THE KEY TOWARDS

SQL

DBMS

ASAD ASHRAF KAREL

Basic syntaxes AND concepts of SQL

Structure of the syntax:

SET sql_mode = only_full_group_by; (Before running group by function, run this syntax**)**

SET sql_mode = ‘ ’**;** To remove above setting.

- SELECT * FROM <table> WHERE <condition> GROUP BY <column_name> ORDER BY <column_name> ;
- SELECT * FROM <table> GROUP BY <column_name> HAVING <aggregate_function> OREDR BY <column_name> <key_word> ;
- SELECT * FROM <table> GROUP BY <column_name> HAVING <aggregate_function> OREDR BY <column_name> <key_word> LIMIT <int>;

Creating databse:

```
CRAETE TABLE  Student(

        student_name VARCHAR(100) NOT NULL,

        student_gender CHAR(1) NOT NULL,
        mail_id VARCHAR(100) UNIQUE,
        mob_num BIGINT  UNIQUE
);
DESCRIBE student;
```

DML IN DBMS:

```
CREATE DATABASE DML;
USE DML;
```

TBALE 1:

```
CREATE TABLE GL_STUDENTS
(
GL_CODE INT PRIMARY KEY,
SNAME VARCHAR(50),
GRAD_MARK INT,
EMAIL VARCHAR(30) UNIQUE,
MOB_NUMBER CHAR(10) NOT NULL UNIQUE,
LOCATION VARCHAR(30),
CONSTRAINT GL_STUDENTS_CHK CHECK(LOCATION IN ('MUMBAI', 'PUNE', 'CHENNAI', 'GURGAO'))
);
```

TABLE2:

```
CREATE TABLE SCORES
(
GL_CODE INT PRIMARY KEY REFERENCES GL_STUDENTS (GL_CODE),

MODULE1 INT DEFAULT 0,

MODULE2 INT DEFAULT 0,

MODULE3 INT DEFAULT 0

);
```

Because, this PRIMARY KEY is FOREIGN KEY for first table. If we forgot to mention this as foreign key, we are just assigning it as a FOREIGN KEY. It must be before inserting values.

```
ALTER TABLE SCORES ADD FOREIGN KEY (GL_CODE) REFERENCES GL_STUDENTS (GL_CODE);   [A]
```

If we want to get some restrictions on any column. we even can alter the situation.

```
ALTER TABLE SCORES ADD CHECK (MODULE1 > 60);
```

Now this will allow marks to MODULE3 which are greater than 60.

Like this, we can alter table as many possible times.

```
ALTER TABLE TRANSACTION ADD UNIQUE (IFSC_CODE);
```

TABLE 3:

```
CREATE TABLE GL_PLACEMENTS

(

GL_CODE INT ,

COMPANY VARCHAR(30),

STATUS VARCHAR(10),

CONSTRAINT GL_PLACEMENTS_PK PRIMARY KEY (GL_CODE, COMPANY),

CONSTRAINT GL_PLACEMENTS_FK FOREIGN KEY (GL_CODE) REFERENCES GL_STUDENTS (GL_CODE)

);
```

TABLE 3 has some uncertainties to declare its primary key, because none of the column is singularly capable to become a primary key. At least two columns are capable to become a primary key. Hence we declare 2 columns as primary key. These keys are called **composite key**. Here in above table, GL_CODE & COMPANY is set as primary key.

GL_CODE is still running from all tables, which is even referred here as a primary key. So to make a link of this TABLE 3 with previous, we just make is referential key (FOREIGN KEY).

UPDATING DML:

Without breaking any rule defined above, inserting vales will be applicable, otherwise error will be thrown up.

After making tables, now it is not possible to violet any table. We cannot update or delete the table, whether it would be parent table or child table. Hence there must be any way to do so, at least on parent table. Hence we do certain tasks:

We want a system such that, if I delete something from parent table, automatically child table respective should also be get deleted.

```
ALTER TABLE SCORES DROP FOREIGN KEY SCORES_IBFK_1;
```

Now [A] got affected. Now command it to allow the changes. Still the table and column do exists, but the constraint FOREIGN KEY is removed.

```
ALTER TABLE SCORES ADD FOREIGN KEY (GL_CODE) REFERENCES GL_STUDENTS(GL_CODE) ON UPDATE CASCADE;
```

Or just do initially:

```
CREATE TABLE SCORES_1

(

GL_CODE INT PRIMARY KEY REFERENCES GL_STUDENTS (GL_CODE) ON UPDATE CASCADE ON DELETE CASCADE,

MODULE1 INT DEFAULT 0,

MODULE2 INT DEFAULT 0,

MODULE3 INT DEFAULT 0

);                                          [B]
```

FOLLOWINGS ARE THE METHODS TO DO SO:

- CREATED TABLE
- INSERTED VALUES
- CREATED ANOTHER TABLE WITH CASCADE CLAUSE LIKE **[B]**.
- INSERTED THE VALUES.
- NOW UPDATED THE PARENT TABLE. CHILD TABLE AUTOMATICALLY GOT UPDATED.

EXAMPLE:

```
CREATE TABLE A1
(
RN INT PRIMARY KEY ,
NAME VARCHAR(10),
AGE INT NOT NULL CHECK (AGE>20)
);


CREATE TABLE B1
(
RN INT PRIMARY KEY REFERENCES A1(RN) ON  UPDATE CASCADE ON  DELETE CASCADE,
MARKS INT CHECK (MARKS BETWEEN 0 AND 100),
FOREIGN KEY (RN) REFERENCES A1 (RN) ON UPDATE CASCADE ON DELETE CASCADE
);


INSERT INTO A1 VALUES (1, 'ASAD', 25);


INSERT INTO B1 VALUES (1, 45);


SET SQL_SAFE_UPDATES=0;
UPDATE  A1 SET RN= 4 WHERE RN=1;
```

```
DELETE FROM A1 WHERE RN=4;

SELECT * FROM B1;
```

VIEW IN DBMS:

Simple view:

```
CREATE OR REPLACE VIEW first_view AS SELECT * FROM employees;
```

Complex view:

```
CREATE VIEW emp_dep AS SELECT first_name, department_name, salary, hire_date

FROM employees e, departments d WHERE e.department_id = d.department_id;

UPDATE employees SET first_name = 'R. David' WHERE first_name = 'David';
```

Here the original dataset has been changed, for sure the derived table is also changed along the same scenario.

Horizontal view:

```
CREATE VIEW IT_Programmers AS SELECT * FROM employees WHERE job_id = 'IT_PROG';
```

Vertical view:

```
CREATE VIEW  emp_contact AS SELECT first_name, last_name, email, phone_number FROM employees;
```

Row-column view:

```
CREATE VIEW dept_contact AS SELECT first_name, last_name, email, phone_number

FROM employees WHERE department_id =50;
```

Group view:

```
CREATE VIEW count_dept AS SELECT department_id, count(*) AS count FROM employees GROUP BY department_id;
```

DML with views:

Simple view:

```
CREATE VIEW s1 AS SELECT * FROM student;

INSERT INTO s1 VALUES (30, 'XYZ',60,'xy@dfd.com','M');
```

AND

```
UPDATE s1 SET grad_marks =80 WHERE roll=30;
```

Some other operations:

```
CREATE VIEW s3 AS SELECT * FROM student WHERE grad_marks > 40;

INSERT INTO s3 VALUES (35, 'Name1',30,'nam@gg','M');
```

No error even if grad_marks are less than 40. The original table is just changed.

Check option:

```
CREATE VIEW s4 AS SELECT * FROM student WHERE grad_marks>50 WITH CHECK OPTION;

INSERT INTO s4 VALUE (40, 'XYZ',40,'hyd@dfd.com','M');
```

Check option error as marks are less than 50.

Check local:

```
CREATE VIEW s5 AS SELECT * FROM s4 WHERE gender = 'M' WITH LOCAL CHECK OPTION;
```

```
INSERT INTO s5 VALUES (21,'Fun',55,'ggg@dg','F');
```

Error check option failed as gender is 'F'

```
INSERT INTO s5 VALUES (31,'Fun',55,'gg@dg','M');
```

Executed

```
INSERT INTO s5 VALUES (45,'Fun',45,'ggd@dg','M');
```

Error check option failed as marks are less than 50.

```
INSERT INTO s4 VALUES (22, 'Toy',65, 'mail', 'F');
```

Executed

Show all views in database:

```
SELECT * FROM  information_schema.views WHERE table_schema = 'hr';
```

Copying table from another database:

```
CRAETE TABLE <new_table> (SELECT * FROM <database>.<table>);
```

Inserting values into the column

```
INSERT INTO Student

 VALUES (100, 'Amit_Sharma', 'M',null, 8987654534);

INSERT INTO  Student

VALUES (101, 'Amita_Sharma', 'F',null, 9087654534);
```

Copy dataset

```
CREATE TABLE Stud1

SELECT * FROM Student;
```

Deleting row

```
DELETE FROM <table> WHERE <condition>
```

Dropping column

```
ALTER TABLE <Table>  DROP <column>
```

Replacing

```
SELECT *,REPLACE(geo_location, '_','  ' ) FROM bank_inventory;
```

Renaming table

```
ALTER TABLE <table> RENAME <new_name>;
```

Rename column

```
ALTER TABLE <table> RENAME COLUMN <column_name> TO <new_name> ;
```

Adding column into the table

```
ALTER TABLE <table_name>   ADD  <column_name>   VARCHAR(10)  NOT NULL;

ALTER  TALE  USER  ADD  <column_name>  BOOLEAN  DEFAULT  TRUE;
```

Updating values into the database:

```
SET sql_safe_updates =0;

UPDATE emp_data

SET last_name='hathway' WHERE emp_id=1;
```

emp_data - Table

last_name - Column_name

emp_id - Condition, <where the value has to be changed>

Creating/ dropping index

CREATE INDEX <index_name> ON <table> (column_name);

DROP INDEX <index_name> ON <table> ;

SHOW INDEX FROM <table> ; To see the index description. Even primary key is kind of index by default.

Aggregate functions in mysql

SUM( ), AVG( ), MIN( ), MAX( ), COUNT( ), MEDIAN( ), VARIANCE( ), STDDEV( ) etc

To check these aggregate functions, we cannot use WHERE filter condition, we must have to use HAVING clause. WHERE clause is used for non-aggregate functions only.

Set operators

UNION UNION ALL

Time

World time table:

https://www.worldtimebuddy.com/

SELECT DAY (NOW( )), MONTH (NOW( )), YEAR (NOW( )), HOUR (NOW( )), MINUTE (NOW( )), SECOND (NOW( ));

SELECT CURRENT_TIMESTAMP ( );

Good stuff to understand Date_format:

https://www.mysqltutorial.org/mysql-date_format/#:~:text=To%20format%20a%20date%20value%20to%20a%20specific,table%20below%20for%20a%20list%20of%20predefined%20

Converting string into date:

```
DATE_FORMAT(STR_TO_DATE(<column_name>,'%d-%m-%Y'),'%d-%m-%Y')
```

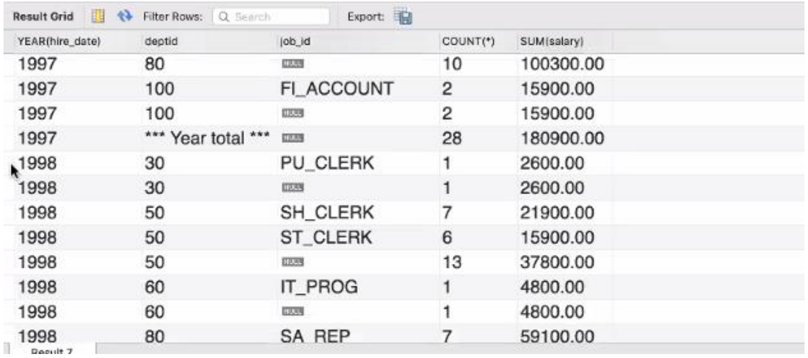
CONVERTING TEXT INTO DATE IN DATASET:

```
UPDATE <table_name>

SET <column_name> = STR_TO_DATE(<column_name>, '%d-%m-%Y');



ALTER TABLE <table_name>

MODIFY <column_name> DATE;
```

Roll-up:

```
SELECT YEAR(hire_date),
IFNULL(department_id, '*** Year total ***') AS deptid, COUNT(*), SUM(salary)
FROM Employees
GROUP BY YEAR(hire_date), department_id WITH ROLLUP;
```

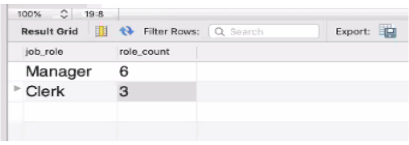
Output:



YEAR(hire_date)	deptid	job_id	COUNT(*)	SUM(salary)
1997	80	NULL	10	100300.00
1997	100	FI_ACCOUNT	2	15900.00
1997	100	NULL	2	15900.00
1997	*** Year total ***	NULL	28	180900.00
1998	30	PU_CLERK	1	2600.00
1998	30	NULL	1	2600.00
1998	50	SH_CLERK	7	21900.00
1998	50	ST_CLERK	6	15900.00
1998	50	NULL	13	37800.00
1998	60	IT_PROG	1	4800.00
1998	60	NULL	1	4800.00
1998	80	SA_REP	7	59100.00

Case-Group_by:

```
SELECT CASE
            WHEN job_title LIKE '%Manager' THEN 'Manager'
            WHEN job_title LIKE '%Clerk' THEN 'Clerk'
END AS job_role, COUNT(*) AS role_count
FROM Jobs
WHERE job_title LIKE '%Manager' OR job_title LIKE '%Clerk'
GROUP BY (CASE
            WHEN job_title LIKE '%Manager' THEN 'Manager'
            WHEN job_title LIKE '%Clerk' THEN 'Clerk'
END);
```

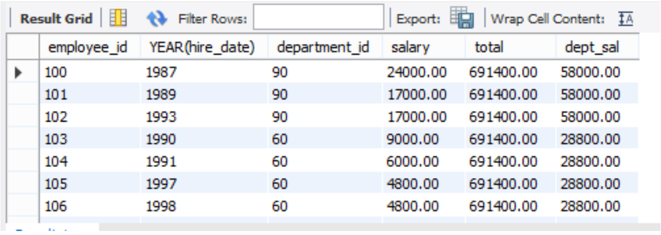


job_role	role_count
Manager	6
Clerk	3

Windows functions in SQL:

Partition by:

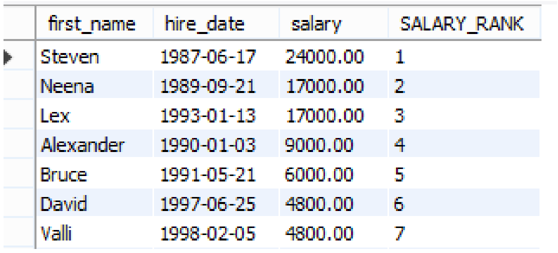
```
SELECT      employee_id,  YEAR(hire_date), department_id, salary,

SUM(salary) OVER() AS total,

SUM(salary) OVER(PARTITION BY department_id) AS dept_sal

FROM employees

ORDER BY employee_id,  year(hire_date), department_id, salary;
```



employee_id	YEAR(hire_date)	department_id	salary	total	dept_sal
100	1987	90	24000.00	691400.00	58000.00
101	1989	90	17000.00	691400.00	58000.00
102	1993	90	17000.00	691400.00	58000.00
103	1990	60	9000.00	691400.00	28800.00
104	1991	60	6000.00	691400.00	28800.00
105	1997	60	4800.00	691400.00	28800.00
106	1998	60	4800.00	691400.00	28800.00

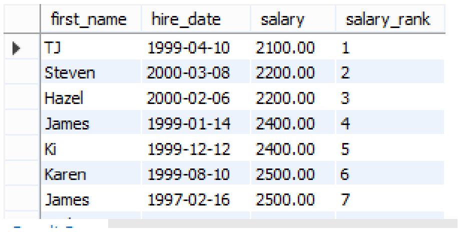
Row_number()

```
SELECT first_name, hire_date,salary FROM employees ORDER BY salary;
SELECT first_name, hire_date, salary,

ROW_NUMBER() OVER() AS SALARY_RANK

FROM employees;
```



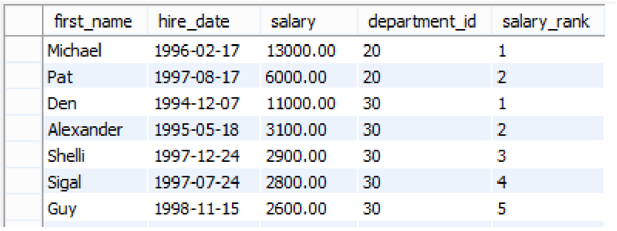
first_name	hire_date	salary	SALARY_RANK
Steven	1987-06-17	24000.00	1
Neena	1989-09-21	17000.00	2
Lex	1993-01-13	17000.00	3
Alexander	1990-01-03	9000.00	4
Bruce	1991-05-21	6000.00	5
David	1997-06-25	4800.00	6
Valli	1998-02-05	4800.00	7

```
SELECT first_name, hire_date, salary,

ROW_NUMBER() OVER( ORDER BY salary) AS salary_rank

FROM employees;
```



first_name	hire_date	salary	salary_rank
TJ	1999-04-10	2100.00	1
Steven	2000-03-08	2200.00	2
Hazel	2000-02-06	2200.00	3
James	1999-01-14	2400.00	4
Ki	1999-12-12	2400.00	5
Karen	1999-08-10	2500.00	6
James	1997-02-16	2500.00	7

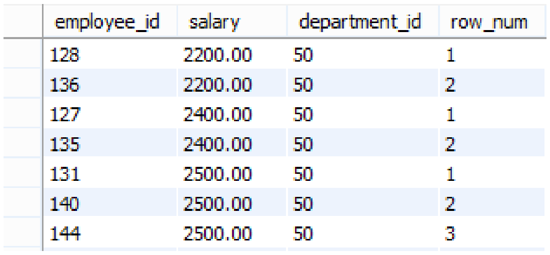
```
SELECT first_name, hire_date, salary, department_id,

ROW_NUMBER() OVER( PARTITION BY department_id) AS salary_rank

FROM employees;
```



first_name	hire_date	salary	department_id	salary_rank
Michael	1996-02-17	13000.00	20	1
Pat	1997-08-17	6000.00	20	2
Den	1994-12-07	11000.00	30	1
Alexander	1995-05-18	3100.00	30	2
Shelli	1997-12-24	2900.00	30	3
Sigal	1997-07-24	2800.00	30	4
Guy	1998-11-15	2600.00	30	5

Unique rows with row_number()

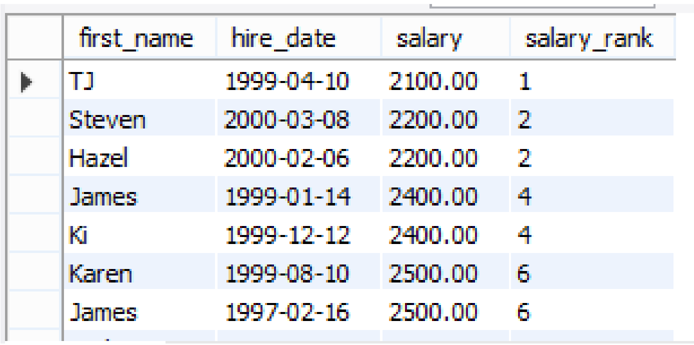
```
CREATE TABLE temp1 (
  id INT,
  name VARCHAR(10) NOT NULL
);
INSERT INTO temp1(id,name)
VALUES(1,'A'),
   (2,'B'),
   (3,'C'),
   (4,'D');
SELECT * FROM temp1;


SELECT employee_id, salary, department_id,
ROW_NUMBER() OVER (PARTITION BY department_id, salary ORDER BY salary) AS row_num
FROM employees;
```



employee_id	salary	department_id	row_num
128	2200.00	50	1
136	2200.00	50	2
127	2400.00	50	1
135	2400.00	50	2
131	2500.00	50	1
140	2500.00	50	2
144	2500.00	50	3

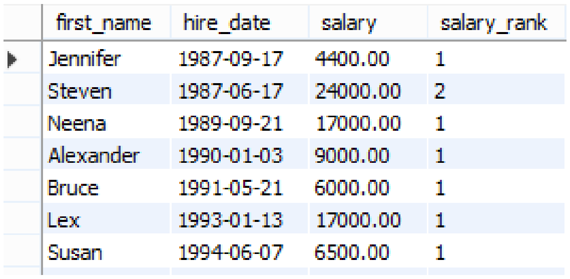
Rank()

```
SELECT first_name, hire_date, salary,
RANK() OVER( ORDER BY salary) AS salary_rank
FROM employees;
```



first_name	hire_date	salary	salary_rank
TJ	1999-04-10	2100.00	1
Steven	2000-03-08	2200.00	2
Hazel	2000-02-06	2200.00	2
James	1999-01-14	2400.00	4
Ki	1999-12-12	2400.00	4
Karen	1999-08-10	2500.00	6
James	1997-02-16	2500.00	6

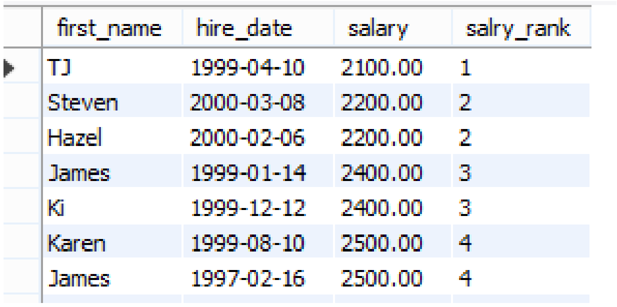
```
SELECT first_name, hire_date, salary,

RANK() over( PARTITION BY YEAR(hire_date) ORDER BY salary ) AS salary_rank

FROM employees;
```



first_name	hire_date	salary	salary_rank
Jennifer	1987-09-17	4400.00	1
Steven	1987-06-17	24000.00	2
Neena	1989-09-21	17000.00	1
Alexander	1990-01-03	9000.00	1
Bruce	1991-05-21	6000.00	1
Lex	1993-01-13	17000.00	1
Susan	1994-06-07	6500.00	1

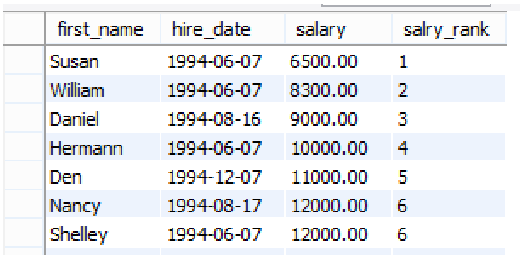
Dense_rank()

```
SELECT first_name, hire_date, salary,

DENSE_RANK() OVER( order by salary) AS salry_rank

FROM employees;
```



first_name	hire_date	salary	salry_rank
TJ	1999-04-10	2100.00	1
Steven	2000-03-08	2200.00	2
Hazel	2000-02-06	2200.00	2
James	1999-01-14	2400.00	3
Ki	1999-12-12	2400.00	3
Karen	1999-08-10	2500.00	4
James	1997-02-16	2500.00	4

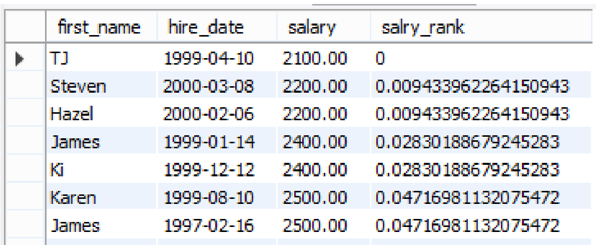
```
SELECT first_name, hire_date, salary,
DENSE_RANK() OVER( PARTITION BY YEAR(hire_date) ORDER BY salary) AS salry_rank

FROM employees;
```



first_name	hire_date	salary	salry_rank
Susan	1994-06-07	6500.00	1
William	1994-06-07	8300.00	2
Daniel	1994-08-16	9000.00	3
Hermann	1994-06-07	10000.00	4
Den	1994-12-07	11000.00	5
Nancy	1994-08-17	12000.00	6
Shelley	1994-06-07	12000.00	6

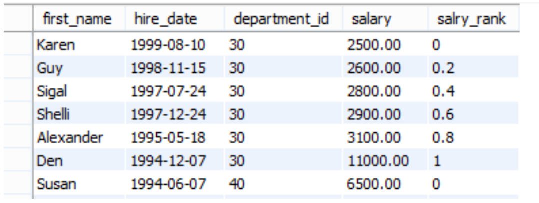
Percent_rank()

```
SELECT first_name, hire_date, salary,

PERCENT_RANK() OVER( ORDER BY salary ) AS salry_rank

FROM employees;
```



first_name	hire_date	salary	salry_rank
TJ	1999-04-10	2100.00	0
Steven	2000-03-08	2200.00	0.009433962264150943
Hazel	2000-02-06	2200.00	0.009433962264150943
James	1999-01-14	2400.00	0.02830188679245283
Ki	1999-12-12	2400.00	0.02830188679245283
Karen	1999-08-10	2500.00	0.04716981132075472
James	1997-02-16	2500.00	0.04716981132075472

```
SELECT first_name, hire_date, department_id, salary,

PERCENT_RANK() OVER( PARTITION BY department_id ORDER BY salary) AS salry_rank

FROM employees;
```

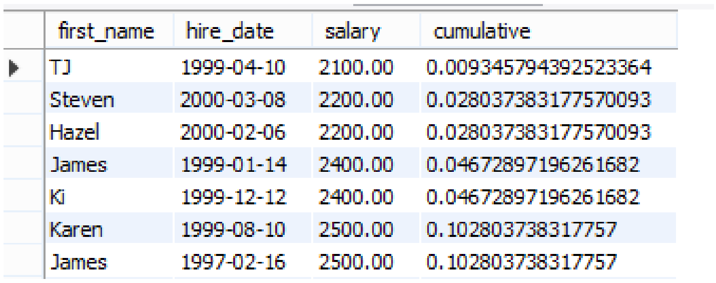


first_name	hire_date	department_id	salary	salry_rank
Karen	1999-08-10	30	2500.00	0
Guy	1998-11-15	30	2600.00	0.2
Sigal	1997-07-24	30	2800.00	0.4
Shelli	1997-12-24	30	2900.00	0.6
Alexander	1995-05-18	30	3100.00	0.8
Den	1994-12-07	30	11000.00	1
Susan	1994-06-07	40	6500.00	0

cume_dist()

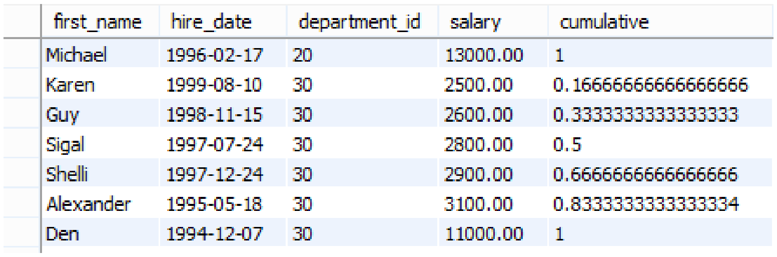
```
SELECT first_name, hire_date, salary,

CUME_DIST() OVER (ORDER BY salary) AS cumulative
```

```
FROM employees;
```



	first_name	hire_date	salary	cumulative
▶	TJ	1999-04-10	2100.00	0.009345794392523364
	Steven	2000-03-08	2200.00	0.028037383177570093
	Hazel	2000-02-06	2200.00	0.028037383177570093
	James	1999-01-14	2400.00	0.04672897196261682
	Ki	1999-12-12	2400.00	0.04672897196261682
	Karen	1999-08-10	2500.00	0.102803738317757
	James	1997-02-16	2500.00	0.102803738317757

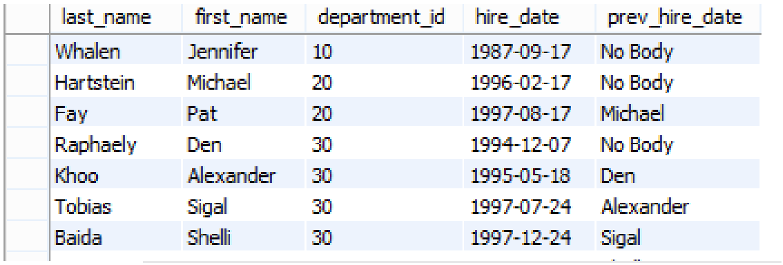
```
SELECT first_name, hire_date, department_id, salary,

CUME_DIST() OVER (partition by department_id ORDER BY salary) AS cumulative

FROM employees;
```



first_name	hire_date	department_id	salary	cumulative
Michael	1996-02-17	20	13000.00	1
Karen	1999-08-10	30	2500.00	0.16666666666666666
Guy	1998-11-15	30	2600.00	0.3333333333333333
Sigal	1997-07-24	30	2800.00	0.5
Shelli	1997-12-24	30	2900.00	0.6666666666666666
Alexander	1995-05-18	30	3100.00	0.8333333333333334
Den	1994-12-07	30	11000.00	1

Lag()

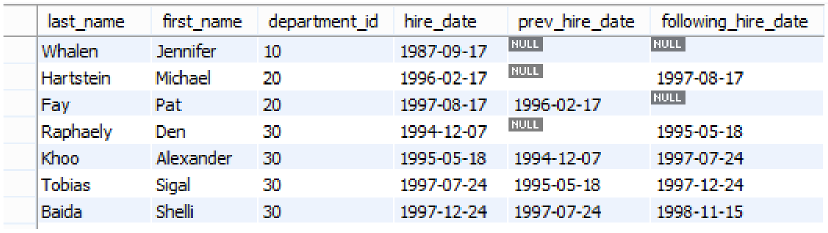
```
SELECT last_name, first_name, department_id, hire_date,

     LAG(first_name, 1, "No Body") OVER (PARTITION BY department_id

                    ORDER BY hire_date) prev_hire_date

  FROM employees

 ORDER BY department_id, hire_date, last_name, first_name;
```



last_name	first_name	department_id	hire_date	prev_hire_date
Whalen	Jennifer	10	1987-09-17	No Body
Hartstein	Michael	20	1996-02-17	No Body
Fay	Pat	20	1997-08-17	Michael
Raphaely	Den	30	1994-12-07	No Body
Khoo	Alexander	30	1995-05-18	Den
Tobias	Sigal	30	1997-07-24	Alexander
Baida	Shelli	30	1997-12-24	Sigal

Lead()

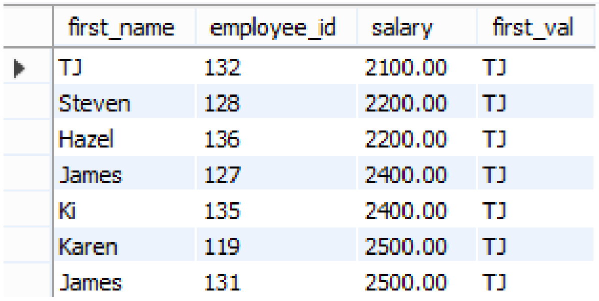
```
SELECT last_name, first_name, department_id, hire_date,
     LAG(hire_date, 1, null) OVER (PARTITION BY department_id
                    ORDER BY hire_date) prev_hire_date,
     LEAD(hire_date, 1, null) OVER (PARTITION BY department_id
                    ORDER BY hire_date) following_hire_date
  FROM employees
ORDER BY department_id, hire_date, last_name, first_name;
```



last_name	first_name	department_id	hire_date	prev_hire_date	following_hire_date
Whalen	Jennifer	10	1987-09-17	NULL	NULL
Hartstein	Michael	20	1996-02-17	NULL	1997-08-17
Fay	Pat	20	1997-08-17	1996-02-17	NULL
Raphaely	Den	30	1994-12-07	NULL	1995-05-18
Khoo	Alexander	30	1995-05-18	1994-12-07	1997-07-24
Tobias	Sigal	30	1997-07-24	1995-05-18	1997-12-24
Baida	Shelli	30	1997-12-24	1997-07-24	1998-11-15

first_value()

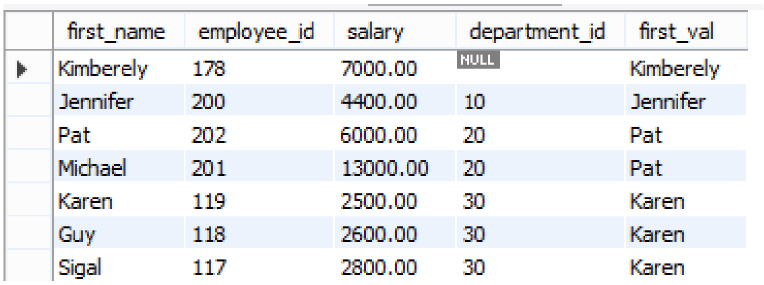
```
SELECT first_name, employee_id, salary,
FIRST_VALUE(first_name) OVER(ORDER BY salary)
FROM employees
ORDER BY salary;
```



first_name	employee_id	salary	first_val
TJ	132	2100.00	TJ
Steven	128	2200.00	TJ
Hazel	136	2200.00	TJ
James	127	2400.00	TJ
Ki	135	2400.00	TJ
Karen	119	2500.00	TJ
James	131	2500.00	TJ

```
SELECT first_name, employee_id, salary, department_id,
FIRST_VALUE(first_name) OVER(PARTITION BY department_id ORDER BY salary)
```

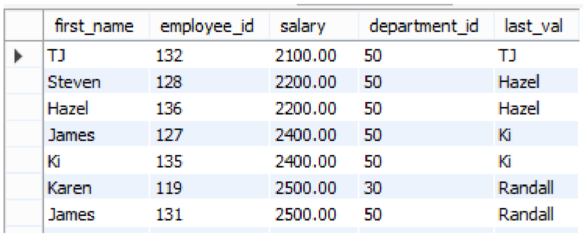
```
 FROM employees

 ORDER BY department_id,salary;
```



first_name	employee_id	salary	department_id	first_val
Kimberely	178	7000.00	NULL	Kimberely
Jennifer	200	4400.00	10	Jennifer
Pat	202	6000.00	20	Pat
Michael	201	13000.00	20	Pat
Karen	119	2500.00	30	Karen
Guy	118	2600.00	30	Karen
Sigal	117	2800.00	30	Karen

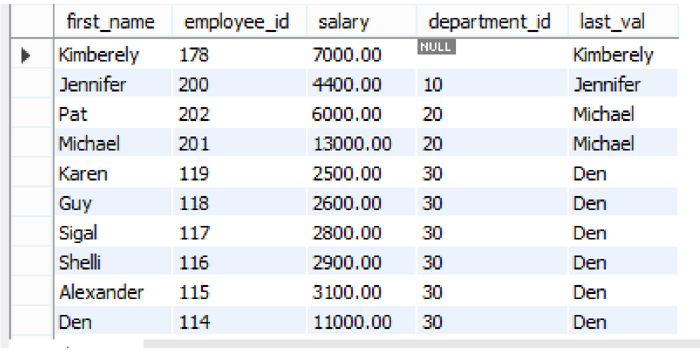
last_value()

```
SELECT first_name, employee_id, salary, department_id,
 LAST_VALUE(first_name) OVER(ORDER BY salary)
 FROM employees
 ORDER BY salary;
```



first_name	employee_id	salary	department_id	last_val
TJ	132	2100.00	50	TJ
Steven	128	2200.00	50	Hazel
Hazel	136	2200.00	50	Hazel
James	127	2400.00	50	Ki
Ki	135	2400.00	50	Ki
Karen	119	2500.00	30	Randall
James	131	2500.00	50	Randall

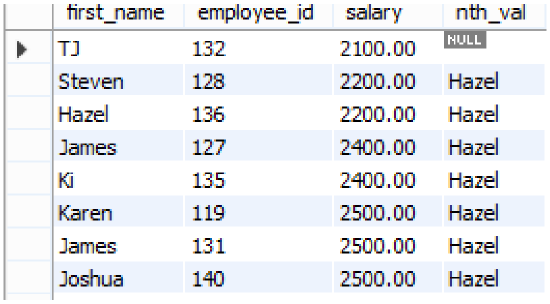
```
 SELECT first_name, employee_id, salary, department_id,

 LAST_VALUE(first_name) OVER(PARTITION BY department_id order by salary RANGE BETWEEN

        UNBOUNDED PRECEDING AND

        UNBOUNDED FOLLOWING)

 FROM employees

 ORDER BY department_id,salary;
```



first_name	employee_id	salary	department_id	last_val
Kimberely	178	7000.00	NULL	Kimberely
Jennifer	200	4400.00	10	Jennifer
Pat	202	6000.00	20	Michael
Michael	201	13000.00	20	Michael
Karen	119	2500.00	30	Den
Guy	118	2600.00	30	Den
Sigal	117	2800.00	30	Den
Shelli	116	2900.00	30	Den
Alexander	115	3100.00	30	Den
Den	114	11000.00	30	Den

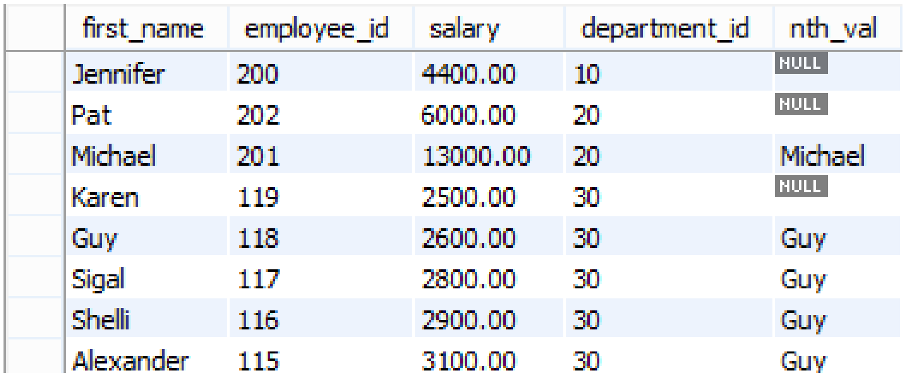
Nth_value()

```
SELECT first_name, employee_id, salary,

NTH_VALUE(first_name,3) OVER(ORDER BY salary)

FROM employees

ORDER BY salary;
```



first_name	employee_id	salary	nth_val
TJ	132	2100.00	NULL
Steven	128	2200.00	Hazel
Hazel	136	2200.00	Hazel
James	127	2400.00	Hazel
Ki	135	2400.00	Hazel
Karen	119	2500.00	Hazel
James	131	2500.00	Hazel
Joshua	140	2500.00	Hazel

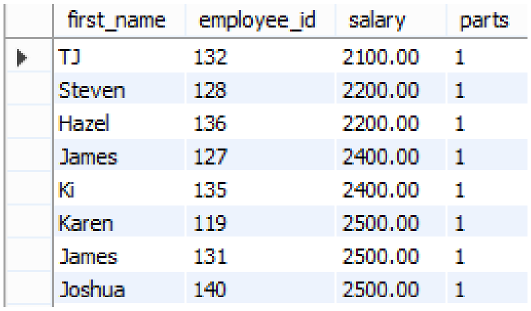
```
SELECT first_name, employee_id, salary, department_id,

NTH_VALUE(first_name,3) OVER(PARTITION BY department_id order by salary)

FROM employees

ORDER BY department_id,salary;
```



first_name	employee_id	salary	department_id	nth_val
Jennifer	200	4400.00	10	NULL
Pat	202	6000.00	20	NULL
Michael	201	13000.00	20	Michael
Karen	119	2500.00	30	NULL
Guy	118	2600.00	30	Guy
Sigal	117	2800.00	30	Guy
Shelli	116	2900.00	30	Guy
Alexander	115	3100.00	30	Guy

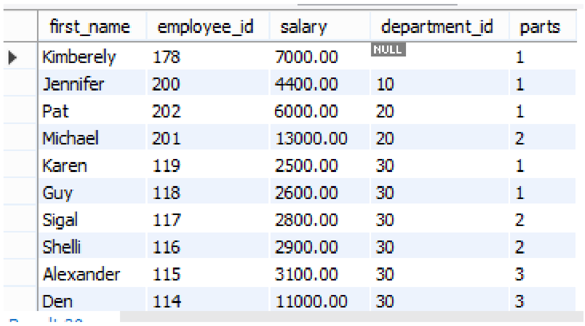
ntile()() **-** This will distribute the dataset into the parts.

```
SELECT first_name, employee_id, salary,

NTILE(3) OVER(ORDER BY salary)

FROM employees ORDER BY salary;
```



first_name	employee_id	salary	parts
TJ	132	2100.00	1
Steven	128	2200.00	1
Hazel	136	2200.00	1
James	127	2400.00	1
Ki	135	2400.00	1
Karen	119	2500.00	1
James	131	2500.00	1
Joshua	140	2500.00	1

```
SELECT first_name, employee_id, salary, department_id,

NTILE(3) OVER(PARTITION BY department_id ORDER BY salary)

FROM employees

ORDER BY department_id,salary;
```



first_name	employee_id	salary	department_id	parts
Kimberely	178	7000.00	NULL	1
Jennifer	200	4400.00	10	1
Pat	202	6000.00	20	1
Michael	201	13000.00	20	2
Karen	119	2500.00	30	1
Guy	118	2600.00	30	1
Sigal	117	2800.00	30	2
Shelli	116	2900.00	30	2
Alexander	115	3100.00	30	3
Den	114	11000.00	30	3

INFORMATION SCHEMA :

```
SELECT CONSTRAINT_NAME,
                    UNIQUE_CONSTRAINT_NAME,
                    MATCH_OPTION,
                    UPDATE_RULE,
                    DELETE_RULE,
                    TABLE_NAME,
                    REFERENCED_TABLE_NAME
 FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
 WHERE CONSTRAINT_SCHEMA = 'HR';
```

TRANSACTIONS:

ACID: Atomicity Consistency Isolation Durability

Atomicity: Either the transaction gets completed, or in case of any issue, complete transaction is ROLLED BACK.

Consistency: Before or after the complete transaction total amount must be equal.

A transacting to B 1000/-, if suppose.

A = 2000/-

1st step is to read A, whether it has sufficient amount to be transacted or not. If it comes true, then with ALU mathematically 1000/- will be subtracted, and the value will be stored in RAM.

B=3000/- CURRENT AMOUNT IN B. (LOCAL MEMORY)

Now complete data would be transferred from LOCAL MEMORY to main memory.

Now A= 1000/- (1000/- debited) and B=4000/-(1000/- credited) which means total amount is 5000/- as we anticipated. This is complete transaction.

Isolation: Not allowed parallel transactions. In short a parallel schedule is converted into serial schedule. Then the transaction is now in **consistency**.

Durability: Every change in database should be permanently saved in hard Dick. Like previous, if the transaction is completed, the change in dataset must be permanent.

TRANSACTION IN ONE'S AMOUT TO OTHERS, A SIMPLE CODE:

```
SET SQL_SAFE_UPDATES =0;

START TRANSACTION;

UPDATE account_details SET balance = balance - 1000 WHERE FIRST_name = 'MONICA';

UPDATE account_details SET balance = balance + 1000 WHERE FIRST_name = 'JOSEPH';

COMMIT;
```

1000/- transferred from MONICA's account to JOSEPH's account.

```
START TRANSACTION;
```

Check bank balance :

```
SELECT Balance_amount FROM BANK_ACCOUNT WHERE Account_Number = '4000-1956-2001' ;
```

Withdraw money from ATM:

```
INSERT INTO BANK_ACCOUNT_Transaction VALUES ('4000-1956-2001' ,  -2300.00,  'ATM Withdrawal' ,
'CA' , NOW() ) ;
```

```
UPDATE BANK_Account SET   balance_amount  = balance_amount - 2300 WHERE  Account_Number =
'4000-1956-2001' ;
```

Bank charges 0.2% on withdrawn money :

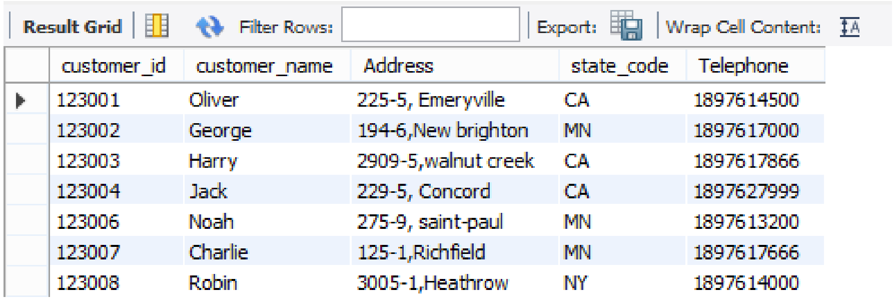
```
INSERT  INTO   BANK_ACCOUNT_Transaction  VALUES  ('4000-1956-2001'  ,   -2300.00  *  0.02,    'ATM
Withdrawal' ,  'CA' , NOW());
```

Update the old balance with new balance:

```
UPDATE  BANK_ACCOUNT  SET   balance_amount  =  balance_amount  -  2300  *  0.02   WHERE
Account_Number = '4000-1956-2001' ;

COMMIT ;
```

SAVEPOINT:

INITIALLY THE TABLE IS LIKE:

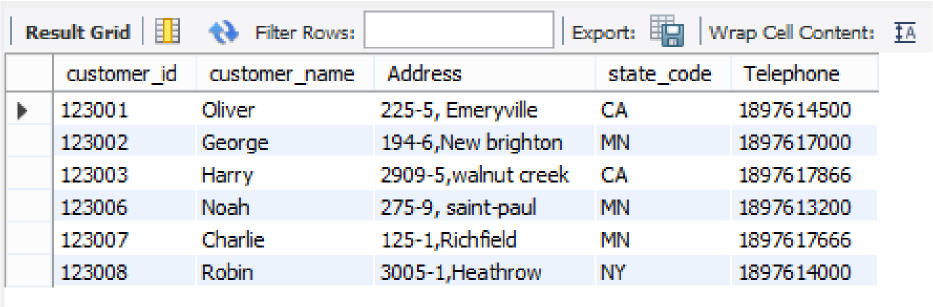


customer_id	customer_name	Address	state_code	Telephone
123001	Oliver	225-5, Emeryville	CA	1897614500
123002	George	194-6,New brighton	MN	1897617000
123003	Harry	2909-5,walnut creek	CA	1897617866
123004	Jack	229-5, Concord	CA	1897627999
123006	Noah	275-9, saint-paul	MN	1897613200
123007	Charlie	125-1,Richfield	MN	1897617666
123008	Robin	3005-1,Heathrow	NY	1897614000

```
SET SQL_SAFE_UPDATES=0;

START TRANSACTION;

SAVEPOINT customer_1;

DELETE FROM bank_CUSTOMER WHERE CUSTomer_ID=123004;

SAVEPOINT customer_2;

DELETE FROM bank_CUSTOMER WHERE CUSTomer_ID=123001;

SAVEPOINT customer_3;
```

```sql
DELETE FROM bank_CUSTOMER WHERE CUSTomer_ID=123002;

ROLLBACK TO customer_2;

COMMIT;
```

AFTER SAVEPOINT THE TABLE BE LIKE:

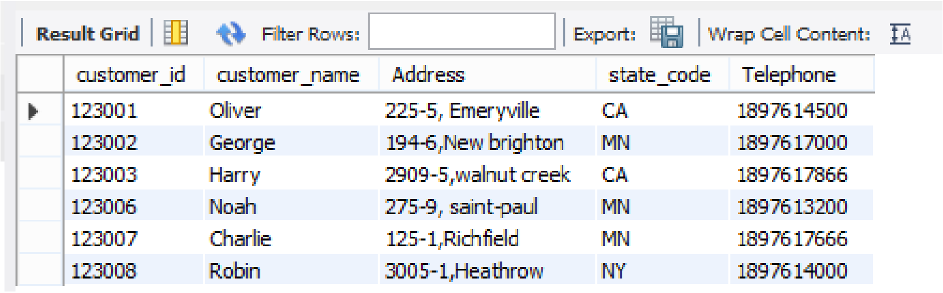


customer_id	customer_name	Address	state_code	Telephone
123001	Oliver	225-5, Emeryville	CA	1897614500
123002	George	194-6,New brighton	MN	1897617000
123003	Harry	2909-5,walnut creek	CA	1897617866
123006	Noah	275-9, saint-paul	MN	1897613200
123007	Charlie	125-1,Richfield	MN	1897617666
123008	Robin	3005-1,Heathrow	NY	1897614000

123004 is deleted.

DELETING SAVEPOINT:

INITIALLY THE TABLE WAS LIKE:

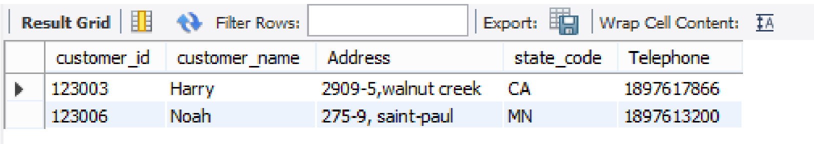


customer_id	customer_name	Address	state_code	Telephone
123001	Oliver	225-5, Emeryville	CA	1897614500
123002	George	194-6,New brighton	MN	1897617000
123003	Harry	2909-5,walnut creek	CA	1897617866
123006	Noah	275-9, saint-paul	MN	1897613200
123007	Charlie	125-1,Richfield	MN	1897617666
123008	Robin	3005-1,Heathrow	NY	1897614000

```sql
START TRANSACTION;

SAVEPOINT customer_1;

DELETE

        FROM bank_CUSTOMER

        WHERE CUSTomer_ID =123001;

SAVEPOINT customer_2;

DELETE

        FROM bank_CUSTOMER

        WHERE CUSTomer_ID =123002;
```

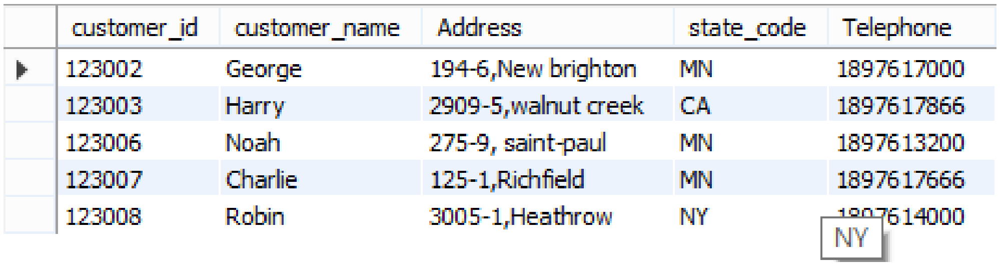
```
SAVEPOINT customer_3;

DELETE

   FROM bank_CUSTOMER

   WHERE Customer_Id=123007;

SAVEPOINT customer_4;

DELETE

   FROM bank_CUSTOMER

   WHERE Customer_Id=123008;

RELEASE SAVEPOINT customer_3;
```

OUTPUT BEFORE THE ROLE BACK:



customer_id	customer_name	Address	state_code	Telephone
123003	Harry	2909-5,walnut creek	CA	1897617866
123006	Noah	275-9, saint-paul	MN	1897613200

```
ROLLBACK TO Customer_2 ;
```



customer_id	customer_name	Address	state_code	Telephone
123002	George	194-6,New brighton	MN	1897617000
123003	Harry	2909-5,walnut creek	CA	1897617866
123006	Noah	275-9, saint-paul	MN	1897613200
123007	Charlie	125-1,Richfield	MN	1897617666
123008	Robin	3005-1,Heathrow	NY	1897614000

After ROLLBACK, we wanted to save the data of customer_3 and beyond this will be safe. But before that every transaction has been deleted. Hence if we want to ROLLBACK that which is deleted, that only gives the output, rather every syntax will give error.

ROLLBACK TO customer_4 ; ***Error***

ROLLBACK TO customer_3 ; ***Error***

ROLLBACK TO Customer_2 ; ***Executed***

We have RELEASE SAVEPOINT customer_3, which means, before customer_3 every (mentioned in SAVEPOINT) data has been deleted. If we want to recollect that, we have to use only ROLLBACK on that SAVEPOINT which was deleted.

ROLLBACK TO Customer_1 ;

ROLLBACK TO Customer_2 ;

BOTH ABOVE DELETED, HENCE SECURE ANY OF THEM WOULD RECOLLECT OUR DATA, WHICH WAS DELETED.

ROLLBACK TO Customer_1 will secure both **Customer_1** as well as **Customer_2**, where **ROLLBACK TO Customer_**2 will secure only **Customer_2**.

UPDATE WITH READ:

SET TRANSACTION READ WRITE;

Update bank_ACCOUNT

SET balance_amount = balance_amount - 2300

WHERE Account_Number = '4000-1956-2001' ;

This will update the current amount in that specific account.

SET TRANSACTION READ ONLY ;

This will allow only reading the transaction, we cannot write anything into the dataset.

SESSION TRANSACTION:

SET SESSION TRANSACTION READ WRITE ;

In read and write mode:

START TRANSACTION;

```
SET SESSION TRANSACTION READ ONLY ;
```

Effective from next transaction

```
UPDATE Bank_ACCOUNT

SET

balance_amount = balance_amount - 2300

WHERE Account_Number = '4000-1956-2001';
```

Into the session only one transaction will be written, remaining will follow another read condition only.

ISOLATION READ WRITE:

This keeps one transaction in waiting state, until it does not complete another transaction at the same account is impermissible. Following kinds of isolation transition:

- REPEATABLE READ
- READ COMMITTED
- SERIALIZABLE

- **REPEATABLE READ**

```
SET SESSION TRANSACTION READ WRITE ;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Such like below, multiple transactions are readable in one commit:

```
SELECT Balance_amount FROM bank_ACCOUNT WHERE Account_Number = '4000-1956-2001';

SELECT Balance_amount FROM bank_ACCOUNT WHERE Account_Number = '4000-1956-3456';
```

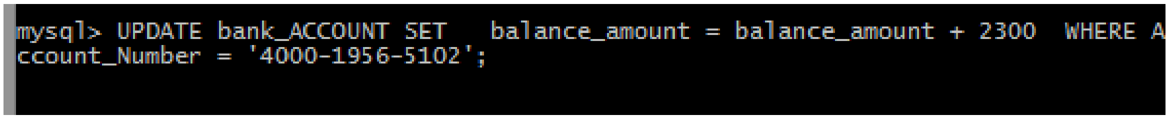
But if writing update is going on different in a platform, then another transaction is not possible at the same instance.

```
UPDATE bank_ACCOUNT SET   balance_amount = balance_amount + 2300  WHERE Account_Number =
'5000-1700-6091' ;
```

IF NOT YET COMMITTED;

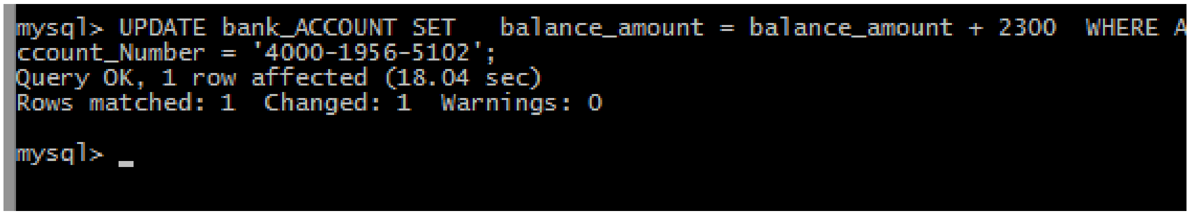
UPDATE bank_ACCOUNT SET   balance_amount = balance_amount + 2300 WHERE Account_Number = '4000-1956-5102' ;

THIS WILL BE IN WAIT UNTILL WE DO NOT COMMIT THE PREVIOUS TRANSACTION.



```
mysql> UPDATE bank_ACCOUNT SET   balance_amount = balance_amount + 2300  WHERE A
ccount_Number = '4000-1956-5102';
```

AS SOON WE COMMIT PREVIOUS TRANSACTION, THIS TRANSACTION WILL BE AUTOMATICALLY DONE.



```
mysql> UPDATE bank_ACCOUNT SET   balance_amount = balance_amount + 2300  WHERE A
ccount_Number = '4000-1956-5102';
Query OK, 1 row affected (18.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> _
```

- **READ COMMITTED**

This is as same as previous one, **REPEATABLE READ**, as it allows only one transaction at once, but surely the table is locked completely, we can insert the data into the database. Here these both operations are allowed in both platforms without ROLLBACK or COMMIT. But in previous **REPEATABLE READ** only one operation is allowed at both platforms. This is the simplest difference in both cases.

UPDATE bank_ACCOUNT SET   balance_amount = balance_amount + 2300 WHERE Account_Number = '5000-1700-6091' ;

INSERT INTO bank_ACCOUNT VALUES (123002, '4000-1956-9999' , 'SAVINGS', 69000, 'ACTIVE' ,'P') ;

One most important thing: Only update with insert is allowed here, because the table is currently locked. We cannot update at a similar time at both platforms.

LOCKING IN DBMS:

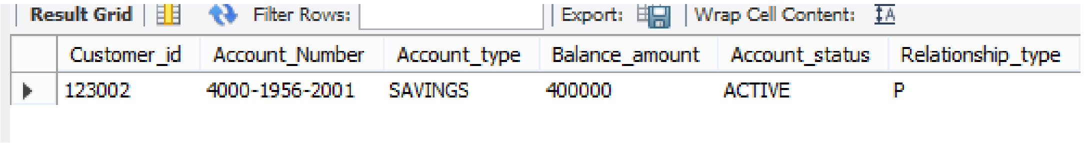
DATABASE → TABLE → PAGE → ROW (These can be locked for DML processes. To maintain data integrity and ACID properties)

Shared /READ lock: This is can be offered to multiple users to read the transactions.

Exclusive /WRITE lock: This is the lock used to write the transactions, and only one user can access this, this is not at all share mode. We cannot offer this to multiple users. If initially shared locked mode is activated to any transaction, by default exclusive lock gets activated if we use UPDATE function.

If any transaction is active with exclusive lock, no other transaction can acquire any kind of the lock, unless and until it does not get COMMITed.

```
START TRANSACTION;

SELECT * FROM bank_account

WHERE Account_Number = '4000-1956-2001'

LOCK IN SHARE MODE;
```

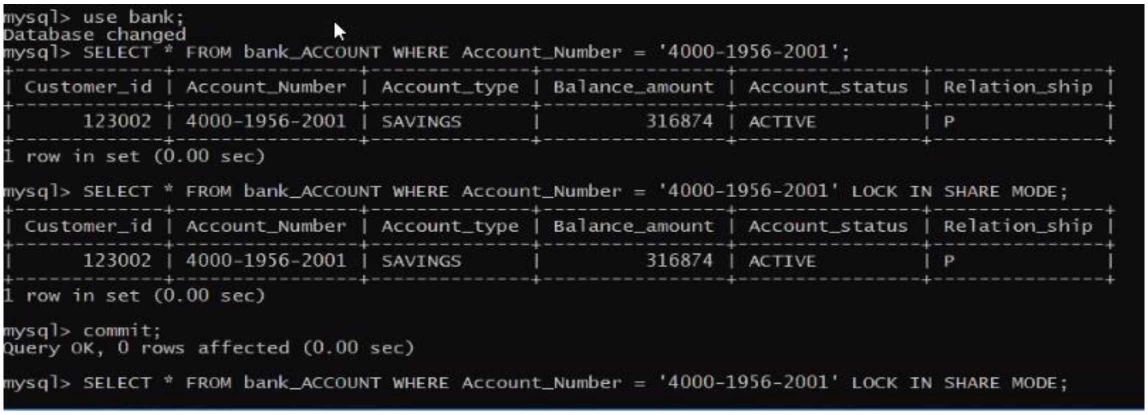


Customer_id	Account_Number	Account_type	Balance_amount	Account_status	Relationship_type
123002	4000-1956-2001	SAVINGS	400000	ACTIVE	P

Similar reading of transaction is possible in another instance. Like multiple users can read the transactions. If the shared lock is promoted to exclusive locked, only one side transaction is allowed. This lock won't be accomplished unless and until the COMMIT is not ordered, still another transaction cannot take up any lock. It has to wait till COMMIT.

```
SET SQL_SAFE_UPDATES=0;

UPDATE bank_account

SET Balance_amount = Balance_amount + 0.04 * Balance_amount

WHERE Account_Number = '4000-1956-2001';   Executed
```

But at another platform, the reading transaction is on wait, until this above write process does not get COMMIT.



```
mysql> use bank;
Database changed
mysql> SELECT * FROM bank_ACCOUNT WHERE Account_Number = '4000-1956-2001';
+-------------+----------------+--------------+----------------+----------------+---------------+
| Customer_id | Account_Number | Account_type | Balance_amount | Account_status | Relation_ship |
+-------------+----------------+--------------+----------------+----------------+---------------+
|      123002 | 4000-1956-2001 | SAVINGS      |         316874 | ACTIVE         | P             |
+-------------+----------------+--------------+----------------+----------------+---------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM bank_ACCOUNT WHERE Account_Number = '4000-1956-2001' LOCK IN SHARE MODE;
+-------------+----------------+--------------+----------------+----------------+---------------+
| Customer_id | Account_Number | Account_type | Balance_amount | Account_status | Relation_ship |
+-------------+----------------+--------------+----------------+----------------+---------------+
|      123002 | 4000-1956-2001 | SAVINGS      |         316874 | ACTIVE         | P             |
+-------------+----------------+--------------+----------------+----------------+---------------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM bank_ACCOUNT WHERE Account_Number = '4000-1956-2001' LOCK IN SHARE MODE;
```

How to apply exclusive lock for update?

START TRANSACTION;

SELECT * FROM bank_customer WHERE customer_id = 123002 FOR UPDATE;

UPDATE bank_customer SET Address = '2999 New brighton' WHERE customer_id = 123002;

COMMIT;

<u>What if you don’t want to permit any source to use the table? Let’s see a question type.</u>

Write a query to make sure that no other mysql session should be able to insert any user ids or passwords

LOCK TABLE id_passwords WRITE;

Here now, no operations will get performed. To reuse the table just give the command to unlock the tables.

If we want some DML on Locked tables:

UPDATE bank_customer SET Address = '2999 New brighton' WHERE customer_id = 123002;

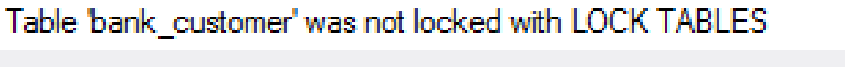


Table 'bank_customer' was not locked with LOCK TABLES

This is the error we are about to face. Now to remove the locks:

UNLOCK TABLES;

You are now in simple condition to apply locks.

DEAD LOCK: It is a kind of stubborn or ego not to work neither let any other to work.

- **Mutual exclusion:** Source of transaction must be active for one process at a time. All transactions should take consistently.
- **No preemption:** If one transaction is in process, do not allow other transaction process until the previous does not get COMMIT.
- **Hold and wait:** Hold a process until it does not get completed, then wait for another process.
- **Circular wait:** If a process P_1 has not yet completed and waiting for another source S_1, at the same time another process P_2 is not yet completed and waiting for the source S_1, then none of the transaction will take place. Hence this is circular wait.

Now by looking at 4 conditions we can pretend weather we are suffering from **DEAD LOCK** situation or not. To avoid this situation we must have to do at least one transaction accomplished, to allow other process to get in for further process. Either you ROLLBACK or COMMIT any one of the process. That is only the solution.

A typical example of DEAD LOCK:

Write a query such that users can perform concurrent DML operations on the same customer_id = 123002 in bank_customer.

One user performs an updates House Address for that customer_id with "2999 New brighton"

Other user performs an update Telephone number with 189891899.

SOLUTION:

SESSION 1:

```
START TRANSACTION;

SELECT * FROM bank_customer WHERE customer_id = 123002 FOR UPDATE;

UPDATE bank_customer SET Address = '2999 New brighton' WHERE customer_id = 123002;

COMMIT;
```

SESSION 2:

```
UPDATE bank_customer SET Telephone = 189891899 WHERE customer_id = 123002;

COMMIT ;
```

If we active both above transactions, DEAD LOCK takes place, which does not let any transaction (process) to gets completed. Hence first COMMIT finishes first transaction then second process gets completed very accurately. If anywhere in the transaction timeout goes up, automatically any one of the transaction gets aborted, because compiler recognizes that there may be the situation of DEAD LOCK.

FOR UPDATE is just the demonstration of exclusive lock.

SOME BASIC QUERIES:

How the avoid duplicate entry into the table when two users try to insert the same record at a time.

Solution:

```
INSERT INTO bank_account

   SELECT * FROM bank_account AS ND WHERE ND.customer_id=123009

   ON DUPLICATE KEY UPDATE customer_id = (SELECT MAX(customer_id)+1 FROM bank_account);

COMMIT;
```