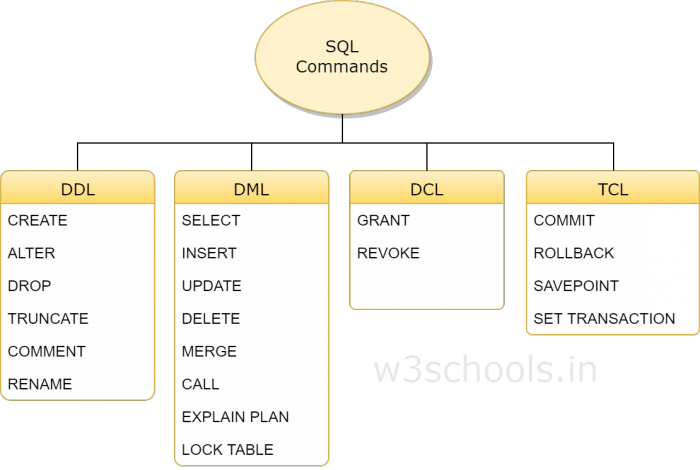
MYSQL DB: queryPractice;

Use queryPractice;



## DDL

DDL is short name of **Data Definition Language,** which deals with database schemas and descriptions, of how the data should reside in the database.

* [CREATE](http://www.w3schools.in/mysql/php-mysql-create/) – to create database and its objects like (table, index, views, store procedure, function and triggers)
* ALTER – alters the structure of the existing database
* DROP – delete objects from the database
* TRUNCATE – remove all records from a table, including all spaces allocated for the records are removed
* COMMENT – add comments to the data dictionary
* RENAME – rename an object

## DML

DML is short name of **Data Manipulation Language** which deals with data manipulation, and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE etc, and it is used to store, modify, retrieve, delete and update data in database.

* [SELECT](http://www.w3schools.in/mysql/php-mysql-select/) – retrieve data from the a database
* [INSERT](http://www.w3schools.in/mysql/php-mysql-insert/) – insert data into a table
* [UPDATE](http://www.w3schools.in/mysql/php-mysql-update/) – updates existing data within a table
* [DELETE](http://www.w3schools.in/mysql/php-mysql-delete/) – Delete all records from a database table
* MERGE – UPSERT operation (insert or update)
* CALL – call a PL/SQL or Java subprogram
* EXPLAIN PLAN – interpretation of the data access path
* LOCK TABLE – concurrency Control

## DCL

DCL is short name of **Data Control Language** which includes commands such as GRANT, and mostly concerned with rights, permissions and other controls of the database system.

* GRANT – allow users access privileges to database
* REVOKE – withdraw users access privileges given by using the GRANT command

## TCL

TCL is short name of Transaction Control Language which deals with transaction within a database.

* COMMIT – commits a Transaction
* ROLLBACK – rollback a transaction in case of any error occurs
* SAVEPOINT – to rollback the transaction making points within groups
* SET TRANSACTION – specify characteristics for the transaction

What does UNION do? What is the difference between UNION and UNION ALL?

Hide answer

UNION merges the contents of two structurally-compatible tables into a single combined table.

The difference between UNION and UNION ALL is that UNION will omit duplicate records where as UNION ALL will include duplicate records.

It is important to note that the performance of UNION ALL will typically be better than UNION, since UNION requires the server to do the additional work of removing any duplicates. So, in cases where is certain that there will not be any duplicates, or where having duplicates is not a problem, use of UNION ALL would be recommended for performance reasons.

Both UNION and UNION ALL concatenate the result of two different SQLs. They differ in the way they handle duplicates.

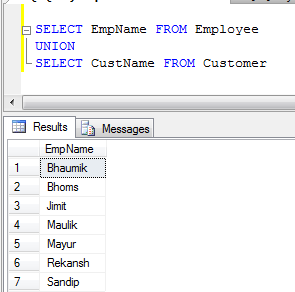
* UNION performs a DISTINCT on the result set, eliminating any duplicate rows.
* UNION ALL does not remove duplicates and it therefore faster than UNION.

**Note:** While using this commands all selected columns need to be of the same data type.

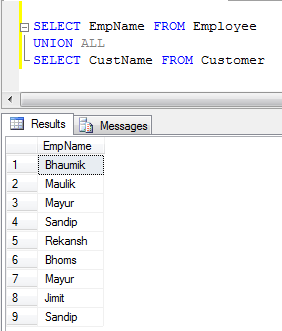
Example: If we have two tables, 1) Employee and 2) Customer

|  |  |
| --- | --- |
| 1. Employee table data: | 1. Customer table data: |
| enter image description here | enter image description here |

1. UNION Example (It removes all duplicate records):



1. UNION ALL Example (It just concatenate records, not eliminate duplicates, so it is faster than UNION):



List and explain the different types of JOIN clauses supported in ANSI-standard SQL.

Hide answer

ANSI-standard SQL specifies five types of JOIN clauses as follows:

|  |  |
| --- | --- |
| Employee | Location |
| |  |  | | --- | --- | | EmpID | EmpName | | 13 | Jason | | 8 | Alex | | 3 | Ram | | 17 | Babu | | 25 | Johnson | | |  |  | | --- | --- | | EmpID | EmpLoc | | 13 | San Jose | | 8 | Los Angeles | | 3 | Pune, India | | 17 | Chennai, India | | 39 | Bangalore, India | |  |  | |
|  |  |

* INNER JOIN (a.k.a. “simple join”): Returns all rows for which there is at least one match in BOTH tables. *This is the default type of join if no specific JOIN type is specified.*
* The difference between an inner join and an outer join is that an inner join will return ***only*** the rows that actually match based on the join predicate. Once again, this is best illustrated via an example. Here’s what the SQL for an inner join will look like:

|  |
| --- |
| select \* from employee inner join location on  employee.empID = location.empID |

* This can also be written as:

|  |
| --- |
| select \* from employee, location  where employee.empID = location.empID |

* Now, here is what the result of running that SQL would look like:

|  |  |  |  |
| --- | --- | --- | --- |
| Employee.EmpID | Employee.EmpName | Location.EmpID | Location.EmpLoc |
| 13 | Jason | 13 | San Jose |
| 8 | Alex | 8 | Los Angeles |
| 3 | Ram | 3 | Pune, India |
| 17 | Babu | 17 | Chennai, India |

* LEFT JOIN (or LEFT OUTER JOIN): Returns all rows from the left table, and the matched rows from the right table; i.e., the results will contain *all* records from the left table, even if the JOIN condition doesn’t find any matching records in the right table. This means that if the ON clause doesn’t match any records in the right table, the JOIN will still return a row in the result for that record in the left table, but with NULL in each column from the right table.
* select \* from employee left outer join location on employee.empID = location.empID;
* select \* from employee left join location on employee.empID = location.empID;

|  |  |  |  |
| --- | --- | --- | --- |
| Employee.EmpID | Employee.EmpName | Location.EmpID | Location.EmpLoc |
| 13 | Jason | 13 | San Jose |
| 8 | Alex | 8 | Los Angeles |
| 3 | Ram | 3 | Pune, India |
| 17 | Babu | 17 | Chennai, India |
| 25 | Johnson | NULL | NULL |

* RIGHT JOIN (or RIGHT OUTER JOIN): Returns all rows from the right table, and the matched rows from the left table. This is the exact opposite of a LEFT JOIN; i.e., the results will contain *all* records from the right table, even if the JOIN condition doesn’t find any matching records in the left table. This means that if the ON clause doesn’t match any records in the left table, the JOIN will still return a row in the result for that record in the right table, but with NULL in each column from the left table.

|  |
| --- |
| select \* from employee right outer join location  on employee.empID = location.empID;  // taking out the "outer", this also works:  select \* from employee right join location  on employee.empID = location.empID; |

* Using the tables presented above, we can show what the result set of a right outer join would look like:

|  |  |  |  |
| --- | --- | --- | --- |
| Employee.EmpID | Employee.EmpName | Location.EmpID | Location.EmpLoc |
| 13 | Jason | 13 | San Jose |
| 8 | Alex | 8 | Los Angeles |
| 3 | Ram | 3 | Pune, India |
| 17 | Babu | 17 | Chennai, India |
| NULL | NULL | 39 | Bangalore, India |

* FULL JOIN (or FULL OUTER JOIN): Returns all rows for which there is a match in EITHER of the tables. Conceptually, a FULL JOIN combines the effect of applying both a LEFT JOIN and a RIGHT JOIN; i.e., its result set is equivalent to performing a UNION of the results of left and right outer queries.
* CROSS JOIN: Returns all records where each row from the first table is combined with each row from the second table (i.e., returns the Cartesian product of the sets of rows from the joined tables). Note that a CROSS JOIN can either be specified using the CROSS JOIN syntax (“explicit join notation”) or (b) listing the tables in the FROM clause separated by commas without using a WHERE clause to supply join criteria (“implicit join notation”).

Write a SQL query that will return the maximum value from the “Numbers” column, without using a SQL aggregate like MAX or MIN.

|  |
| --- |
| Compare |
| |  | | --- | | Numbers | | 30 | | 70 | | -8 | | 90 | |

SELECT Smaller.Numbers, Larger.Numbers

FROM Compare as Larger JOIN Compare AS Smaller

ON Smaller.Numbers < Larger.Numbers

Now, let's use the sample table we created, and we end up with this table after running the query above:

|  |
| --- |
|  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Smaller | Larger | | -8 | 90 | | 30 | 90 | | 70 | 90 | | -8 | 70 | | 30 | 70 | | 70 | 90 | |

[Comment](https://www.toptal.com/sql/interview-questions)

For a table orders having a column defined simply as customer\_id VARCHAR(100), consider the following two query results:

SELECT count(\*) AS total FROM orders;

+-------+

| total |

+-------+

| 100 |

+-------+

SELECT count(\*) AS cust\_123\_total FROM orders WHERE customer\_id = '123';

+----------------+

| cust\_123\_total |

+----------------+

| 15 |

+----------------+

Given the above query results, what will be the result of the query below?

SELECT count(\*) AS cust\_not\_123\_total FROM orders WHERE customer\_id <> '123';

Hide answer

The obvious answer is 85 (i.e, 100 - 15). However, that is not necessarily correct. Specifically, any records with a customer\_id of NULL will *not* be included in *either* count (i.e., they won’t be included in cust\_123\_total, nor will they be included in cust\_not\_123\_total). For example, if exactly one of the 100 customers has a NULL customer\_id, the result of the last query will be:

+--------- ----------+

| cust\_not\_123\_total |

+--------------------+

| 84 |

+--------------------+

Given the following tables:

sql> SELECT \* FROM runners;

+----+--------------+

| id | name |

+----+--------------+

| 1 | John Doe |

| 2 | Jane Doe |

| 3 | Alice Jones |

| 4 | Bobby Louis |

| 5 | Lisa Romero |

+----+--------------+

sql> SELECT \* FROM races;

+----+----------------+-----------+

| id | event | winner\_id |

+----+----------------+-----------+

| 1 | 100 meter dash | 2 |

| 2 | 500 meter dash | 3 |

| 3 | cross-country | 2 |

| 4 | triathalon | NULL |

+----+----------------+-----------+

What will be the result of the query below?

SELECT \* FROM runners WHERE id NOT IN (SELECT winner\_id FROM races)

Explain your answer and also provide an alternative version of this query that will avoid the issue that it exposes.

Hide answer

Surprisingly, given the sample data provided, the result of this query will be an empty set. The reason for this is as follows: If the set being evaluated by the SQL NOT IN condition contains *any* values that are null, then the outer query here will return an empty set, even if there are many runner ids that match winner\_ids in the races table.

Knowing this, a query that avoids this issue would be as follows:

SELECT \* FROM runners WHERE id NOT IN (SELECT winner\_id FROM races WHERE winner\_id IS NOT null)

Note, this is assuming the standard SQL behavior that you get without modifying the default ANSI\_NULLS setting.

Given two tables created and populated as follows:

CREATE TABLE dbo.envelope(id int, user\_id int);

CREATE TABLE dbo.docs(idnum int, pageseq int, doctext varchar(100));

INSERT INTO dbo.envelope VALUES

(1,1),

(2,2),

(3,3);

INSERT INTO dbo.docs(idnum,pageseq) VALUES

(1,5),

(2,6),

(null,0);

What will the result be from the following query:

UPDATE docs SET doctext=pageseq FROM docs INNER JOIN envelope ON envelope.id=docs.idnum

WHERE EXISTS (

SELECT 1 FROM dbo.docs

WHERE id=envelope.id

);

Explain your answer.

Hide answer

The result of the query will be as follows:

idnum pageseq doctext

1 5 5

2 6 6

NULL 0 NULL

The EXISTS clause in the above query is a red herring. It will *always* be true since ID is *not* a member of dbo.docs. As such, it will refer to the envelope table comparing itself to itself!

The idnum value of NULL will not be set since the join of NULL will not return a result when attempting a match with any value of envelope.

What is wrong with this SQL query? Correct it so it executes properly.

SELECT Id, YEAR(BillingDate) AS BillingYear

FROM Invoices

WHERE BillingYear >= 2010;

Hide answer

The expression BillingYear in the WHERE clause is invalid. Even though it is defined as an alias in the SELECT phrase, which appears before the WHERE phrase, the logical processing order of the phrases of the statement is different from the written order. Most programmers are accustomed to code statements being processed generally top-to-bottom or left-to-right, but T-SQL processes phrases in a different order.

The correct query should be:

SELECT Id, YEAR(BillingDate) AS BillingYear

FROM Invoices

WHERE YEAR(BillingDate) >= 2010;

Given these contents of the Customers table:

Id Name ReferredBy

1 John Doe NULL

2 Jane Smith NULL

3 Anne Jenkins 2

4 Eric Branford NULL

5 Pat Richards 1

6 Alice Barnes 2

Here is a query written to return the list of customers not referred by Jane Smith:

SELECT Name FROM Customers WHERE ReferredBy <> 2;

What will be the result of the query? Why? What would be a better way to write it?

Hide answer

Although there are 4 customers not referred by Jane Smith (including Jane Smith herself), the query will only return one: Pat Richards. All the customers who were referred by nobody at all (and therefore have NULL in their ReferredBy column) don’t show up. But certainly those customers weren’t referred by Jane Smith, and certainly NULL is not equal to 2, so why didn’t they show up?

SQL Server uses three-valued logic, which can be troublesome for programmers accustomed to the more satisfying two-valued logic (TRUE or FALSE) most programming languages use. In most languages, if you were presented with two predicates: ReferredBy = 2 and ReferredBy <> 2, you would expect one of them to be true and one of them to be false, given the same value of ReferredBy. In SQL Server, however, if ReferredBy is NULL, neither of them are true and neither of them are false. Anything compared to NULL evaluates to the third value in three-valued logic: UNKNOWN.

The query should be written:

SELECT Name FROM Customers WHERE ReferredBy IS NULL OR ReferredBy <> 2

Watch out for the following, though!

SELECT Name FROM Customers WHERE ReferredBy = NULL OR ReferredBy <> 2

This will return the same faulty set as the original. Why? We already covered that: Anything compared to NULL evaluates to the third value in the three-valued logic: UNKNOWN. That “anything” includes NULL itself! That’s why SQL Server provides the IS NULL and IS NOT NULL operators to specifically check for NULL. Those particular operators will always evaluate to true or false.

Even if a candidate doesn’t have a great amount of experience with SQL Server, diving into the intricacies of three-valued logic in general can give a good indication of whether they have the ability learn it quickly or whether they will struggle with it.

Given two tables created as follows

create table test\_a(id numeric);

create table test\_b(id numeric);

insert into test\_a(id) values

(10),

(20),

(30),

(40),

(50);

insert into test\_b(id) values

(10),

(30),

(50);

Write a query to fetch values in table test\_a that are and not in test\_b **without** using the NOT keyword.

Hide answer

In SQL Server, PostgreSQL, and SQLite, this can be done using the [except](https://en.wikipedia.org/wiki/Set_operations_(SQL)#EXCEPT_operator) keyword as follows:

select \* from test\_a

except

select \* from test\_b;

In Oracle, the [minus](http://www.techonthenet.com/sql/minus.php) keyword is used instead. Note that if there are multiple columns, say ID and Name, the column should be explicitly stated in Oracle queries: Select ID from test\_a minus select ID from test\_b

MySQL does not support the except function, so it is necessary to use not in.

Given a table TBL with a field Nmbr that has rows with the following values:

1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1

Write a query to add 2 where Nmbr is 0 and add 3 where Nmbr is 1.

Hide answer

This can be done as follows:

update TBL set Nmbr = case when Nmbr > 0 then Nmbr+3 else Nmbr+2 end;

Write a SQL query to find the 10th highest employee salary from an Employee table. Explain your answer.

(Note: You may assume that there are at least 10 records in the Employee table.)

Hide answer

This can be done as follows:

SELECT TOP (1) Salary FROM

(

SELECT DISTINCT TOP (10) Salary FROM Employee ORDER BY Salary DESC

) AS Emp ORDER BY Salary

This works as follows:

First, the SELECT DISTINCT TOP (10) Salary FROM Employee ORDER BY Salary DESC query will select the top 10 salaried employees in the table. However, those salaries will be listed in *descending* order. That was necessary for the first query to work, but now picking the top 1 from that list will give you the *highest* salary not the the *10th highest* salary.

Therefore, the second query reorders the 10 records in *ascending* order (which the default sort order) and *then* selects the top record (which will now be the lowest of those 10 salaries).

Not all databases support the TOP keyword. For example, MySQL and PostreSQL use the LIMIT keyword, as follows:

SELECT Salary FROM

(

SELECT DISTINCT Salary FROM Employee ORDER BY Salary DESC LIMIT 10

) AS Emp ORDER BY Salary LIMIT 1;

Or even more concisely, in MySQL this can be:

SELECT DISTINCT Salary FROM Employee ORDER BY Salary DESC LIMIT 9,1;

And in PostgreSQL this can be:

SELECT DISTINCT Salary FROM Employee ORDER BY Salary DESC LIMIT 1 OFFSET 9;

Write a SQL query using UNION ALL (**not** UNION) that uses the WHERE clause to eliminate duplicates. Why might you want to do this?

Hide answer

You can avoid duplicates using UNION ALL and still run much faster than UNION DISTINCT (which is actually same as UNION) by running a query like this:

SELECT \* FROM mytable WHERE a=X UNION ALL SELECT \* FROM mytable WHERE b=Y AND a!=X

The key is the AND a!=X part. This gives you the benefits of the UNION (a.k.a., UNION DISTINCT) command, while avoiding much of its performance hit.

Given the following tables:

SELECT \* FROM users;

user\_id username

1 John Doe

2 Jane Don

3 Alice Jones

4 Lisa Romero

SELECT \* FROM training\_details;

user\_training\_id user\_id training\_id training\_date

1 1 1 "2015-08-02"

2 2 1 "2015-08-03"

3 3 2 "2015-08-02"

4 4 2 "2015-08-04"

5 2 2 "2015-08-03"

6 1 1 "2015-08-02"

7 3 2 "2015-08-04"

8 4 3 "2015-08-03"

9 1 4 "2015-08-03"

10 3 1 "2015-08-02"

11 4 2 "2015-08-04"

12 3 2 "2015-08-02"

13 1 1 "2015-08-02"

14 4 3 "2015-08-03"

Write a query to to get the list of users who took the a training lesson more than once in the same day, grouped by user and training lesson, each ordered from the most recent lesson date to oldest date.

Hide answer

SELECT

u.user\_id,

username,

training\_id,

training\_date,

count( user\_training\_id ) AS count

FROM users u JOIN training\_details t ON t.user\_id = u.user\_id

GROUP BY user\_id,

training\_id,

training\_date

HAVING count( user\_training\_id ) > 1

ORDER BY training\_date DESC;

user\_id username training\_id training\_date count

4 Lisa Romero 2 August, 04 2015 00:00:00 2

4 Lisa Romero 3 August, 03 2015 00:00:00 2

1 John Doe 1 August, 02 2015 00:00:00 3

3 Alice Jones 2 August, 02 2015 00:00:00 2

List and explain each of the ACID properties that collectively guarantee that database transactions are processed reliably.

Hide answer

**ACID (Atomicity, Consistency, Isolation, Durability)** is a set of properties that guarantee that database transactions are processed reliably. They are defined as follows:

* **Atomicity.** Atomicity requires that each transaction be “all or nothing”: if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes.
* **Consistency.** The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.
* **Isolation.** The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method (i.e. if it uses strict - as opposed to relaxed - serializability), the effects of an incomplete transaction might not even be visible to another transaction.
* **Durability.** Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.
* Given a table dbo.users where the column user\_id is a unique numeric identifier, how can you efficiently select the first 100 odd user\_id values from the table?
* (Assume the table contains well over 100 records with odd user\_id values.)
* Hide answer
* SELECT TOP 100 user\_id FROM dbo.users WHERE user\_id % 2 = 1 ORDER BY user\_id
* How can you select all the even number records from a table? All the odd number records?
* Hide answer
* To select all the **even** number records from a table:
* Select \* from table where id % 2 = 0
* To select all the **odd** number records from a table:
* Select \* from table where id % 2 != 0
* What are the NVL and the NVL2 functions in SQL? How do they differ?
* Hide answer
* Both the NVL(exp1, exp2) and NVL2(exp1, exp2, exp3) functions check the value exp1 to see if it is null.
* With the NVL(exp1, exp2) function, if exp1 is *not* null, then the value of exp1 is returned; otherwise, the value of exp2 is returned, but case to the same data type as that of exp1.
* With the NVL2(exp1, exp2, exp3) function, if exp1 is *not* null, then exp2 is returned; otherwise, the value of exp3 is returned.

What is the difference between the WHERE and HAVING clauses?

Hide answer

When GROUP BY is not used, the WHERE and HAVING clauses are essentially equivalent.

However, when GROUP BY**is** used:

* The WHERE clause is used to filter records from a result. The filtering occurs before any groupings are made.
* The HAVING clause is used to filter values from a group (i.e., to check conditions after aggregation into groups has been performed).
* **In SQL, what’s the difference between the having clause and the where clause?**
* The difference between the having and where clause is best illustrated by an example. Suppose we have a table called emp\_bonus as shown below. Note that the table has multiple entries for employees A and B.

|  |
| --- |
| emp\_bonus |
| |  |  | | --- | --- | | **Employee** | **Bonus** | | A | 1000 | | B | 2000 | | A | 500 | | C | 700 | | B | 1250 | |

* If we want to calculate the total bonus that each employee received, then we would write a SQL statement like this:

|  |
| --- |
| select employee, sum(bonus) from emp\_bonus group by employee; |

* **The Group By Clause**
* In the SQL statement above, you can see that we use the "group by" clause with the employee column. What the group by clause does is allow us to find the sum of the bonuses for *each* employee. Using the ‘group by’ in combination with the ‘sum(bonus)’ statement will give us the sum of all the bonuses for employees A, B, and C.
* Running the SQL above would return this:

|  |  |
| --- | --- |
| **Employee** | **Sum(Bonus)** |
| A | 1500 |
| B | 3250 |
| C | 700 |

* Now, suppose we wanted to find the employees who received more than $1,000 in bonuses for the year of 2007. You might think that we could write a query like this:

|  |
| --- |
| BAD SQL:  select employee, sum(bonus) from emp\_bonus  group by employee where sum(bonus) > 1000; |

* **The WHERE clause does not work with aggregates like SUM**
* The SQL above will not work, because the where clause doesn’t work with aggregates – like sum, avg, max, etc.. Instead, what we will need to use is the having clause. The having clause was added to sql just so we could compare aggregates to other values – just how the ‘where’ clause can be used with non-aggregates. Now, the correct sql will look like this:

|  |
| --- |
| GOOD SQL:  select employee, sum(bonus) from emp\_bonus  group by employee having sum(bonus) > 1000; |

* **Difference between having and where clause**
* So we can see that the difference between the having and where clause in sql is that the where clause can *not* be used with aggregates, but the having clause can. One way to think of it is that the having clause is an additional filter to the where clause.
* Suppose we have a Customer table containing the following data:
* CustomerID CustomerName
* 1 Prashant Kaurav
* 2 Ashish Jha
* 3 Ankit Varma
* 4 Vineet Kumar
* 5 Rahul Kumar
* Write a single SQL statement to concatenate all the customer names into the following single semicolon-separated string:
* Prashant Kaurav; Ashish Jha; Ankit Varma; Vineet Kumar; Rahul Kumar
* Hide answer
* SELECT CustomerName+ '; '
* From Customer
* For XML PATH('')
* Given a table Employee having columns empName and empId, what will be the result of the SQL query below?
* select empName from Employee order by 2 desc;
* Hide answer
* “Order by 2” is only valid when there are at least two columns being used in select statement. However, in this query, even though the Employee table has 2 columns, the query is only selecting 1 column name, so “Order by 2” will cause the statement to throw an error while executing the above sql query.
* What is the difference between char and varchar2?
* Hide answer
* When stored in a database, varchar2 uses only the allocated space. E.g. if you have a varchar2(1999) and put 50 bytes in the table, it will use 52 bytes.
* But when stored in a database, char always uses the maximum length and is blank-padded. E.g. if you have char(1999) and put 50 bytes in the table, it will consume 2000 bytes.
* varchar means fixed length character data(size) ie., min size-1 and max-2000
* where as varchar2 means variable length character data ie., min-1 to max-4000
* Table is as follows:

| **ID** | **C1** | **C2** | **C3** |
| --- | --- | --- | --- |
| 1 | Red | Yellow | Blue |
| 2 | NULL | Red | Green |
| 3 | Yellow | NULL | Violet |

* Print the rows which have ‘Yellow’ in one of the columns C1, C2, or C3, but without using OR.
* Hide answer
* SELECT \* FROM table
* WHERE 'Yellow' IN (C1, C2, C3)
* Write a query to insert/update Col2’s values to look exactly opposite to Col1’s values.

| **Col1** | **Col2** |
| --- | --- |
| 1 | 0 |
| 0 | 1 |
| 0 | 1 |
| 0 | 1 |
| 1 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 0 |

* Hide answer
* update table set col2 = case when col1 = 1 then 0 else 1 end

How do you get the last id without the max function?

Hide answer

In MySQL:

select id from table order by id desc limit 1

In SQL Server:

select top 1 id from table order by id desc

What is the difference between IN and EXISTS?

Hide answer

IN:

* Works on List result set
* Doesn’t work on subqueries resulting in Virtual tables with multiple columns
* Compares every value in the result list
* Performance is comparatively SLOW for larger resultset of subquery

EXISTS:

* Works on Virtual tables
* Is used with co-related queries
* Exits comparison when match is found
* Performance is comparatively FAST for larger resultset of subquery
* How can you use a CTE to return the fifth highest (or *Nth* highest) salary from a table?
* Hide answer
* Declare @N int
* set @N = 5;
* WITH CTE AS
* (
* SELECT Name, Salary, EmpID, RN = ROW\_NUMBER()
* OVER (ORDER BY Salary DESC)
* FROM Employee
* )
* SELECT Name, Salary, EmpID
* FROM CTE
* WHERE RN = @N

How do you get the Nth-highest salary from the Employee table without a subquery or CTE?

Hide answer

SELECT salary from Employee order by salary DESC LIMIT 2,1 //worked in MYSQL

Given the following table named A:

x

------

2

-2

4

-4

-3

0

2

Write a single query to calculate the sum of all positive values of x and the sum of all negative values of x.

Hide answer

select sum(case when x>0 then x else 0 end)sum\_pos,sum(case when x<0 then x else 0 end)sum\_neg from a;

Given the table mass\_table:

| **Weight** |
| --- |
| 5.67 |
| 34.567 |
| 365.253 |
| 34 |

Write a query that produces the output:

| **Weight** | **kg** | **gms** |
| --- | --- | --- |
| 5.67 | 5 | 67 |
| 34.567 | 34 | 567 |
| 365.253 | 365 | 253 |
| 34 | 34 | 0 |

Hide answer

select weight, trunc(weight) as kg, nvl(substr(weight - trunc(weight), 2), 0) as gms

from mass\_table;

Consider the Employee table below.

| **Emp\_Id** | **Emp\_name** | **Salary** | **Manager\_Id** |
| --- | --- | --- | --- |
| 10 | Anil | 50000 | 18 |
| 11 | Vikas | 75000 | 16 |
| 12 | Nisha | 40000 | 18 |
| 13 | Nidhi | 60000 | 17 |
| 14 | Priya | 80000 | 18 |
| 15 | Mohit | 45000 | 18 |
| 16 | Rajesh | 90000 | – |
| 17 | Raman | 55000 | 16 |
| 18 | Santosh | 65000 | 17 |

Write a query to generate below output:

| **Manager\_Id** | **Manager** | **Average\_Salary\_Under\_Manager** |
| --- | --- | --- |
| 16 | Rajesh | 65000 |
| 17 | Raman | 62500 |
| 18 | Santosh | 53750 |

Hide answer

select b.emp\_id as "Manager\_Id",

b.emp\_name as "Manager",

avg(a.salary) as "Average\_Salary\_Under\_Manager"

from Employee a,

Employee b

where a.manager\_id = b.emp\_id

group by b.emp\_id, b.emp\_name

order by b.emp\_id;

How do you copy data from one table to another table ?

Hide answer

INSERT INTO table2 (column1, column2, column3, ...)

SELECT column1, column2, column3, ...

FROM table1

WHERE condition;

Find the SQL statement below that is equal to the following: SELECT name FROM customer WHERE state = 'VA';

1. SELECT name IN customer WHERE state IN ('VA');
2. SELECT name IN customer WHERE state = 'VA';
3. SELECT name IN customer WHERE state = 'V';
4. SELECT name FROM customer WHERE state IN ('VA');

Hide answer

1. SELECT name FROM customer WHERE state IN ('VA');

Comment

[**4 Ways to find Nth highest salary in SQL - Oracle, MSSQL and MySQL**](http://javarevisited.blogspot.in/2016/01/4-ways-to-find-nth-highest-salary-in.html)

One of the most common SQL interview questions is to find the Nth highest salary of employee, where N could be 2, 3, 4 or anything e.g. find the [second highest salary in SQL](http://java67.blogspot.com/2015/01/second-highest-salary-in-mysql-and-sql-server.html). Sometimes this question is also twisted as to find the *nth minimum salary in SQL*. Since many Programmers only know the easy way to solve this problem e.g. by using SQL IN clause, which doesn't scale well, they struggle to write the SQL query when Interviewer keep asking about 4th highest, 5th highest and so on. In order to solve this problem effectively, you need to know about some key concepts like correlated subquery, window functions like **ROW\_NUMER()**, **RANK()** and **DENSE\_RANK()**etc. Once you know the generic logic to solve this problem, you can tackle all those variations by yourself.  
  
  
In this article, I'll show you 4 ways to solve this problem e.g. by using the correlated subquery, using **ROW\_NUMBER()**, using TOP in SQL SERVER and by using LIMIT keyword in MySQL. Btw, if you are new to SQL and just learning these query from interviews sake then I suggest you to first read a good book on SQL e.g. [Head First SQL](http://aax-us-east.amazon-adsystem.com/x/c/QgEBAh4hrz4BWoJ6qu3j0REAAAFfJdUQAgEAAAFKARAXLRo/https:/assoc-redirect.amazon.com/g/r/http:/www.amazon.com/Head-First-SQL-Brain-Learners/dp/0596526849/ref=as_at?creativeASIN=0596526849&linkCode=w61&imprToken=6dHFpjmXX4Uve5B86ljFrQ&slotNum=0&tag=javamysqlanta-20). It will help you to build your fundamentals.

**Sample table and data for Nth Highest Salary Problem**

Before solving this problem we need some sample data to visualize the problem better, let's create employee table with some data.  
  
Use below query to create table and build schema:

-- creating Employee table in Oracle

CREATE TABLE Employee (name varchar(10), salary int);

-- inserting sample data into Employee table

INSERT INTO Employee VALUES ('Rick', 3000);

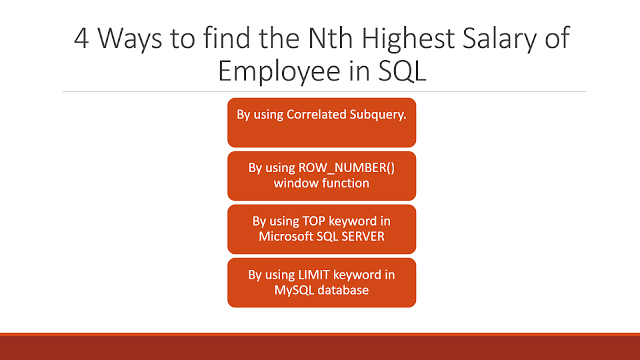
INSERT INTO Employee VALUES ('John', 4000);

INSERT INTO Employee VALUES ('Shane', 3000);

INSERT INTO Employee VALUES ('Peter', 5000);

INSERT INTO Employee VALUES ('Jackob', 7000);

**Nth highest salary using correlated subquery**

[](http://aax-us-east.amazon-adsystem.com/x/c/QgEBAh4hrz4BWoJ6qu3j0REAAAFfJdUQAgEAAAFKARAXLRo/https:/assoc-redirect.amazon.com/g/r/http:/www.amazon.com/Head-First-SQL-Brain-Learners/dp/0596526849/ref=as_at?creativeASIN=0596526849&linkCode=w61&imprToken=6dHFpjmXX4Uve5B86ljFrQ&slotNum=1&tag=javamysqlanta-20)

One of the most common ways to solve this problem of finding the Nth maximum salary from Employee table is by using the [correlated subquery](http://javarevisited.blogspot.com/2012/07/subquery-example-in-sql-correlated-vs.html). This is a special type of subquery where the subquery depends upon the main query and execute for every row returned by the main query.  It's slow but it can solve problems which are difficult to solve otherwise. Let's see the *SQL query to find the Nth highest salary using the Correlated subquery*.  
  
**SQL Query:**

SELECT name, salary

FROM #Employee e1

WHERE N-1 = (SELECT COUNT(DISTINCT salary) FROM #Employee e2

WHERE e2.salary > e1.salary)

for the 2nd maximum you can replace N with 2, and for 3rd maximum replace N with 3, here is the output:  
  
**2nd highest salary:**

SELECT name, salary

FROM #Employee e1

WHERE N-1 = (SELECT COUNT(DISTINCT salary) FROM #Employee e2

WHERE e2.salary > e1.salary)SELECT name, salary

FROM #Employee e1

WHERE 2-1 = (SELECT COUNT(DISTINCT salary) FROM #Employee e2

WHERE e2.salary > e1.salary)

Result:

name salary

Peter 5000

**3rd highest salary:**

SELECT name, salary

FROM #Employee e1

WHERE 3-1 = (SELECT COUNT(DISTINCT salary) FROM #Employee e2

WHERE e2.salary > e1.salary)

Result:

name salary

John 4000

**Explanation :**  
The **distinct**keyword is there to deal with duplicate salaries in the table. In order to find the Nth highest salary, we are only considering unique salaries. Highest salary means no salary is higher than it, Second highest means only one salary is higher than it, 3rd highest means two salaries are higher than it, similarly Nth highest salary means N-1 salaries are higher than it.  
  
**Pros :**  
1) The generic solution works in all database including Oracle, MySQL, SQL SERVER and PostgreSQL.  
  
**Cons :**  
1) Slow, because the inner query will run for every row processed by the outer query.  
  
See [SQL Puzzles and Answers](http://aax-us-east.amazon-adsystem.com/x/c/QgEBAh4hrz4BWoJ6qu3j0REAAAFfJdUQAgEAAAFKARAXLRo/https:/assoc-redirect.amazon.com/g/r/http:/www.amazon.com/Puzzles-Answers-Edition-Kaufmann-Management/dp/0123735963/ref=as_at?creativeASIN=0123735963&linkCode=w61&imprToken=6dHFpjmXX4Uve5B86ljFrQ&slotNum=2&tag=javamysqlanta-20" \t "_blank) book for more of such SQL queries for practicing and improving your SQL query skill.

**Nth highest salary BY USING SUB QUERY**

**Below example for 2nd higest**

SELECT MAX(Salary) FROM Employee

WHERE Salary NOT IN (SELECT MAX(Salary) FROM Employee )

**Nth highest salary in SQL SERVER using TOP keyword**

You can use the TOP keyword to find the Nth highest salary in SQL SERVER. This is also faster than the previous solution because here we are calculating Nth maximum salary without a subquery.

SELECT TOP 1 salary

FROM (

SELECT DISTINCT TOP N salary

FROM #Employee

ORDER BY salary DESC

) AS temp

ORDER BY salary

**Explanation:**  
By default ORDER BY clause print rows in ascending order, since we need the highest salary at the top, we have used ORDER BY DESC, which will display salaries in descending order. Again DISTINCT is used to remove duplicates. The outer query will then pick the top most salary, which would be your Nth highest salary.  
  
3rd highest salary in SQL SERVER

SELECT TOP 1 salary

FROM (

SELECT DISTINCT TOP 3 salary

FROM #Employee

ORDER BY salary DESC

) AS temp

ORDER BY salary

Result:

salary

4000

Here is [another example](http://javarevisited.blogspot.com/2012/12/how-to-find-second-highest-or-maximum-salary-sql.html)where we have used the TOP keyword to find the second highest salary in Microsoft SQL SERVER 2008.

**Nth maximum salary in MySQL using LIMIT keyword (IN SINGLE TABLE not Associate with any Foreign key)**

Similar to TOP, MySQL also supports a LIMIT keyword, which provides pagination capability. You can find the nth highest salary in MySQL without using subquery as shown below:

mysql> select \* from DATA;

+-----+--------+

| ID | SALARY |

+-----+--------+

| 100 | 9000 |

| 101 | 6000 |

| 102 | 18000 |

| 103 | 8000 |

| 104 | 4000 |

//3rd Highest = N-1=2

mysql> select ID , salary from DATA order by salary desc Limit 2,1;

+-----+--------+

| ID | salary |

+-----+--------+

| 103 | 8000 |

+-----+--------+

1 row in set (0.00 sec)

mysql> select ID , salary from DATA order by salary desc Limit 0,1;

+-----+--------+

| ID | salary |

+-----+--------+

| 102 | 18000 |

+-----+--------+

1 row in set (0.00 sec)

mysql> select ID , salary from DATA order by salary desc Limit 1,1;

+-----+--------+

| ID | salary |

+-----+--------+

| 100 | 9000 |

+-----+--------+

SELECT salary FROM Employee ORDER BY salary DESC LIMIT N-1, 1

2nd highest salary in MySQL without subquery:

SELECT salary FROM Employee ORDER BY salary DESC LIMIT 1,1

salary

5000

3rd highest salary in MySQL using LIMIT clause:

SELECT salary FROM Employee ORDER BY salary DESC LIMIT 2,1

salary

4000

Nth highest salary in MySQL using LIMIT clause:

SELECT salary FROM Employee ORDER BY Salary DESC LIMIT n-1,1

**Explanation :**  
The benefit of this approach is that it's faster than correlated query approach but its vendor dependent. This solution will only work in MySQL database.

**Nth highest salary in Oracle using ROW\_NUMBER() function**

SELECT \* FROM (

SELECT e.\*,

ROW\_NUMBER() OVER (ORDER BY salary DESC) rn

FROM Employee e

)

WHERE rn = N; /\*N is the nth highest salary\*/

Here is the [2nd highest salary in Oracle using ROW\_NUMBER()](http://javarevisited.blogspot.com/2015/11/2nd-highest-salary-in-oracle-using-rownumber-rank-example.html) window function:

SELECT \* FROM (

SELECT e.\*,

ROW\_NUMBER() OVER (ORDER BY salary DESC) rn

FROM Employee e

)

WHERE rn = 2;

Output

NAME SALARY RN

Peter 5000 2

and here is 3rd highest salary in Oracle:

SELECT \* FROM (

SELECT e.\*,

ROW\_NUMBER() OVER (ORDER BY salary DESC) rn

FROM Employee e

)

WHERE rn = 3;

By the above code has a problem. It is not handling duplicate salaries properly. For example, in our table we have two employees with salary 3000, that's our 4th highest salary, but above code will print the same salary, albeit different employee for both 4th and 5th maximum as shown below:

SELECT \* FROM (

SELECT e.\*,

ROW\_NUMBER() OVER (ORDER BY salary DESC) rn

FROM Employee e

)

WHERE rn = 5;

Result:

NAME SALARY RN

Shane 3000 5

In oracle, you can also use SQL statements to build schema and run sample SQL.  
  
You can also do the same thing by using **RANK()** window function in Oracle, but that's for another day. This is more than enough to answer the SQL interview question, the print nth highest salary of an employee in the Oracle.  
  
  
That's all about **how to find the nth highest salary in SQL**. The easiest way to find nth maximum/minimum salary is by using the correlated subquery, but it's not the fastest way. Better ways are database dependent e.g. you cause TOP keyword in SQL SERVER, LIMIT keyword in MySQL and ROW\_NUMBER() window function in Oracle to calculate the nth highest salary. The normal subquery way is good for the second maximum but after that, it become really nested and cluttered.

Persistent Interview Question:

//Why can we use aggregate function in where clause ---need to check

## WHERE vs HAVING Clause Example in SQL

## **MySQL aggregate functions**including AVG,COUNT, SUM, MAX and MIN.

[Difference betwen WHERE and HAVING clause in SQL - Interview Question](http://1.bp.blogspot.com/-ALTiC41IG4I/TZ_n2bwh6iI/AAAAAAAAAFE/Jy4cmcZqUG4/s1600/mysql.gif)In this example of WHERE and HAVING clause, we have two tables Employee and Department. Employee contains details of employees e.g. id, name, age, salary and department id, while Department contains id and department name. In order to show, which employee works for which department we need to [join two tables](http://javarevisited.blogspot.sg/2012/11/how-to-join-three-tables-in-sql-query-mysql-sqlserver.html) on DEPT\_ID to get the the department name. Our requirement is to find how many employees are working in each department and average salary of department. In order to use WHERE clause, we will only include employees who are earning  more than 5000. Before executing our query which contains WHERE, HAVING, and GROUP BY clause, let see data from Employee and Department table:

**SELECT** \* **FROM** Employee;

mysql> create table Employee (EMP\_ID int, EMP\_NAME varchar(30),EMP\_AGE int, EMP\_SALARY varchar(10), DEPT\_ID int , CONSTRAINT fk\_department FOREIGN KEY ( DEPT\_ID ) REFEREN

DELETE CASCADE ON UPDATE RESTRICT);

mysql> insert into Employee (EMP\_ID,EMP\_NAME,EMP\_AGE,EMP\_SALARY,DEPT\_ID) values(1,"Virat",23,10000,1);

Query OK, 1 row affected (0.02 sec)

mysql> insert into Employee (EMP\_ID,EMP\_NAME,EMP\_AGE,EMP\_SALARY,DEPT\_ID) values(2,"Rohit",24,7000,2);

Query OK, 1 row affected (0.00 sec)

mysql> insert into Employee (EMP\_ID,EMP\_NAME,EMP\_AGE,EMP\_SALARY,DEPT\_ID) values(3,"Suresh",25,8000,3);

Query OK, 1 row affected (0.06 sec)

mysql> insert into Employee (EMP\_ID,EMP\_NAME,EMP\_AGE,EMP\_SALARY,DEPT\_ID) values(4,"Shikhar",27,6000,1);

Query OK, 1 row affected (0.05 sec)

mysql> insert into Employee (EMP\_ID,EMP\_NAME,EMP\_AGE,EMP\_SALARY,DEPT\_ID) values(5,"Vijay",28,5000,2);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_AGE** | **EMP\_SALARY** | **DEPT\_ID** |
| 1 | Virat | 23 | 10000 | 1 |
| 2 | Rohit | 24 | 7000 | 2 |
| 3 | Suresh | 25 | 8000 | 3 |
| 4 | Shikhar | 27 | 6000 | 1 |
| 5 | Vijay | 28 | 5000 | 2 |

**SELECT** \* **FROM** Department;

|  |  |
| --- | --- |
| **DEPT\_ID** | **DEPT\_NAME** |
| 1 | Accounting |
| 2 | Marketing |
| 3 | Sales |

create table Department (DEPT\_ID int NOT NULL AUTO\_INCREMENT, DEPT\_NAME varchar(30), PRIMARY KEY (DEPT\_ID));

insert into Department (DEPT\_ID , DEPT\_NAME ) values (1,"Accounting");

insert into Department (DEPT\_ID , DEPT\_NAME ) values (2,"Marketing");

insert into Department (DEPT\_ID , DEPT\_NAME ) values (3,"Sales");

**SELECT** d.DEPT\_NAME, **count**(e.EMP\_NAME) **as** NUM\_EMPLOYEE, **avg**(e.EMP\_SALARY) **as** AVG\_SALARY **FROM** Employee e,Department d **WHERE** e.DEPT\_ID=d.DEPT\_ID **AND** EMP\_SALARY > **5000** **GROUP** **BY** d.DEPT\_NAME;

|  |  |  |
| --- | --- | --- |
| **DEPT\_NAME** | **NUM\_EMPLOYEE** | **AVG\_SALARY** |
| Accounting | 1 | 8000 |
| Marketing | 1 | 6000 |
| Sales | 2 | 8000 |

From the number of employee (NUM\_EMPLOYEE) column you can see that only Vijay who work for Marketing department is not included in result set because his earning 5000. This example shows that, condition in WHERE clause is used to filter rows before you aggregate them and then HAVING clause comes in picture for final filtering, which is clear from following query, now Marketing department is excluded because it doesn't pass condition in HAVING clause i..e AVG\_SALARY > 7000

**SELECT** d.DEPT\_NAME, **count**(e.EMP\_NAME) **as** NUM\_EMPLOYEE, **avg**(e.EMP\_SALARY) **as** AVG\_SALARY **FROM** Employee e,Department d **WHERE** e.DEPT\_ID=d.DEPT\_ID **AND** EMP\_SALARY > **5000** **GROUP** **BY** d.DEPT\_NAME **HAVING** AVG\_SALARY > 7000;

|  |  |  |
| --- | --- | --- |
| **DEPT\_NAME** | **NUM\_EMPLOYEE** | **AVG\_SALARY** |
| Accounting | 1 | 8000 |
| Sales | 2 | 8000 |

mysql> SELECT d.DEPT\_NAME, e.EMP\_NAME, count(e.EMP\_NAME) as NUM\_EMPLOYEE FROM Employee e,Department d WHERE e.DEPT\_ID=d.DEPT\_ID GROUP BY d.DEPT\_NAME HAVING NUM\_EMPLOYEE> 1;

+------------+----------+--------------+

| DEPT\_NAME | EMP\_NAME | NUM\_EMPLOYEE |

+------------+----------+--------------+

| Accounting | Virat | 2 |

| Marketing | Rohit | 2 |

+------------+----------+--------------+

## Difference between WHERE and HAVING in SQL

Apart from this key difference we have seen in this article, here are few more differences between WHERE and HAVING clause, which is worth remembering and can be used to compare both of them :

1) Apart from SELECT queries, you can use WHERE clause with UPDATE and DELETE clause but HAVING clause can only be used with SELECT query. For example following query, which involve WHERE clause will work but other which uses HAVING clause will not work :

**update** DEPARTMENT **set** DEPT\_NAME="NewSales" **WHERE** DEPT\_ID=**1** ;  // works fine

**update** DEPARTMENT **set** DEPT\_NAME="NewSales" **HAVING** DEPT\_ID=**1** ; // error

Incorrect syntax near the keyword 'HAVING'.: **update** DEPARTMENT **set** DEPT\_NAME='NewSales' **HAVING** DEPT\_ID=**1**

2) WHERE clause is used for filtering rows and it applies on each and every row, while HAVING clause is used to filter groups in SQL.

3) One syntax level **difference between WHERE and HAVING clause** is that, former is used before GROUP BY clause, while later is used after GROUP BY clause.

4) When WHERE and HAVING clause are used together in a SELECT query with aggregate function,  WHERE clause is applied first on individual rows and only rows which pass the condition is included for creating groups. Once group is created, HAVING clause is used to filter groups based upon condition specified.

Read more: <http://javarevisited.blogspot.com/2013/08/difference-between-where-vs-having-clause-SQL-databse-group-by-comparision.html#ixzz51ZYqXec3>

//Persistent Interview question

mysql> select \* from address;

+--------+---------------+---------+-------------+

| emp\_id | address\_line1 | zipcode | city |

+--------+---------------+---------+-------------+

| 1 | Albany Dr | 95129 | San Jose |

| 2 | Arques Ave | 95051 | Santa Clara |

| 3 | BTM 1st Stage | 560100 | Bangalore |

| 4 | City Centre | 100100 | New Delhi |

+--------+---------------+---------+-------------+

4 rows in set (0.00 sec)

mysql> select \* from employess;

ERROR 1146 (42S02): Table 'testdb.employess' doesn't exist

mysql> select \* from employee;

+--------+----------+------------+

| emp\_id | emp\_name | emp\_salary |

+--------+----------+------------+

| 1 | Pankaj | 100 |

| 2 | David | 200 |

| 3 | Lisa | 300 |

| 4 | Jack | 400 |

+--------+----------+------------+

mysql> select \* from address,employee;

+--------+---------------+---------+-------------+--------+----------+------------+

| emp\_id | address\_line1 | zipcode | city | emp\_id | emp\_name | emp\_salary |

+--------+---------------+---------+-------------+--------+----------+------------+

| 1 | Albany Dr | 95129 | San Jose | 1 | Pankaj | 100 |

| 2 | Arques Ave | 95051 | Santa Clara | 1 | Pankaj | 100 |

| 3 | BTM 1st Stage | 560100 | Bangalore | 1 | Pankaj | 100 |

| 4 | City Centre | 100100 | New Delhi | 1 | Pankaj | 100 |

| 1 | Albany Dr | 95129 | San Jose | 2 | David | 200 |

| 2 | Arques Ave | 95051 | Santa Clara | 2 | David | 200 |

| 3 | BTM 1st Stage | 560100 | Bangalore | 2 | David | 200 |

| 4 | City Centre | 100100 | New Delhi | 2 | David | 200 |

| 1 | Albany Dr | 95129 | San Jose | 3 | Lisa | 300 |

| 2 | Arques Ave | 95051 | Santa Clara | 3 | Lisa | 300 |

| 3 | BTM 1st Stage | 560100 | Bangalore | 3 | Lisa | 300 |

| 4 | City Centre | 100100 | New Delhi | 3 | Lisa | 300 |

| 1 | Albany Dr | 95129 | San Jose | 4 | Jack | 400 |

| 2 | Arques Ave | 95051 | Santa Clara | 4 | Jack | 400 |

| 3 | BTM 1st Stage | 560100 | Bangalore | 4 | Jack | 400 |

| 4 | City Centre | 100100 | New Delhi | 4 | Jack | 400 |

+--------+---------------+---------+-------------+--------+----------+------------+

16 rows in set (0.00 sec)

==================STORE PROCEDURE CALL============================================

|  |
| --- |
| **public** **void** invokeProcedureAndSendUpdates(**final** String subscriberCodes) {  **try** {  getJpaTemplate().execute(**new** JpaCallback() {  **public** Object doInJpa(EntityManager entity) **throws** javax.persistence.PersistenceException {  Connection connection = **null**;  **try** {  connection = getJpaTemplate().getJpaDialect().getJdbcConnection(entity, **false**).getConnection();  ***LOGGER***.debug(" DB Connection is " + connection);  // if (connection != null) {  connection.setAutoCommit(**false**);  **final** List<String> subscriberCodeList = getSubscribersList(subscriberCodes);  **if** (subscriberCodeList != **null** && !subscriberCodeList.isEmpty()) {  **for** (**final** String subscriberCode : subscriberCodeList) {  invokeProceduresForSubscriber(connection, subscriberCode);  } |

**private** **void** invokeProceduresForSubscriber(**final** Connection connection, **final** String subscriberCode)

**throws** SQLException {

CallableStatement cStmtActToInact = **null**;

**try** {

cStmtActToInact = connection

.prepareCall("{? = call PKG\_PURGE\_UPDATE\_PROFILE.PROFILE\_UPD\_ACTIVE\_TO\_INACTIVE(?)}");

executeProcedureAndSendUpdate(cStmtActToInact, subscriberCode);

}

**finally** {

closeStatement(cStmtActToInact);

}

}