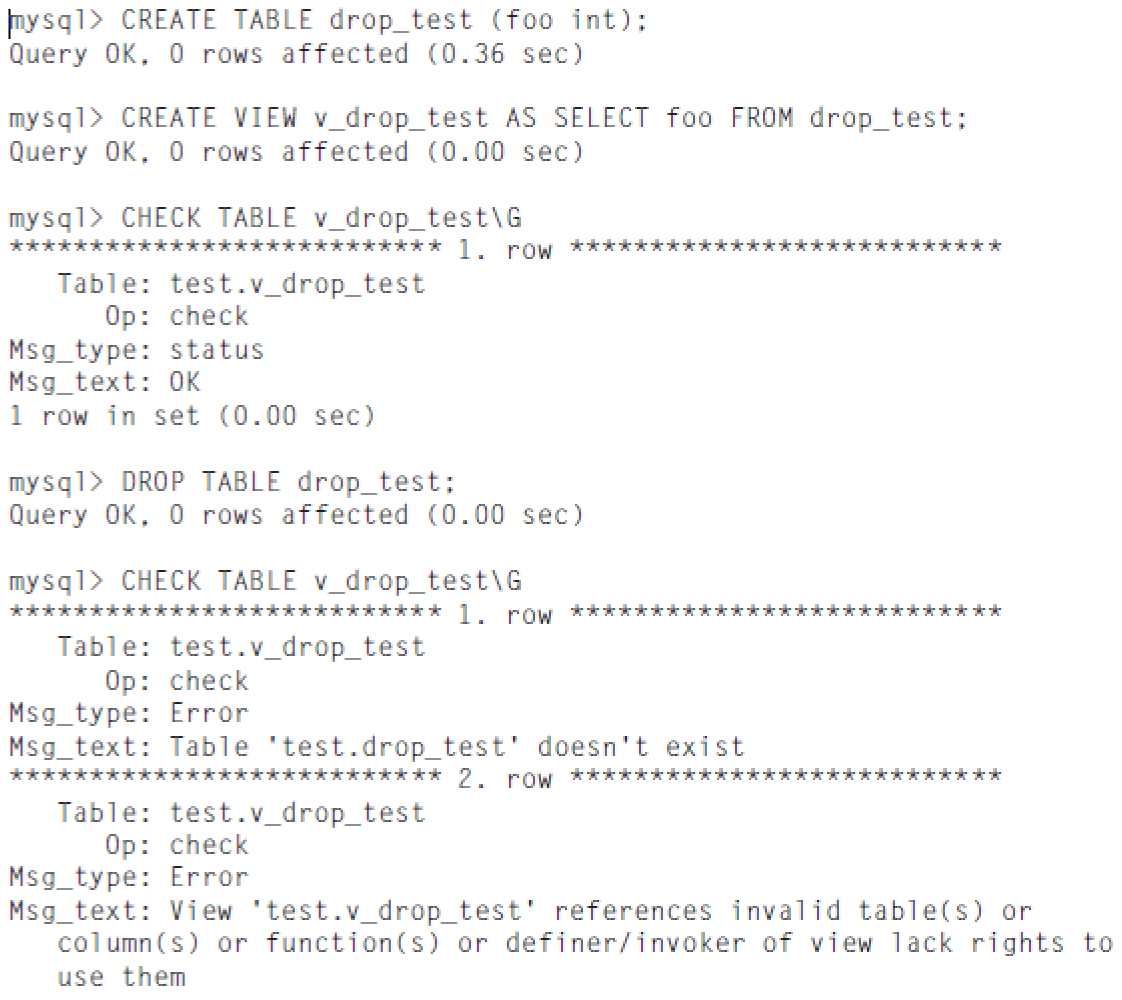
```
mysql> use test;
Database changed
mysql> DROP VIEW staff_name;
Query OK, 0 rows affected (0.00 sec)

mysql> DROP VIEW staff_name;
ERROR 1051 (42S02): Unknown table 'staff_name'
mysql> DROP VIEW IF EXISTS staff_name;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+---------------------------------+
| Level | Code | Message                         |
+-------+------+---------------------------------+
| Note  | 1051 | Unknown table 'test.staff_name' |
+-------+------+---------------------------------+
1 row in set (0.00 sec)
```

For a view, SHOW CREATE TABLE is an alias for SHOW CREATE VIEW.

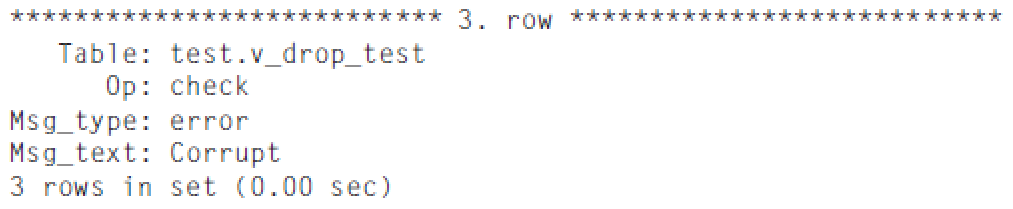
Lesson 2. View definition limitations and unexpected behavior

Not all SELECT statements are valid within a view definition. The SELECT statement in a view definition is not allowed to:

- Have a derived table (subquery in the FROM clause).
- Access user variables, system variables, local variables, stored routine variables, or prepared statement parameters.
- Refer to a TEMPORARY table.
- Refer to a table that does not exist. However, this is only checked at view creation time and any time a view is used. It is possible to drop an underlying base table or view that a view refers to without the DROP statement generating an error. Any attempt to use the view after an underlying base table or view is dropped will result in an error. The CHECK TABLE statement applied to a view will reveal if a referenced table or view no longer exists:



```
mysql> CREATE TABLE drop_test (foo int);
Query OK, 0 rows affected (0.36 sec)

mysql> CREATE VIEW v_drop_test AS SELECT foo FROM drop_test;
Query OK, 0 rows affected (0.00 sec)

mysql> CHECK TABLE v_drop_test\G
*************************** 1. row ***************************
   Table: test.v_drop_test
      Op: check
Msg_type: status
Msg_text: OK
1 row in set (0.00 sec)

mysql> DROP TABLE drop_test;
Query OK, 0 rows affected (0.00 sec)

mysql> CHECK TABLE v_drop_test\G
*************************** 1. row ***************************
   Table: test.v_drop_test
      Op: check
Msg_type: Error
Msg_text: Table 'test.drop_test' doesn't exist
*************************** 2. row ***************************
   Table: test.v_drop_test
      Op: check
Msg_type: Error
Msg_text: View 'test.v_drop_test' references invalid table(s) or
   column(s) or function(s) or definer/invoker of view lack rights to
   use them
```



```
*************************** 3. row ***************************
   Table: test.v_drop_test
      Op: check
Msg_type: error
Msg_text: Corrupt
3 rows in set (0.00 sec)
```

Several clauses and options within a SELECT statement may produce unexpected behavior when used as part of a view definition (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

- An ORDER BY in a view definition is not used if a query on the view has its own ORDER BY. That is, the query's ORDER BY statement overrides the view definition's ORDER BY statement.

- Unlike with an ORDER BY clause, there are no guidelines on how a query with a certain option is processed when the query refers to a view defined with that same option. The options this applies to are:
    - DISTINCT
    - INTO
    - FOR UPDATE
    - ALL
    - PROCEDURE
    - SQL_SMALL_RESULT
    - LOCK IN SHARE MODE

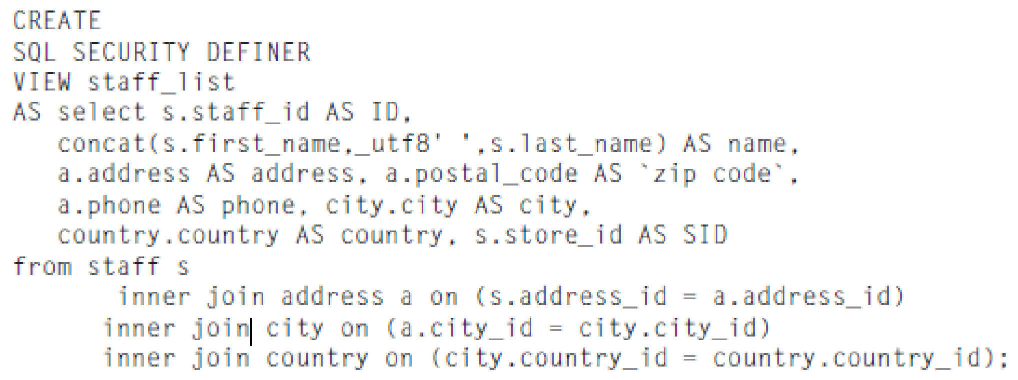
Security and privacy

Views in other database systems can act as security measures, by putting limitations and constraints in the view definition. Views in other database systems use limitations and constraints as their only security measures. MySQL has added an SQL SECURITY extension to the view definition in standard SQL. With the SQL SECURITY parameter, MySQL gives views the ability to be a window into data that a user may not have permission to see. Thus, MySQL allows views to have extended means of security (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

To define a MySQL view that acts like regular SQL, where the permissions are based on the user querying the view, use SQL SECURITY INVOKER. If the user does not have permission to run the SELECT query in the view definition, the user is not allowed to query the view. It is very useful to give a user permission to see parts of a table. This can be done with column-level privileges but there is a better way using views. With SQL SECURITY DEFINER specified, the access settings of the user who created the view are applied. SQL SECURITY DEFINER is the default value, and is useful when you want to give someone a partial window into the data without giving them full access to a table (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

As an example, Ziesel has a new employee, Benjamin, to whom she delegates sending out New Year cards to the staff at their homes. Because Ziesel is very concerned with privacy, she would like to allow Benjamin access to the database to see staff members phone numbers and mailing addresses, but not their e-mail addresses (or, for that matter, password hashes) (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

So Ziesel creates a view she calls staff_list. She starts with the staff table, taking the staff_id and store_id fields, and creating a name field that is a concatenation of the first and last names. The staff table contains a numeric reference to the address table, which in turn has numeric references to city and country tables, so she has to join to the address, city, and country to get a usable mailing address (Ramakrishhan, Raghu and Gehrke,Johannes,2018):



```
CREATE
SQL SECURITY DEFINER
VIEW staff_list
AS select s.staff_id AS ID,
   concat(s.first_name,_utf8' ',s.last_name) AS name,
   a.address AS address, a.postal_code AS `zip code`,
   a.phone AS phone, city.city AS city,
   country.country AS country, s.store_id AS SID
from staff s
       inner join address a on (s.address_id = a.address_id)
      inner join city on (a.city_id = city.city_id)
      inner join country on (city.country_id = country.country_id);
```

Even though Benjamin’s MySQL user does not have permission to the staff table, he can get the information he needs because he has the permission to SELECT from the staff_list view, and the view was created with SQL SECURITY DEFINER (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

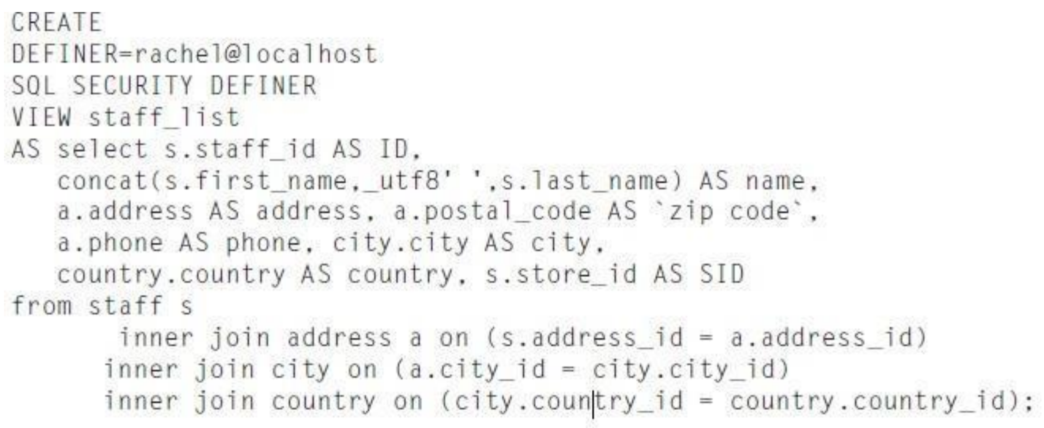
Depending on your organization’s policies, you may opt to put views in a separate database on the same MySQL server. This allows you to control view access using database-level granularities instead of table-level granularities. Views can also let a user see a subset of rows in one or more base tables they do not have access to, which cannot be done with the privilege system in MySQL. This is accomplished by creating a view using SQL SECURITY DEFINER with a definition that filters some rows (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Lesson 3. Specify a view's definer

A user with a lot of permissions uses the SQL SECURITY DEFINER property of views in order to easily give another user a look at some of the data. In our example, Ziesel wanted Benjamin to be able to see address information of the staff, but not e-mail or password information. Because SQL SECURITY DEFINER was specified, when Benjamin queries the view, he queries it as if he has Ziesel's permissions. What if Ziesel wants Benjamin to run the view as if he had someone else's permissions, though?

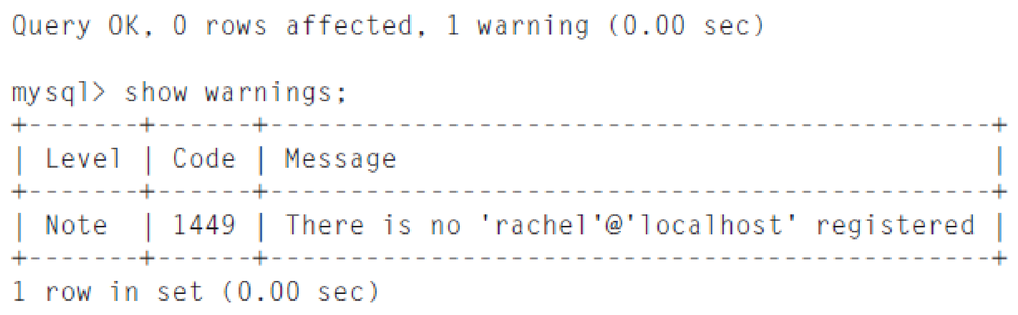
MySQL has another extension to standard SQL — the ability to change the view's definer; by default, DEFINER = CURRENT_USER. However, users with the SUPER privilege may specify a definer other than themselves. If a view specifies SQL SECURITY DEFINER and changes the value of DEFINER, then when the view is run it is as if the user invoking the view has the same permissions as the DEFINER (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Going back to our example, Ziesel can specify that anyone who queries the staff_list view should query it as if they were logged in as Rachel, Ziesel's administrative assistant. Rachel has the user rachel@localhost (Ramakrishhan, Raghu and Gehrke,Johannes,2018):



```
CREATE
DEFINER=rachel@localhost
SQL SECURITY DEFINER
VIEW staff_list
AS select s.staff_id AS ID,
   concat(s.first_name,_utf8' ',s.last_name) AS name,
   a.address AS address, a.postal_code AS `zip code`,
   a.phone AS phone, city.city AS city,
   country.country AS country, s.store_id AS SID
from staff s
       inner join address a on (s.address_id = a.address_id)
      inner join city on (a.city_id = city.city_id)
      inner join country on (city.country_id = country.country_id);
```

Now when Benjamin runs **SELECT * FROM staff_list**, it is run as if he were logged in as rachel@localhost. If the user rachel@localhost did not exist, the view would have been created and a warning would have been generated (Gupta & Mittal, 2017):



```
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> show warnings;
+-------+------+------------------------------------------+
| Level | Code | Message                                  |
+-------+------+------------------------------------------+
| Note  | 1449 | There is no 'rachel'@'localhost' registered |
+-------+------+------------------------------------------+
1 row in set (0.00 sec)
```

Only a user with the SUPER privileges can set the DEFINER to be anything other than CURRENT_USER. Specifying a different definer has no effect on views that specify SQL SECURITY INVOKER (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Lesson 4. Abstraction and simplification

In the previous example, Ziesel created a view for Benjamin to access relevant staff data. Another effect that a view can have is simplifying end-user queries. Instead of having to write a query joining four tables, Benjamin can write a very simple **SELECT * FROM staff_list** query. Views are a way to simplify queries for programmers more familiar and comfortable with procedural languages than they are with SQL (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

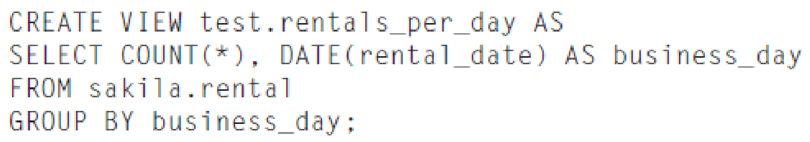
Many database administrators take advantage of the abstraction that views can provide and make schemas with views of views — meaning that a view is defined with a query that uses another view. This is perfectly valid, and can ease the tension that arises between developers and schema designers when developers want to write simple queries, and schema designers want to have efficient schemas (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The downside to making queries more simple is that someone using the query may assume that a simple **SELECT * FROM staff_list** query does a full table scan of one table. In reality, four tables are being joined in the query and the performance cost is a lot more than the simple query implies (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Performance

Views in MySQL are always dynamic. Whenever Benjamin runs **SELECT * FROM staff_list** the query performs as if he is running that four table join query. The results are not cached or materialized into a permanent structure. The exception to this is if the query_cache is turned on the results may be cached, following standard query_cache rules (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The dynamic nature of views makes them perform poorly for many uses, including aggregations. When reporting rentals per day, Ziesel decided a view would make reporting easier (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

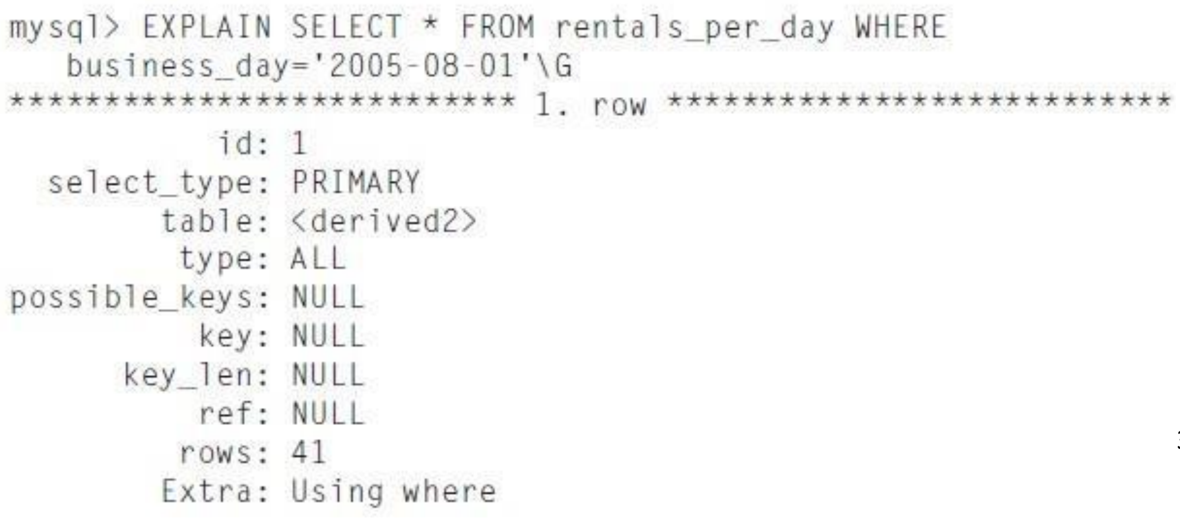


```
CREATE VIEW test.rentals_per_day AS
SELECT COUNT(*), DATE(rental_date) AS business_day
FROM sakila.rental
GROUP BY business_day;
```

Ziesel realized that there were redundant calculations going on. Once a day is over, the number of movies rented that day will not change. However, the query cache was invalidated whenever the rental table was updated — every time a new rental was made or a rental was returned. When she ran:

SELECT * FROM rentals_per_day WHERE business_day='2005-08-01'

the query took a long time to complete. So Ziesel ran an **EXPLAIN** query:



```
mysql> EXPLAIN SELECT * FROM rentals_per_day WHERE
   business_day='2005-08-01'\G
************************* 1. row *************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 41
        Extra: Using where
```

```
************************* 2. row *************************
           id: 2
  select_type: DERIVED
        table: rental
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 15596
        Extra: Using temporary; Using filesort
2 rows in set (0.05 sec)
```

She realized that MySQL was first calculating **DATE(rental_date)** for every single row in the base table rental, then grouping similar dates together, and only afterwards checking for matching rows. Her simple SELECT query was actually parsed into (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

```
SELECT * FROM
  (SELECT COUNT(*), DATE(rental_date) AS business_day
  FROM sakila.rental
  GROUP BY business_day) AS view_defn
WHERE business_day='2005-08-01'
```

Ziesel quickly understood that simplifying her query with a view did not have acceptable performance. A more optimized query to get the data she wanted would be (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

```
SELECT COUNT(*)
FROM rental
WHERE rental_date between '2005-08-01 00:00:00'
AND '2005-08-01 23:59:59';
```

But there is no way to define a view that groups by dates to obtain that optimization. It can only be done without a view (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

View algorithm

MySQL has an extension to the standard SQL to help with view performance. The ALGORITHM clause specifies which algorithm MySQL will use to process the view. The MERGE algorithm processes a query on a view by merging the query with the view definition.

This is what Ziesel saw in the previous example — the query

```
SELECT *
FROM rentals_per_day
WHERE business_day='2005-08-01'
```

on the rentals_per_day view defined by the SQL query

```
CREATE VIEW test.rentals_per_day
AS
SELECT COUNT(*), DATE(rental_date) AS business_day


FROM sakila.rental
GROUP BY business_day
```

was processed as:

```
SELECT *
 FROM
   (SELECT COUNT(*), DATE(rental_date) AS business_day
   FROM sakila.rental
   GROUP BY business_day) AS view_defn
 WHERE business_day='2005-08-01'
```

MySQL offers another way to process a view. If the **ALGORITHM** is defined as **TEMPTABLE**, the **VIEW** is processed and the output is stored in a temporary table. Then the temporary table is used in the query (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

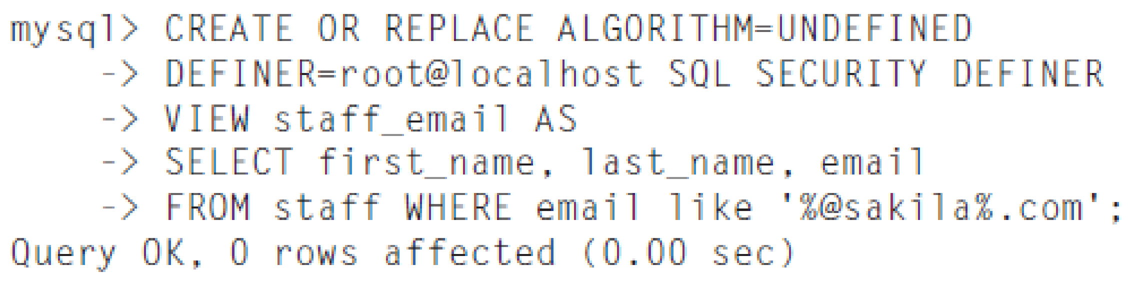
The default setting is **ALGORITHM=UNDEFINED**, which means MySQL will choose which is more efficient. The **MERGE** algorithm is preferred over the **TEMPTABLE** algorithm in most cases because it is faster overall. However, locks on the base tables are released when the temporary table is generated, so **TEMPTABLE** can lead to better server performance overall because there is less time spent waiting for locks to be released (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

If **ALGORITHM=MERGE** is set for a view that can only use **TEMPTABLE**, MySQL issues a warning and saves the view as **ALGORITHM=UNDEFINED** (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Simulating check constraints

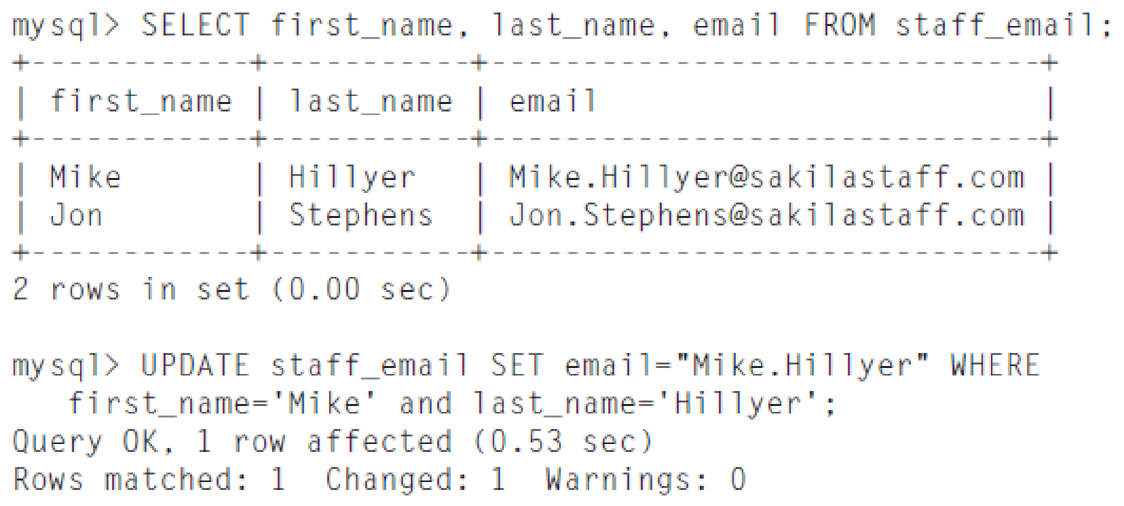
A check constraint is a predicate that limits the data that can be stored inside a table. MySQL does not offer check constraints, though data can be limited through data types, primary keys, unique keys, and foreign keys (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Check constraints can be simulated with the **WITH CHECK OPTION** in a view definition. **WITH CHECK OPTION** means that data updates made to the view will not occur in the base table unless the **WHERE** clause of the view definition is satisfied (Ramakrishhan, Raghu and Gehrke,Johannes,2018).



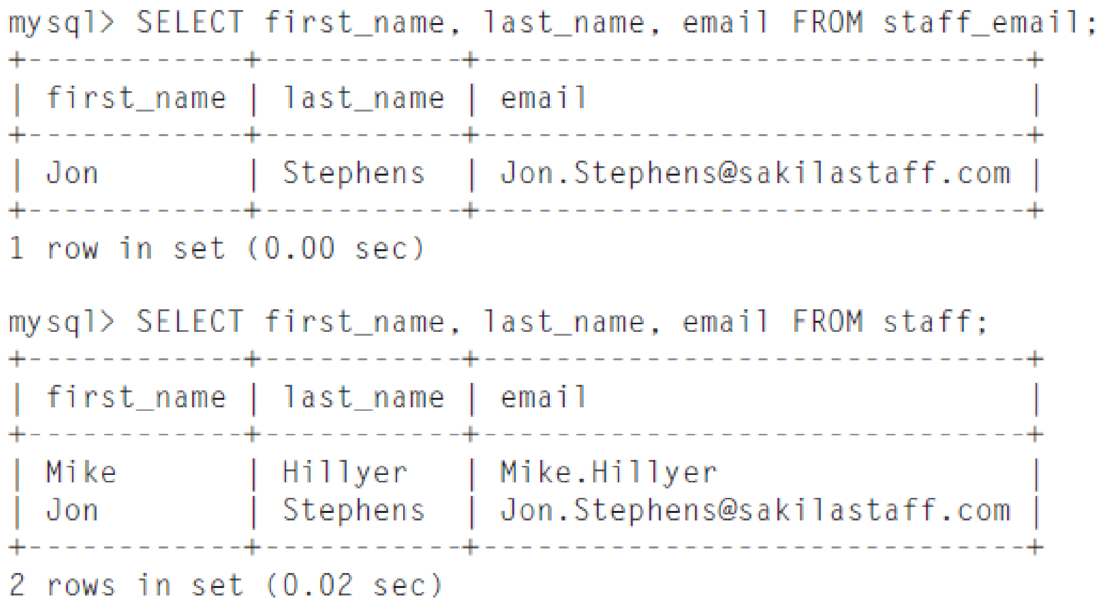
```
mysql> CREATE OR REPLACE ALGORITHM=UNDEFINED
    -> DEFINER=root@localhost SQL SECURITY DEFINER
    -> VIEW staff_email AS
    -> SELECT first_name, last_name, email
    -> FROM staff WHERE email like '%@sakila%.com';
Query OK, 0 rows affected (0.00 sec)
```

As an example, Ziesel wants only corporate e-mail addresses to be stored for staff members. First, she examines the case where no **WITH CHECK OPTION** clause is defined in the view, and sees that an **UPDATE** violating the **WHERE** clause of the view definition is allowed (Ramakrishhan, Raghu and Gehrke,Johannes,2018):



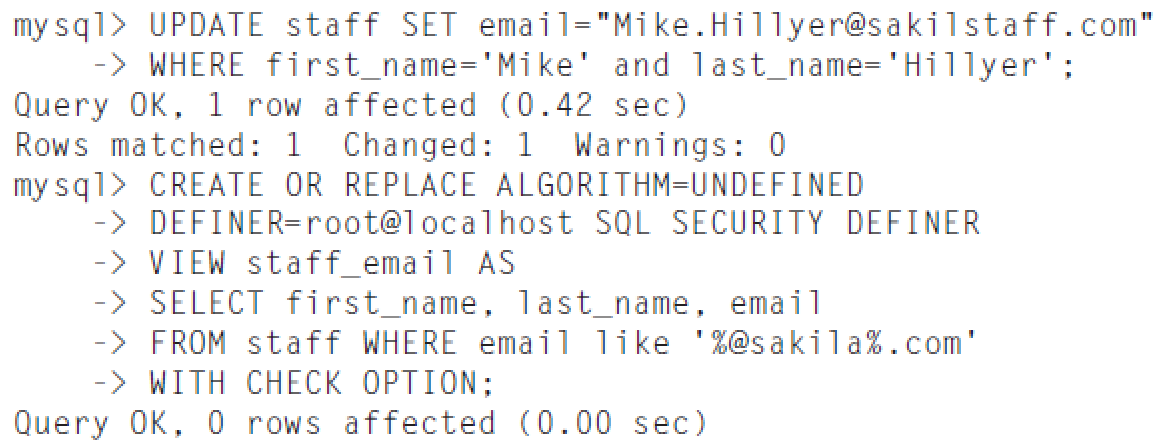
```
mysql> SELECT first_name, last_name, email FROM staff_email;
+------------+-----------+----------------------------+
| first_name | last_name | email                      |
+------------+-----------+----------------------------+
| Mike       | Hillyer   | Mike.Hillyer@sakilastaff.com |
| Jon        | Stephens  | Jon.Stephens@sakilastaff.com |
+------------+-----------+----------------------------+
2 rows in set (0.00 sec)

mysql> UPDATE staff_email SET email="Mike.Hillyer" WHERE
   first_name='Mike' and last_name='Hillyer';
Query OK, 1 row affected (0.53 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

This **UPDATE** statement violated the view definition of **WHERE email LIKE '%@sakila%.com'**, but one row was changed in the base table anyway. The view no longer shows Mike Hillyer, because he is filtered out. The base table should have an updated e-mail for Mike Hillyer (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

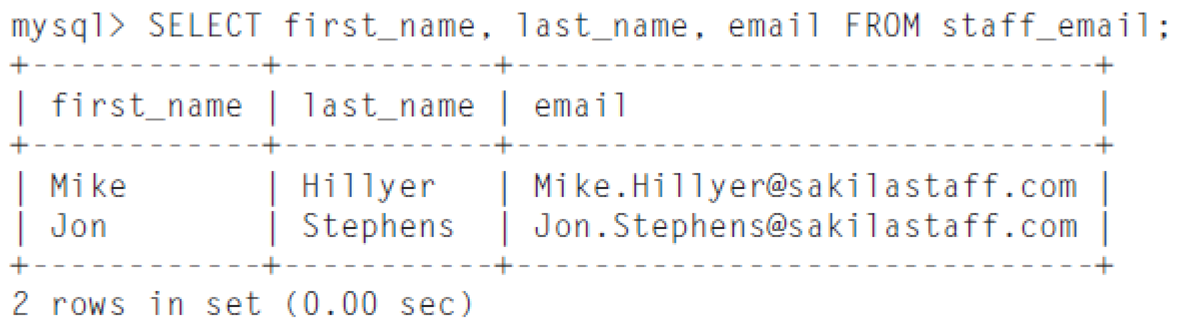


```
mysql> SELECT first_name, last_name, email FROM staff_email;
+------------+-----------+------------------------------+
| first_name | last_name | email                        |
+------------+-----------+------------------------------+
| Jon        | Stephens  | Jon.Stephens@sakilastaff.com |
+------------+-----------+------------------------------+
1 row in set (0.00 sec)

mysql> SELECT first_name, last_name, email FROM staff;
+------------+-----------+------------------------------+
| first_name | last_name | email                        |
+------------+-----------+------------------------------+
| Mike       | Hillyer   | Mike.Hillyer                 |
| Jon        | Stephens  | Jon.Stephens@sakilastaff.com |
+------------+-----------+------------------------------+
2 rows in set (0.02 sec)
```

Now, Ziesel resets the data and changes the view to have a **WITH CHECK OPTION** clause:

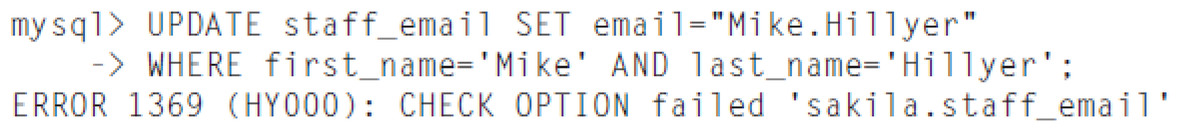


```
mysql> UPDATE staff SET email="Mike.Hillyer@sakilstaff.com"
    -> WHERE first_name='Mike' and last_name='Hillyer';
Query OK, 1 row affected (0.42 sec)
Rows matched: 1  Changed: 1  Warnings: 0
mysql> CREATE OR REPLACE ALGORITHM=UNDEFINED
    -> DEFINER=root@localhost SQL SECURITY DEFINER
    -> VIEW staff_email AS
    -> SELECT first_name, last_name, email
    -> FROM staff WHERE email like '%@sakila%.com'
    -> WITH CHECK OPTION;
Query OK, 0 rows affected (0.00 sec)
```



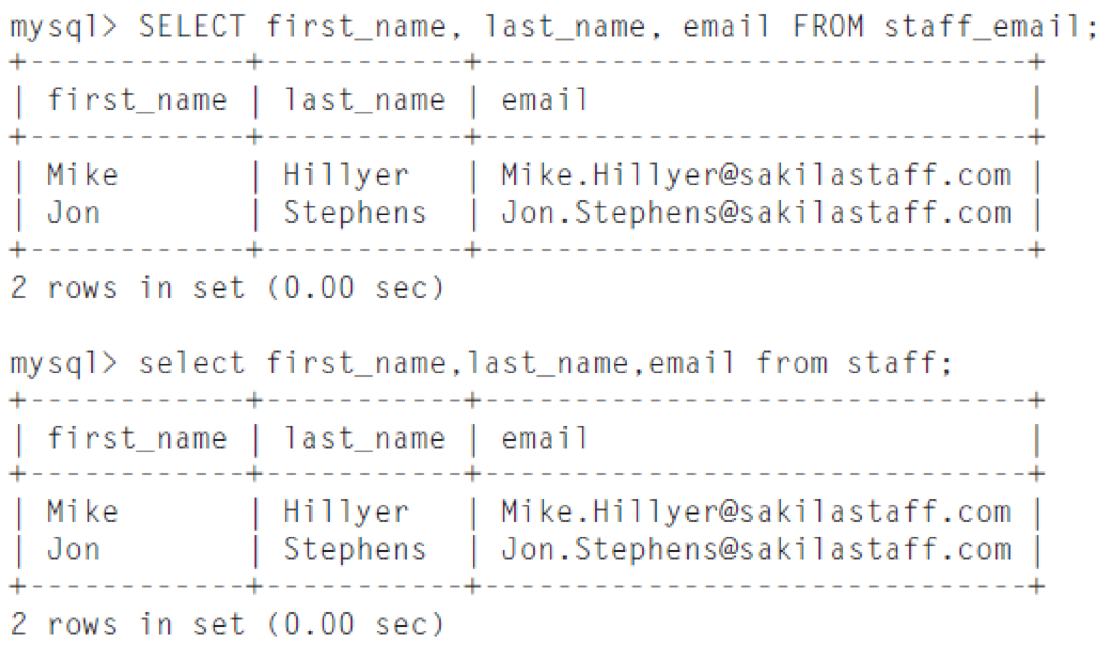
```
mysql> SELECT first_name, last_name, email FROM staff_email;
+------------+-----------+------------------------------+
| first_name | last_name | email                        |
+------------+-----------+------------------------------+
| Mike       | Hillyer   | Mike.Hillyer@sakilastaff.com |
| Jon        | Stephens  | Jon.Stephens@sakilastaff.com |
+------------+-----------+------------------------------+
2 rows in set (0.00 sec)
```

The **WITH CHECK OPTION** clause means that updates that violate the view's definition will not be allowed:



```
mysql> UPDATE staff_email SET email="Mike.Hillyer"
    -> WHERE first_name='Mike' AND last_name='Hillyer';
ERROR 1369 (HY000): CHECK OPTION failed 'sakila.staff_email'
```

And indeed, no modification was made:



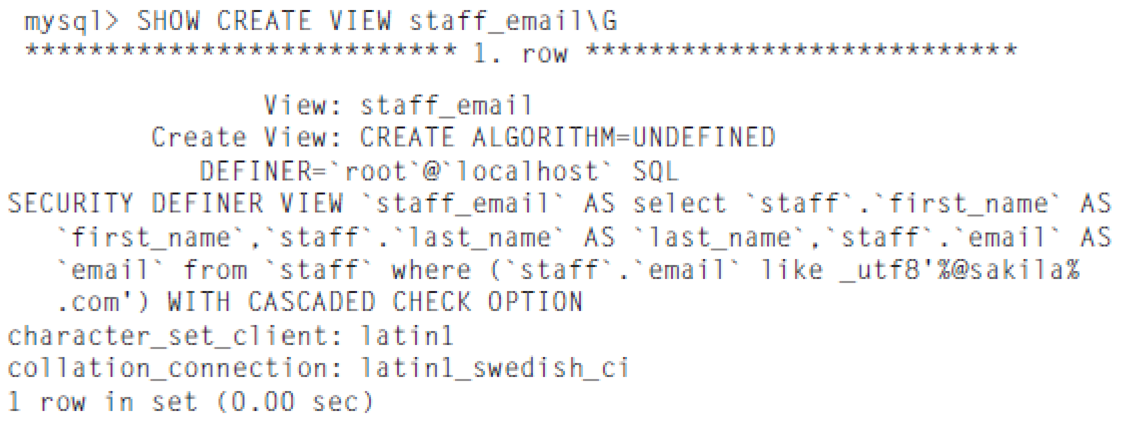
```
mysql> SELECT first_name, last_name, email FROM staff_email;
+------------+-----------+------------------------------+
| first_name | last_name | email                        |
+------------+-----------+------------------------------+
| Mike       | Hillyer   | Mike.Hillyer@sakilastaff.com |
| Jon        | Stephens  | Jon.Stephens@sakilastaff.com |
+------------+-----------+------------------------------+
2 rows in set (0.00 sec)

mysql> select first_name,last_name,email from staff;
+------------+-----------+------------------------------+
| first_name | last_name | email                        |
+------------+-----------+------------------------------+
| Mike       | Hillyer   | Mike.Hillyer@sakilastaff.com |
| Jon        | Stephens  | Jon.Stephens@sakilastaff.com |
+------------+-----------+------------------------------+
2 rows in set (0.00 sec)
```

Ziesel successfully used the **WITH CHECK OPTION** of a view to simulate a check constraint. In the latter example, she was able to make sure that staff members' e-mail addresses were limited to a domain of **sakila[something].com** (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

WITH CHECK OPTION on views that reference other views

By default, **WITH CHECK OPTION** will check all filters against all underlying views. In other words, if a view definition references another view, **WITH CHECK OPTION** will cascade and check all the **WHERE** clauses of all the underlying views (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

You can specify that a view checks only its own definition and not that of any underlying views by specifying **WITH LOCAL CHECK OPTION**. If the **LOCAL** keyword is not specified, the view definition is stored containing **WITH CASCADED CHECK OPTION** (Ramakrishhan, Raghu and Gehrke,Johannes,2018):



```
 mysql> SHOW CREATE VIEW staff_email\G
 *************************** 1. row ***************************

                View: staff_email
         Create View: CREATE ALGORITHM=UNDEFINED
             DEFINER=`root`@`localhost` SQL
SECURITY DEFINER VIEW `staff_email` AS select `staff`.`first_name` AS
   `first_name`,`staff`.`last_name` AS `last_name`,`staff`.`email` AS
   `email` from `staff` where (`staff`.`email` like _utf8'%@sakila%
   .com') WITH CASCADED CHECK OPTION
character_set_client: latin1
collation_connection: latin1_swedish_ci
1 row in set (0.00 sec)
```

Lesson 5. Updatable views

As shown earlier, views are not queried using only **SELECT**; it is possible to use DML on a view as well. As another example, if Benjamin needs to update a staff member's address, he can run the following query (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

```
mysql> UPDATE staff_list SET address="20 Workhaven Lane"
    -> WHERE ID=1;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT ID, name, address, zip code, phone, city, country, SID
    -> FROM staff_list\G
*************************** 1. row ***************************
      ID: 1
    name: Mike Hillyer
 address: 20 Workhaven Lane
zip code:
   phone: 14033335568
    city: Lethbridge
 country: Canada
     SID: 1
*************************** 2. row ***************************
      ID: 2
    name: Jon Stephens
 address: 1411 Lillydale Drive
zip code:
   phone: 6172235589
    city: Woodridge
 country: Australia
     SID: 2
2 rows in set (0.00 sec)
```

INSERT and **DELETE** statements work the same way. Benjamin is allowed to update the **VIEW** because the **VIEW** was defined with **SQL SECURITY DEFINER** and **DEFINER=root@localhost**. If Benjamin was not allowed to update the base tables and **SQL SECURITY INVOKER** was defined, he would not be allowed to use **UPDATE**, **INSERT**, or **DELETE** statements on the view (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The **INFORMATION_SCHEMA** shows whether a view is marked updatable:

```
mysql> USE INFORMATION_SCHEMA
Database changed
mysql> SELECT table_schema, table_name, is_updatable FROM views
    -> WHERE table_name IN ('staff_email','staff_list');
+--------------+-------------+--------------+
| table_schema | table_name  | is_updatable |
+--------------+-------------+--------------+
| sakila       | staff_email | YES          |
| sakila       | staff_list  | YES          |
+--------------+-------------+--------------+
2 rows in set (0.00 sec)
```

Updatable view limitations

To use INSERT, UPDATE, and DELETE on a view, base tables must have a one-to-one relationship with their analogous rows in the view. The staff_email view adhered to this rule. This means that a view is not updatable if the definition contains (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

- Aggregations such as COUNT, SUM, MIN, MAX
- GROUP BY or GROUP BY...HAVING
- SELECT DISTINCT
- UNION ALL or UNION
- Joins that do not have a one-to-one relationship

Additionally, there are other limitations that render a view non-updatable:

- Additionally, there are other limitations that render a view non-updatable:
    - If only literal and constant values are used, because there is nothing to update.
- The FROM clause references a non-updatable view

If an underlying view is non-updatable, so is any view referencing it, becauseupdates actually happen on base tables, not views

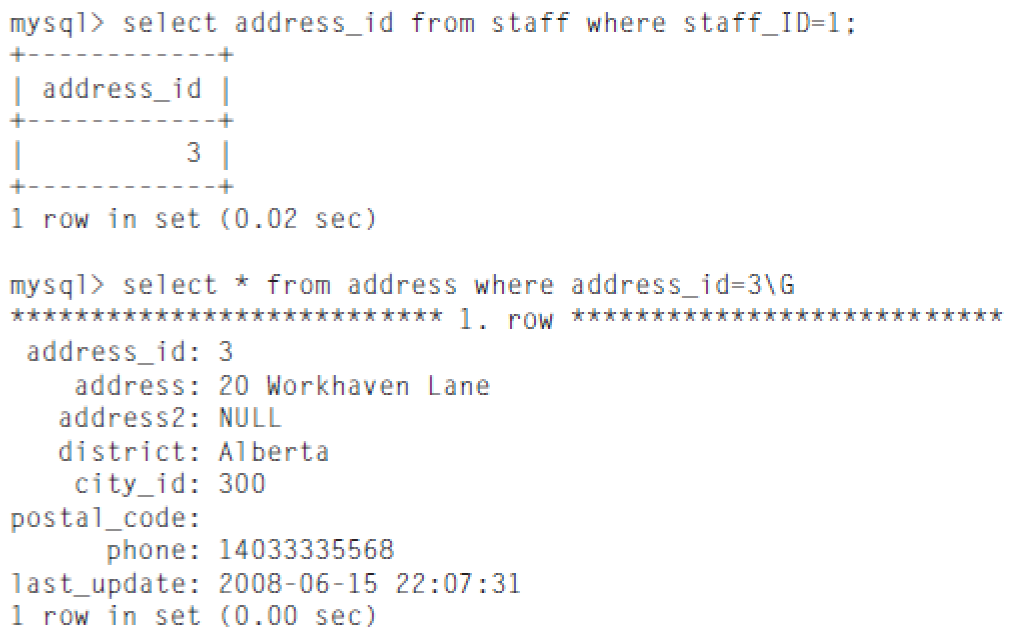
- The SELECT clause contains a subquery
    - MySQL does not allow a table to be both changed and selected from in a subquery.
- ALGORITHM=TEMPTABLE
    - If an intermediate temporary table is used instead of going directly to the underlying base table, the underlying base table cannot be updated

Finally, there are limitations that affect whether an INSERT statement is allowed on a view:

- View field names must be unique.
    - If field names are not unique, an INSERT cannot determine which field to put the corresponding value with.
- The view must be defined with all fields in the base table(s) that have no default values.
    - There is no way to specify values for base tables when inserting into a view, so there must be default values for all fields in base tables that are not in the view.
- Views with calculated fields.
    - There is no way to reverse the calculation to obtain the right value to insert into a base table.

Updatable view problems

Because a view abstracts the data, updating information may not result in the desired behavior. Benjamin submitted what looked like a simple update to Mike Hillyer's address using the **staff_list view**. In reality, though, Benjamin changed a row in the base table **address** (Ramakrishhan, Raghu and Gehrke,Johannes,2018):



```
mysql> select address_id from staff where staff_ID=1;
+------------+
| address_id |
+------------+
|          3 |
+------------+
1 row in set (0.02 sec)

mysql> select * from address where address_id=3\G
*************************** 1. row ***************************
 address_id: 3
    address: 20 Workhaven Lane
   address2: NULL
   district: Alberta
    city_id: 300
postal_code:
      phone: 14033335568
last_update: 2008-06-15 22:07:31
1 row in set (0.00 sec)
```

Earlier, we showed that Benjamin successfully changed Mike Hillyer's address using the **staff_list view**. However, the address table is separate from the **staff** table. Benjamin's **UPDATE** statement actually changed the address associated with **address_id=3**, which may or may not be associated with other staff members — or customers, because the customer table also has an **address_id** reference (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

What if Mike Hillyer was moving from one apartment shared with a customer, to another apartment shared with a staff member? Table 4.1 shows a sample list of addresses and staff members before any update (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

If Mike Hillyer moves to into Ashton Kramer's apartment, the desired change is that Mike Hillyer's row in the staff table contains an address_id of 5. If the UPDATE was done as in the preceding example, Mike Hillyer would still have an address_id of 3, and there would be a side effect of Rachel Cosloy's address being changed even though she did not move. This discrepancy is shown in Table 4.2 (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Table 4.1 Addresses Before Any Update

address_id	Address	Resident
3	23 Workhaven Lane	Mike Hillyer
3	23 Workhaven Lane	Rachel Cosloy
4	1411 Lillydale Drive	Jon Stephens
5	1411 Lillydale Drive	Ashton Kramer

Table 4.2 INCORRECT: Addresses After Updating the View

address_id	Address	Resident
3	1913 Hanoi Way	Mike Hillyer
3	1913 Hanoi Way	Rachel Cosloy
4	1411 Lillydale Drive	Jon Stephens
5	1913 Hanoi Way	Ashton Kramer

Benjamin does not have underlying permissions to the base tables, and can only run an UPDATE statement on the view, not the base tables. The appropriate update is to the staff table, changing Mike Hillyer's address_id to 5, as shown in Table 4.3.

Table 4.3 CORRECT: Addresses After Updating the Base Table

address_id	Address	Resident
5	1913 Hanoi Way	Mike Hillyer
3	23 Workhaven Lane	Rachel Cosloy
4	1411 Lillydale Drive	Jon Stephens
5	1913 Hanoi Way	Ashton Kramer

Note that updating the view presents a logical error. Even if Benjamin had privileges to the base tables, updating the view would generate no warning or error, but produces incorrect results (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

Changing a View Definition

There are two ways to change a view. One method has already been discussed — using the **CREATE OR REPLACE** when defining a view. In addition, MySQL has an **ALTER VIEW** command. **ALTER VIEW** works much like **ALTER TABLE**. The **SELECT** statement that defines the view must always be included in the **ALTER VIEW** statement, even if that part of the view definition is not being modified (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

You may have noticed that in the **CREATE VIEW** statement, four different clauses may come between the words **CREATE** and **VIEW**:

```
CREATE
[OR REPLACE]
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
[DEFINER = { user | CURRENT_USER} ]
[SQL SECURITY { DEFINER | INVOKER }]
VIEW view_name [(column_list)]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

Similarly, the syntax of the **ALTER VIEW** statement is:

```
ALTER
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
[DEFINER = { user | CURRENT_USER} ]
[SQL SECURITY { DEFINER | INVOKER }]
VIEW view_name [(column_list)]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

Only the view's definer or a user with the **SUPER** privilege can **ALTER** a view.

Replication and Views

In both row- and statement-based replication, MySQL treats a view the same way it treats a base table. In statement-based replication, **CREATE VIEW, ALTER VIEW**, and **DROP VIEW** statements are written to the binary log, and thus replicated. In row-based replication, the underlying data is replicated (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

The replicate-do-table and replicate-ignore-table replication options are applied to views and tables in both statement- and row-based replication, with the following outcomes (Ramakrishhan, Raghu and Gehrke,Johannes,2018):

- A view that matches a replicate-do-table pattern will be written to the binary log, and thus replicated.
- A view that matches a replicate-ignore-table pattern will not be written to the binary log, and thus not be replicated.
- replicate-do-table and replicate-ignore-table patterns match the object name only. Therefore, those options will only be applied to views matching the pattern— even if the view references a matching table name.

For additional information and materials:

https://www.youtube.com/watch?v=DCp0oFVG_fk

https://www.youtube.com/watch?v=zUj1dH4IFPI

https://www.youtube.com/watch?v=gxmqJBrVkZE

https://www.youtube.com/watch?v=wRVl2gbgLpk

Assessment Task

Based from your understanding, discuss and elaborate what is the purpose of using views and how it will affect the database schema.

Summary

In this chapter, we have described:

- How to create, change, and drop views

  Invalid SELECT statements in view definitions
- Using views to limit field and row data for security purposes
- How views can simplify and abstract queries
- Performance implications of views
- Using views as check constraints
- How to update underlying base tables using an updatable view
  - Reasons a view may not be updatable
  - Logical errors that may occur when updating data using an updatable view
- How replication handles views

References

Ramakrishhan, Raghu and Gehrke,Johannes (2018) ® Database Management Systems. McGraw Hill

(Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Source :SQL for beginners: A guide to study SQL programming and database management systems

Database management systemsGupta, G. K.,2018

- END OF THE PRELIM TERM MODULE -

CHECK YOUR EXAM SCHEDULE FOR THIS COURSE.

DO NOT FORGET TO TAKE THE EXAM AS SCHEDULED.

THANK YOU AND GOD BLESS