SQL query return data from multiple tables

Asked 12 years, 3 months ago Modified 6 years, 8 months ago

Viewed 927k times

467

I would like to know the following:

- how to get data from multiple tables in my database?
- what types of methods are there to do this?
- what are joins and unions and how are they different from one another?
- When should I use each one compared to the others?

I am planning to use this in my (for example - PHP) application, but don't want to run multiple queries against the database, what options do I have to get data from multiple tables in a single query?

Note: I am writing this as I would like to be able to link to a well written guide on the numerous questions that I constantly come across in the PHP queue, so I can link to this for further detail when I post an answer.

The answers cover off the following:

1. Part 1 - Joins and Unions

mysql sql select

Share

Improve this question

Follow

edited Apr 12, 2018 at 20:51

community wiki
12 revs, 5 users 64%
Fluffeh

6 Answers

Sorted by: Highest score (default)

505

Part 1 - Joins and Unions

This answer covers:

1. Part 1
    - Joining two or more tables using an inner join (See the wikipedia entry for additional info)
    - How to use a union query
    - Left and Right Outer Joins (this stackOverflow answer is excellent to describe types of joins)

    - Intersect queries (and how to reproduce them if your database doesn't support them) - this is a function of SQL-Server (see info) and part of the reason I wrote this whole thing in the first place.

2. Part 2
    - Subqueries - what they are, where they can be used and what to watch out for
    - Cartesian joins AKA - Oh, the misery!

There are a number of ways to retrieve data from multiple tables in a database. In this answer, I will be using ANSI-92 join syntax. This may be different to a number of other tutorials out there which use the older ANSI-89 syntax (and if you are used to 89, may seem much less intuitive - but all I can say is to try it) as it is *much* easier to understand when the queries start getting more complex. Why use it? Is there a performance gain? The short answer is no, but it *is* easier to read once you get used to it. It is easier to read queries written by other folks using this syntax.

I am also going to use the concept of a small caryard which has a database to keep track of what cars it has available. The owner has hired you as his IT Computer guy and expects you to be able to drop him the data that he asks for at the drop of a hat.

I have made a number of lookup tables that will be used by the final table. This will give us a reasonable model to work from. To start off, I will be running my queries against an example database that has the following structure. I will try to think of common mistakes that are made when starting out and explain what goes wrong with them - as well as of course showing how to correct them.

The first table is simply a color listing so that we know what colors we have in the car yard.

```
mysql> create table colors(id int(3) not null auto_inc
    -> color varchar(15), paint varchar(10));
Query OK, 0 rows affected (0.01 sec)

mysql> show columns from colors;
+-------+-------------+------+-----+---------+--------
| Field | Type        | Null | Key | Default | Extra
+-------+-------------+------+-----+---------+--------
| id    | int(3)      | NO   | PRI | NULL    | auto_in
| color | varchar(15) | YES  |     | NULL    |
| paint | varchar(10) | YES  |     | NULL    |
+-------+-------------+------+-----+---------+--------
3 rows in set (0.01 sec)

mysql> insert into colors (color, paint) values ('Red'
    -> ('Green', 'Gloss'), ('Blue', 'Metallic'),
    -> ('White' 'Gloss'), ('Black' 'Gloss');
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from colors;
+----+-------+----------+
| id | color | paint    |
+----+-------+----------+
|  1 | Red   | Metallic |
|  2 | Green | Gloss    |
|  3 | Blue  | Metallic |
|  4 | White | Gloss    |
|  5 | Black | Gloss    |
+----+-------+----------+
5 rows in set (0.00 sec)
```

The brands table identifies the different brands of the cars out caryard could possibly sell.

```
mysql> create table brands (id int(3) not null auto_in
    -> brand varchar(15));
Query OK, 0 rows affected (0.01 sec)

mysql> show columns from brands;
+-------+-------------+------+-----+---------+--------
| Field | Type        | Null | Key | Default | Extra
+-------+-------------+------+-----+---------+--------
| id    | int(3)      | NO   | PRI | NULL    | auto_in
| brand | varchar(15) | YES  |     | NULL    |
+-------+-------------+------+-----+---------+--------
2 rows in set (0.01 sec)

mysql> insert into brands (brand) values ('Ford'), ('T
    -> ('Nissan'), ('Smart'), ('BMW');
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from brands;
+----+--------+
| id | brand  |
+----+--------+
|  1 | Ford   |
|  2 | Toyota |
|  3 | Nissan |
|  4 | Smart  |
|  5 | BMW    |
+----+--------+
5 rows in set (0.00 sec)
```

The model table will cover off different types of cars, it is going to be simpler for this to use different car types rather than actual car models.

```
mysql> create table models (id int(3) not null auto_in
    -> model varchar(15));
Query OK, 0 rows affected (0.01 sec)

mysql> show columns from models;
+-------+-------------+------+-----+---------+--------
| Field | Type        | Null | Key | Default | Extra
+-------+-------------+------+-----+---------+--------
| id    | int(3)      | NO   | PRI | NULL    | auto_in
| model | varchar(15) | YES  |     | NULL    |
+-------+-------------+------+-----+---------+--------
2 rows in set (0.00 sec)

mysql> insert into models (model) values ('Sports'), (
('Luxury');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> select * from models;
+----+--------+
| id | model  |
+----+--------+
|  1 | Sports |
|  2 | Sedan  |
|  3 | 4WD    |
|  4 | Luxury |
+----+--------+
4 rows in set (0.00 sec)
```

And finally, to tie up all these other tables, the table that ties everything together. The ID field is actually the unique lot number used to identify cars.

```
mysql> create table cars (id int(3) not null auto_incr
    -> color int(3), brand int(3), model int(3));
Query OK, 0 rows affected (0.01 sec)

mysql> show columns from cars;
+-------+--------+------+-----+---------+------------
| Field | Type   | Null | Key | Default | Extra
```

```
+-------+--------+------+-----+---------+-------------
| id    | int(3) | NO   | PRI | NULL    | auto_increme
| color | int(3) | YES  |     | NULL    |
| brand | int(3) | YES  |     | NULL    |
| model | int(3) | YES  |     | NULL    |
+-------+--------+------+-----+---------+-------------
4 rows in set (0.00 sec)

mysql> insert into cars (color, brand, model) values (
    -> (4,4,2), (2,2,3), (3,5,4), (4,1,3), (2,2,1), (5
Query OK, 10 rows affected (0.00 sec)
Records: 10  Duplicates: 0  Warnings: 0

mysql> select * from cars;
+----+-------+-------+-------+
| id | color | brand | model |
+----+-------+-------+-------+
|  1 |     1 |     2 |     1 |
|  2 |     3 |     1 |     2 |
|  3 |     5 |     3 |     1 |
|  4 |     4 |     4 |     2 |
|  5 |     2 |     2 |     3 |
|  6 |     3 |     5 |     4 |
|  7 |     4 |     1 |     3 |
|  8 |     2 |     2 |     1 |
|  9 |     5 |     2 |     3 |
| 10 |     4 |     5 |     1 |
+----+-------+-------+-------+
10 rows in set (0.00 sec)
```

This will give us enough data (I hope) to cover off the examples below of different types of joins and also give enough data to make them worthwhile.

So getting into the grit of it, the boss wants to know *The IDs of all the sports cars he has*.

This is a simple two table join. We have a table that identifies the model and the table with the available stock in it. As you can see, the data in the `model` column of the `cars` table relates to the `models` column of the `cars` table we have. Now, we know that the models table has an ID of `1` for `Sports` so lets write the join.

```
select
    ID,
    model
from
    cars
        join models
            on model=ID
```

So this query looks good right? We have identified the two tables and contain the information we need and use a join that correctly identifies what columns to join on.

```
ERROR 1052 (23000): Column 'ID' in field list is ambig
```

Oh noes! An error in our first query! Yes, and it is a plum. You see, the query has indeed got the right columns, but some of them exist in both tables, so the database gets confused about what actual column we mean and where. There are two solutions to solve this. The first is nice and simple, we can use `tableName.columnName` to tell the database exactly what we mean, like this:

```
select
    cars.ID,
    models.model
from
    cars
        join models
            on cars.model=models.ID
```

```
+----+--------+
| ID | model  |
+----+--------+
|  1 | Sports |
|  3 | Sports |
|  8 | Sports |
| 10 | Sports |
|  2 | Sedan  |
|  4 | Sedan  |
|  5 | 4WD    |
|  7 | 4WD    |
|  9 | 4WD    |
|  6 | Luxury |
+----+--------+
10 rows in set (0.00 sec)
```

The other is probably more often used and is called table aliasing. The tables in this example have nice and short simple names, but typing out something like `KPI_DAILY_SALES_BY_DEPARTMENT` would probably get old quickly, so a simple way is to nickname the table like this:

```
select
    a.ID,
    b.model
from
    cars a
        join models b
            on a.model=b.ID
```

Now, back to the request. As you can see we have the information we need, but we also have information that wasn't asked for, so we need to include a where clause in the statement to only get the Sports cars as was asked. As I prefer the table alias method rather than using the table names over and over, I will stick to it from this point onwards.

Clearly, we need to add a where clause to our query. We can identify Sports cars either by `ID=1` or `model='Sports'`. As the ID is indexed and the primary key (and it happens to be less typing), lets use that in our query.

```
select
    a.ID,
    b.model
from
    cars a
        join models b
            on a.model=b.ID
where
    b.ID=1

+----+--------+
| ID | model  |
+----+--------+
|  1 | Sports |
|  3 | Sports |
|  8 | Sports |
| 10 | Sports |
+----+--------+
4 rows in set (0.00 sec)
```

Bingo! The boss is happy. Of course, being a boss and never being happy with what he asked for, he looks at the information, then says *I want the colors as well*.

Okay, so we have a good part of our query already written, but we need to use a third table which is colors. Now, our main information table `cars` stores the car color

ID and this links back to the colors ID column. So, in a similar manner to the original, we can join a third table:

```
select
    a.ID,
    b.model
from
    cars a
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
where
    b.ID=1

+----+--------+
| ID | model  |
+----+--------+
|  1 | Sports |
|  3 | Sports |
|  8 | Sports |
| 10 | Sports |
+----+--------+
4 rows in set (0.00 sec)
```

Damn, although the table was correctly joined and the related columns were linked, we forgot to pull in the actual *information* from the new table that we just linked.

```
select
    a.ID,
    b.model,
    c.color
from
    cars a
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
```

```
where
    b.ID=1

+----+--------+-------+
| ID | model  | color |
+----+--------+-------+
|  1 | Sports | Red   |
|  8 | Sports | Green |
| 10 | Sports | White |
|  3 | Sports | Black |
+----+--------+-------+
4 rows in set (0.00 sec)
```

Right, that's the boss off our back for a moment. Now, to explain some of this in a little more detail. As you can see, the `from` clause in our statement links our main table (I often use a table that contains information rather than a lookup or dimension table. The query would work just as well with the tables all switched around, but make less sense when we come back to this query to read it in a few months time, so it is often best to try to write a query that will be nice and easy to understand - lay it out intuitively, use nice indenting so that everything is as clear as it can be. If you go on to teach others, try to instill these characteristics in their queries - especially if you will be troubleshooting them.

It is entirely possible to keep linking more and more tables in this manner.

```
select
    a.ID,
    b.model,
    c.color
from
    cars a
```

```
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
        join brands d
            on a.brand=d.ID
where
    b.ID=1
```

While I forgot to include a table where we might want to join more than one column in the `join` statement, here is an example. If the `models` table had brand-specific models and therefore also had a column called `brand` which linked back to the `brands` table on the `ID` field, it could be done as this:

```
select
    a.ID,
    b.model,
    c.color
from
    cars a
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
        join brands d
            on a.brand=d.ID
            and b.brand=d.ID
where
    b.ID=1
```

You can see, the query above not only links the joined tables to the main `cars` table, but also specifies joins between the already joined tables. If this wasn't done, the result is called a cartesian join - which is dba speak for bad. A cartesian join is one where rows are returned because the information doesn't tell the database how to limit the results, so the query returns *all* the rows that fit the criteria.

So, to give an example of a cartesian join, lets run the following query:

```
select
    a.ID,
    b.model
from
    cars a
        join models b

+----+---------+
| ID | model   |
+----+---------+
|  1 | Sports  |
|  1 | Sedan   |
|  1 | 4WD     |
|  1 | Luxury  |
|  2 | Sports  |
|  2 | Sedan   |
|  2 | 4WD     |
|  2 | Luxury  |
|  3 | Sports  |
|  3 | Sedan   |
|  3 | 4WD     |
|  3 | Luxury  |
|  4 | Sports  |
|  4 | Sedan   |
|  4 | 4WD     |
|  4 | Luxury  |
|  5 | Sports  |
|  5 | Sedan   |
|  5 | 4WD     |
|  5 | Luxury  |
|  6 | Sports  |
|  6 | Sedan   |
|  6 | 4WD     |
|  6 | Luxury  |
|  7 | Sports |
|  7 | Sedan  |
|  7 | 4WD    |
|  7 | Luxury |
|  8 | Sports |
|  8 | Sedan  |
|  8 | 4WD    |
|  8 | Luxury |
|  9 | Sports |
|  9 | Sedan  |
|  9 | 4WD    |
|  9 | Luxury |
| 10 | Sports |
| 10 | Sedan  |
| 10 | 4WD    |
| 10 | Luxury |
+----+--------+
40 rows in set (0.00 sec)
```

Good god, that's ugly. However, as far as the database is concerned, it is *exactly* what was asked for. In the query, we asked for for the `ID` from `cars` and the `model` from `models`. However, because we didn't specify *how* to join the tables, the database has matched *every* row from the first table with *every* row from the second table.

Okay, so the boss is back, and he wants more information again. *I want the same list, but also include 4WDs in it*.

This however, gives us a great excuse to look at two different ways to accomplish this. We could add another condition to the where clause like this:

```
select
    a.ID,
    b.model,
    c.color
from
```

```
    cars a
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
        join brands d
            on a.brand=d.ID
where
    b.ID=1
    or b.ID=3
```

While the above will work perfectly well, lets look at it differently, this is a great excuse to show how a `union` query will work.

We know that the following will return all the Sports cars:

```
select
    a.ID,
    b.model,
    c.color
from
    cars a
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
        join brands d
            on a.brand=d.ID
where
    b.ID=1
```

And the following would return all the 4WDs:

```
select
    a.ID,
    b.model,
    c.color
```

```
from
    cars a
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
        join brands d
            on a.brand=d.ID
where
    b.ID=3
```

So by adding a `union all` clause between them, the results of the second query will be appended to the results of the first query.

```
select
    a.ID,
    b.model,
    c.color
from
    cars a
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
        join brands d
            on a.brand=d.ID
where
    b.ID=1
union all
select
    a.ID,
    b.model,
    c.color
from
    cars a
        join models b
            on a.model=b.ID
        join colors c
            on a.color=c.ID
        join brands d
```

```
            on a.brand=d.ID
where
    b.ID=3

+----+--------+-------+
| ID | model  | color |
+----+--------+-------+
|  1 | Sports | Red   |
|  8 | Sports | Green |
| 10 | Sports | White |
|  3 | Sports | Black |
|  5 | 4WD    | Green |
|  7 | 4WD    | White |
|  9 | 4WD    | Black |
+----+--------+-------+
7 rows in set (0.00 sec)
```

As you can see, the results of the first query are returned first, followed by the results of the second query.

In this example, it would of course have been much easier to simply use the first query, but `union` queries can be great for specific cases. They are a great way to return specific results from tables from tables that aren't easily joined together - or for that matter *completely* unrelated tables. There are a few rules to follow however.

- The column types from the first query must match the column types from every other query below.
- The names of the columns from the first query will be used to identify the entire set of results.
- The number of columns in each query must be the same.

Now, you might be wondering what the difference is between using `union` and `union all`. A `union` query will remove duplicates, while a `union all` will not. This does mean that there is a small performance hit when using `union` over `union all` but the results may be worth it - I won't speculate on that sort of thing in this though.

On this note, it might be worth noting some additional notes here.

- If we wanted to order the results, we can use an `order by` but you can't use the alias anymore. In the query above, appending an `order by a.ID` would result in an error - as far as the results are concerned, the column is called `ID` rather than `a.ID` - even though the same alias has been used in both queries.
- We can only have one `order by` statement, and it must be as the last statement.

For the next examples, I am adding a few extra rows to our tables.

I have added `Holden` to the brands table. I have also added a row into `cars` that has the `color` value of `12` - which has no reference in the colors table.

Okay, the boss is back again, barking requests out - *I want a count of each brand we carry and the number of cars in it!` - Typical, we just get to an interesting section of our discussion and the boss wants more work.

Rightyo, so the first thing we need to do is get a complete listing of possible brands.

```
select
    a.brand
from
    brands a

+--------+
| brand  |
+--------+
| Ford   |
| Toyota |
| Nissan |
| Smart  |
| BMW    |
| Holden |
+--------+
6 rows in set (0.00 sec)
```

Now, when we join this to our cars table we get the following result:

```
select
    a.brand
from
    brands a
        join cars b
            on a.ID=b.brand
group by
    a.brand

+--------+
| brand  |
+--------+
| BMW    |
| Ford   |
| Nissan |
| Smart  |
| Toyota |
+--------+
5 rows in set (0.00 sec)
```

Which is of course a problem - we aren't seeing any mention of the lovely `Holden` brand I added.

This is because a join looks for matching rows in *both* tables. As there is no data in cars that is of type `Holden` it isn't returned. This is where we can use an `outer` join. This will return *all* the results from one table whether they are matched in the other table or not:

```
select
    a.brand
from
    brands a
        left outer join cars b
            on a.ID=b.brand
group by
    a.brand

+--------+
| brand  |
+--------+
| BMW    |
| Ford   |
| Holden |
| Nissan |
| Smart  |
| Toyota |
+--------+
6 rows in set (0.00 sec)
```

Now that we have that, we can add a lovely aggregate function to get a count and get the boss off our backs for a moment.

```
select
    a.brand,
    count(b.id) as countOfBrand
from
    brands a
        left outer join cars b
            on a.ID=b.brand
group by
    a.brand

+--------+--------------+
| brand  | countOfBrand |
+--------+--------------+
| BMW    |            2 |
| Ford   |            2 |
| Holden |            0 |
| Nissan |            1 |
| Smart  |            1 |
| Toyota |            5 |
+--------+--------------+
6 rows in set (0.00 sec)
```

And with that, away the boss skulks.

Now, to explain this in some more detail, outer joins can be of the `left` or `right` type. The Left or Right defines which table is *fully* included. A `left outer join` will include all the rows from the table on the left, while (you guessed it) a `right outer join` brings all the results from the table on the right into the results.

Some databases will allow a `full outer join` which will bring back results (whether matched or not) from *both* tables, but this isn't supported in all databases.

Now, I probably figure at this point in time, you are wondering whether or not you can merge join types in a query - and the answer is yes, you absolutely can.

```
select
    b.brand,
    c.color,
    count(a.id) as countOfBrand
from
    cars a
        right outer join brands b
            on b.ID=a.brand
        join colors c
            on a.color=c.ID
group by
    a.brand,
    c.color

+--------+-------+--------------+
| brand  | color | countOfBrand |
+--------+-------+--------------+
| Ford   | Blue  |            1 |
| Ford   | White |            1 |
| Toyota | Black |            1 |
| Toyota | Green |            2 |
| Toyota | Red   |            1 |
| Nissan | Black |            1 |
| Smart  | White |            1 |
| BMW    | Blue  |            1 |
| BMW    | White |            1 |
+--------+-------+--------------+
9 rows in set (0.00 sec)
```

So, why is that not the results that were expected? It is because although we have selected the outer join from cars to brands, it wasn't specified in the join to colors - so that particular join will only bring back results that match in both tables.

Here is the query that would work to get the results that we expected:

```
select
    a.brand,
    c.color,
    count(b.id) as countOfBrand
from
    brands a
        left outer join cars b
            on a.ID=b.brand
        left outer join colors c
            on b.color=c.ID
group by
    a.brand,
    c.color

+--------+-------+--------------+
| brand  | color | countOfBrand |
+--------+-------+--------------+
| BMW    | Blue  |            1 |
| BMW    | White |            1 |
| Ford   | Blue  |            1 |
| Ford   | White |            1 |
| Holden | NULL  |            0 |
| Nissan | Black |            1 |
| Smart  | White |            1 |
| Toyota | NULL  |            1 |
| Toyota | Black |            1 |
| Toyota | Green |            2 |
| Toyota | Red   |            1 |
+--------+-------+--------------+
11 rows in set (0.00 sec)
```

As we can see, we have two outer joins in the query and the results are coming through as expected.

Now, how about those other types of joins you ask? What about Intersections?

Well, not all databases support the `intersection` but pretty much all databases will allow you to create an intersection through a join (or a well structured where statement at the least).

An Intersection is a type of join somewhat similar to a `union` as described above - but the difference is that it *only* returns rows of data that are identical (and I do mean identical) between the various individual queries joined by the union. Only rows that are identical in every regard will be returned.

A simple example would be as such:

```
select
    *
from
    colors
where
    ID>2
intersect
select
    *
from
    colors
where
    id<4
```

While a normal `union` query would return all the rows of the table (the first query returning anything over `ID>2` and the second anything having `ID<4`) which would result in a full set, an intersect query would only return the row matching `id=3` as it meets both criteria.

Now, if your database doesn't support an `intersect` query, the above can be easily accomlished with the following query:

```
select
    a.ID,
    a.color,
    a.paint
from
    colors a
        join colors b
            on a.ID=b.ID
where
    a.ID>2
    and b.ID<4

+----+-------+----------+
| ID | color | paint    |
+----+-------+----------+
|  3 | Blue  | Metallic |
+----+-------+----------+
1 row in set (0.00 sec)
```

If you wish to perform an intersection across two different tables using a database that doesn't inherently support an intersection query, you will need to create a join on *every column* of the tables.

Share Improve this answer Follow

edited May 23, 2017 at 12:26

community wiki
12 revs, 5 users 99%
Fluffeh

3 @Fluffeh Nice answers. I have a suggestion: If you want to make it a killer SQL Tutorial, you are only missing to add Venn diagrams; I understood right away left and right joins thanks to them. Personal request: Do you have any tutorial

on common mistakes/performance tunning? – Ondino Oct 23, 2012 at 14:30

31 Oh my. My scroll wheel is broken. Great question and answer. I wish I could upvote this 10 times. – Amal Murali Sep 20, 2013 at 10:53

4 Hehe, thanks for the positive feedback. Keep scrolling though, this was only the first answer. SO said my answer was too long to fit it into one "answer" so I had to use a few :) – Fluffeh Sep 20, 2013 at 10:56

10 Honestly, I think this answer needs to be shortened somewhat. – einpoklum Dec 2, 2015 at 17:07

1 Excellent article. Database Joins 101. – maqs Jun 20, 2018 at 5:28

103

+100

Ok, I found this post very interesting and I would like to share some of my knowledge on creating a query. Thanks for this **Fluffeh**. Others who may read this and may feel that I'm wrong are 101% free to edit and criticise my answer. (*Honestly, I feel very thankful for correcting my mistake(s).*)

I'll be posting some of the frequently asked questions in `MySQL` tag.

Trick No. 1 (*rows that matches to multiple conditions*)

Given this schema

```
CREATE TABLE MovieList
(
    ID INT,
    MovieName VARCHAR(25),
    CONSTRAINT ml_pk PRIMARY KEY (ID),
    CONSTRAINT ml_uq UNIQUE (MovieName)
);

INSERT INTO MovieList VALUES (1, 'American Pie');
INSERT INTO MovieList VALUES (2, 'The Notebook');
INSERT INTO MovieList VALUES (3, 'Discovery Channel: A
INSERT INTO MovieList VALUES (4, 'Mr. Bean');
INSERT INTO MovieList VALUES (5, 'Expendables 2');

CREATE TABLE CategoryList
(
    MovieID INT,
    CategoryName VARCHAR(25),
    CONSTRAINT cl_uq UNIQUE(MovieID, CategoryName),
    CONSTRAINT cl_fk FOREIGN KEY (MovieID) REFERENCES
);

INSERT INTO CategoryList VALUES (1, 'Comedy');
INSERT INTO CategoryList VALUES (1, 'Romance');
INSERT INTO CategoryList VALUES (2, 'Romance');
INSERT INTO CategoryList VALUES (2, 'Drama');
INSERT INTO CategoryList VALUES (3, 'Documentary');
INSERT INTO CategoryList VALUES (4, 'Comedy');
INSERT INTO CategoryList VALUES (5, 'Comedy');
INSERT INTO CategoryList VALUES (5, 'Action');
```

QUESTION

Find *all movies* that belong to at least *both* `Comedy` and `Romance` categories.

Solution

This question can be very tricky sometimes. It may seem that a query like this will be the answer:-

```
SELECT  DISTINCT a.MovieName
FROM    MovieList a
        INNER JOIN CategoryList b
            ON a.ID = b.MovieID
WHERE   b.CategoryName = 'Comedy' AND
        b.CategoryName = 'Romance'
```

SQLFiddle Demo

which is definitely very wrong because it produces *no result*. The explanation of this is that there is only one valid value of `CategoryName` on *each row*. For instance, the first condition returns *true*, the second condition is always false. Thus, by using `AND` operator, both condition should be true; otherwise, it will be false. Another query is like this,

```
SELECT  DISTINCT a.MovieName
FROM    MovieList a
        INNER JOIN CategoryList b
            ON a.ID = b.MovieID
WHERE   b.CategoryName IN ('Comedy','Romance')
```

SQLFiddle Demo

and the result is still incorrect because it matches to record that has *at least* one match on the `categoryName`. The **real solution** *would be by counting the number of* *record instances per movie*. The number of instance should match to the total number of the values supplied in the condition.

```
SELECT  a.MovieName
FROM    MovieList a
        INNER JOIN CategoryList b
            ON a.ID = b.MovieID
WHERE   b.CategoryName IN ('Comedy','Romance')
GROUP BY a.MovieName
HAVING COUNT(*) = 2
```

SQLFiddle Demo (the answer)

- SQL of Relational Division

Trick No. 2 (*maximum record for each entry*)

Given schema,

```
CREATE TABLE Software
(
    ID INT,
    SoftwareName VARCHAR(25),
    Descriptions VARCHAR(150),
    CONSTRAINT sw_pk PRIMARY KEY (ID),
    CONSTRAINT sw_uq UNIQUE (SoftwareName)
);

INSERT INTO Software VALUES (1,'PaintMe','used for pho
INSERT INTO Software VALUES (2,'World Map','contains m
the world');
INSERT INTO Software VALUES (3,'Dictionary','contains
antonym of the words');

CREATE TABLE VersionList
(
    SoftwareID INT,
    VersionNo INT,
    DateReleased DATE,
    CONSTRAINT sw_uq UNIQUE (SoftwareID, VersionNo),
    CONSTRAINT sw_fk FOREIGN KEY (SOftwareID) REFERENC
);

INSERT INTO VersionList VALUES (3, 2, '2009-12-01');
INSERT INTO VersionList VALUES (3, 1, '2009-11-01');
INSERT INTO VersionList VALUES (3, 3, '2010-01-01');
INSERT INTO VersionList VALUES (2, 2, '2010-12-01');
INSERT INTO VersionList VALUES (2, 1, '2009-12-01');
INSERT INTO VersionList VALUES (1, 3, '2011-12-01');
INSERT INTO VersionList VALUES (1, 2, '2010-12-01');
INSERT INTO VersionList VALUES (1, 1, '2009-12-01');
INSERT INTO VersionList VALUES (1, 4, '2012-12-01');
```

QUESTION

Find the latest version on each software. Display the following columns: `SoftwareName`, `Descriptions`, `LatestVersion` (*from VersionNo column*), `DateReleased`

Solution

Some SQL developers mistakenly use `MAX()` aggregate function. They tend to create like this,

```
SELECT  a.SoftwareName, a.Descriptions,
        MAX(b.VersionNo) AS LatestVersion, b.DateRelea
FROM    Software a
        INNER JOIN VersionList b
            ON a.ID = b.SoftwareID
```

```
GROUP BY a.ID
ORDER BY a.ID
```

SQLFiddle Demo

(*most RDBMS generates a syntax error on this because of not specifying some of the non-aggregated columns on the* `group by` *clause*) the result produces the correct `LatestVersion` on each software but obviously the `DateReleased` are incorrect. `MySQL` doesn't support `Window Functions` and `Common Table Expression` yet as some RDBMS do already. The workaround on this problem is to create a `subquery` which gets the individual maximum `versionNo` on each software and later on be joined on the other tables.

```
SELECT  a.SoftwareName, a.Descriptions,
        b.LatestVersion, c.DateReleased
FROM    Software a
        INNER JOIN
        (
            SELECT  SoftwareID, MAX(VersionNO) LatestV
            FROM    VersionList
            GROUP BY SoftwareID
        ) b ON a.ID = b.SoftwareID
        INNER JOIN VersionList c
            ON  c.SoftwareID = b.SoftwareID AND
                c.VersionNO = b.LatestVersion
GROUP BY a.ID
ORDER BY a.ID
```

SQLFiddle Demo (the answer)

So that was it. I'll be posting another soon as I recall any other *FAQ* on `MySQL` tag. Thank you for reading this little article. I hope that you have atleast get even a little knowledge from this.

UPDATE 1

Trick No. 3 (*Finding the latest record between two IDs*)

Given Schema

```
CREATE TABLE userList
(
    ID INT,
    NAME VARCHAR(20),
    CONSTRAINT us_pk PRIMARY KEY (ID),
    CONSTRAINT us_uq UNIQUE (NAME)
);

INSERT INTO userList VALUES (1, 'Fluffeh');
INSERT INTO userList VALUES (2, 'John Woo');
INSERT INTO userList VALUES (3, 'hims056');

CREATE TABLE CONVERSATION
(
    ID INT,
    FROM_ID INT,
    TO_ID INT,
    MESSAGE VARCHAR(250),
    DeliveryDate DATE
);

INSERT INTO CONVERSATION VALUES (1, 1, 2, 'hi john', '
INSERT INTO CONVERSATION VALUES (2, 2, 1, 'hello fluff
INSERT INTO CONVERSATION VALUES (3, 1, 3, 'hey hims',
INSERT INTO CONVERSATION VALUES (4, 1, 3, 'please repl
INSERT INTO CONVERSATION VALUES (5, 3, 1, 'how are you
INSERT INTO CONVERSATION VALUES (6, 3, 2, 'sample mess
```

QUESTION

Find the latest conversation between two users.

Solution

```
SELECT    b.Name SenderName,
          c.Name RecipientName,
          a.Message,
          a.DeliveryDate
FROM      Conversation a
          INNER JOIN userList b
            ON a.From_ID = b.ID
          INNER JOIN userList c
            ON a.To_ID = c.ID
WHERE     (LEAST(a.FROM_ID, a.TO_ID), GREATEST(a.FROM_
DeliveryDate)
IN
(
    SELECT  LEAST(FROM_ID, TO_ID) minFROM,
            GREATEST(FROM_ID, TO_ID) maxTo,
            MAX(DeliveryDate) maxDate
    FROM    Conversation
    GROUP BY minFROM, maxTo
)
```

SQLFiddle Demo

Share Improve this answer Follow

edited May 23, 2017 at 12:26

community wiki

11 revs, 3 users 96%
John Woo

Awesome! A caveat John, your first solution works only because there is a unique constraint on the two fields. You could have used a more general solution to help with a common problem. In my opinion the only solution is to do individual selects for `comedy` and `romance`. `Having` doesnt suit then.. – nawfal Nov 13, 2012 at 15:14

@nawfal not really, if unique constraint was not added, you then need to add `distinct` on the having clause [SQLFiddle Demo](#) :D – John Woo Nov 13, 2012 at 15:31

65

Part 2 - Subqueries

Okay, now the boss has burst in again - *I want a list of all of our cars with the brand and a total of how many of that brand we have!*

This is a great opportunity to use the next trick in our bag of SQL goodies - the subquery. If you are unfamiliar with the term, a subquery is a query that runs inside another query. There are many different ways to use them.

For our request, lets first put a simple query together that will list each car and the brand:

```
select
    a.ID,
    b.brand
from
    cars a
```

```
        join brands b
            on a.brand=b.ID
```

Now, if we wanted to simply get a count of cars sorted by brand, we could of course write this:

```
select
    b.brand,
    count(a.ID) as countCars
from
    cars a
        join brands b
            on a.brand=b.ID
group by
    b.brand

+--------+-----------+
| brand  | countCars |
+--------+-----------+
| BMW    |         2 |
| Ford   |         2 |
| Nissan |         1 |
| Smart  |         1 |
| Toyota |         5 |
+--------+-----------+
```

So, we should be able to simply add in the count function to our original query right?

```
select
    a.ID,
    b.brand,
    count(a.ID) as countCars
from
    cars a
        join brands b
            on a.brand=b.ID
group by
    a.ID,
    b.brand

+----+--------+-----------+
| ID | brand  | countCars |
+----+--------+-----------+
|  1 | Toyota |         1 |
|  2 | Ford   |         1 |
|  3 | Nissan |         1 |
|  4 | Smart  |         1 |
|  5 | Toyota |         1 |
|  6 | BMW    |         1 |
|  7 | Ford   |         1 |
|  8 | Toyota |         1 |
|  9 | Toyota |         1 |
| 10 | BMW    |         1 |
| 11 | Toyota |         1 |
+----+--------+-----------+
11 rows in set (0.00 sec)
```

Sadly, no, we can't do that. The reason is that when we add in the car ID (column a.ID) we have to add it into the group by - so now, when the count function works, there is only one ID matched per ID.

This is where we can however use a subquery - in fact we can do two completely different types of subquery that will return the same results that we need for this. The first is to simply put the subquery in the `select` clause. This means each time we get a row of data, the subquery will run off, get a column of data and then pop it into our row of data.

```
select
    a.ID,
    b.brand,
    (
    select
        count(c.ID)
    from
        cars c
    where
        a.brand=c.brand
    ) as countCars
from
    cars a
        join brands b
            on a.brand=b.ID

+----+--------+-----------+
| ID | brand  | countCars |
+----+--------+-----------+
|  2 | Ford   |         2 |
|  7 | Ford   |         2 |
|  1 | Toyota |         5 |
|  5 | Toyota |         5 |
|  8 | Toyota |         5 |
|  9 | Toyota |         5 |
| 11 | Toyota |         5 |
|  3 | Nissan |         1 |
|  4 | Smart  |         1 |
|  6 | BMW    |         2 |
| 10 | BMW    |         2 |
+----+--------+-----------+
11 rows in set (0.00 sec)
```

And Bam!, this would do us. If you noticed though, this sub query will have to run for each and every single row of data we return. Even in this little example, we only have five different Brands of car, but the subquery ran eleven times as we have eleven rows of data that we are returning. So, in this case, it doesn't seem like the most efficient way to write code.

For a different approach, lets run a subquery and pretend it is a table:

```
select
    a.ID,
    b.brand,
    d.countCars
from
    cars a
        join brands b
            on a.brand=b.ID
        join
            (
            select
                c.brand,
                count(c.ID) as countCars
            from
                cars c
            group by
                c.brand
            ) d
            on a.brand=d.brand

+----+--------+-----------+
| ID | brand  | countCars |
+----+--------+-----------+
|  1 | Toyota |         5 |
|  2 | Ford   |         2 |
|  3 | Nissan |         1 |
|  4 | Smart  |         1 |
|  5 | Toyota |         5 |
|  6 | BMW    |         2 |
|  7 | Ford   |         2 |
|  8 | Toyota |         5 |
|  9 | Toyota |         5 |
| 10 | BMW    |         2 |
| 11 | Toyota |         5 |
+----+--------+-----------+
11 rows in set (0.00 sec)
```

Okay, so we have the same results (ordered slightly different - it seems the database wanted to return results ordered by the first column we picked this time) - but the same right numbers.

So, what's the difference between the two - and when should we use each type of subquery? First, lets make sure we understand how that second query works. We selected two tables in the `from` clause of our query, and then wrote a query and told the database that it was in fact a table instead - which the database is perfectly happy with. There *can* be some benefits to using this method (as well as some limitations). Foremost is that this subquery ran *once*. If our database contained a large volume of data, there could well be a massive improvement over the first method. However, as we are using this as a table, we have to bring in extra rows of data - so that they can actually be joined back to our rows of data. We also have to be sure that there are *enough* rows of data if we are going to use a simple join like in the query above. If you recall, the join will only pull back rows that have matching data on *both* sides of the join. If we aren't careful, this could result in valid data not being returned from our cars table if there wasn't a matching row in this subquery.

Now, looking back at the first subquery, there are some limitations as well. because we are pulling data back into a single row, we can *ONLY* pull back one row of data. Subqueries used in the `select` clause of a query very often use only an aggregate function such as `sum`, `count`, `max` or another similar aggregate function. They don't *have* to, but that is often how they are written.

So, before we move on, lets have a quick look at where else we can use a subquery. We can use it in the `where` clause - now, this example is a little contrived as in our database, there are better ways of getting the following data, but seeing as it is only for an example, lets have a look:

```
select
    ID,
    brand
from
    brands
where
    brand like '%o%'

+----+--------+
| ID | brand  |
+----+--------+
|  1 | Ford   |
|  2 | Toyota |
|  6 | Holden |
+----+--------+
3 rows in set (0.00 sec)
```

This returns us a list of brand IDs and Brand names (the second column is only added to show us the brands) that contain the letter `o` in the name.

Now, we could use the results of this query in a where clause this:

```
select
    a.ID,
    b.brand
from
    cars a
        join brands b
            on a.brand=b.ID
where
    a.brand in
```

```
         (
         select
             ID
         from
             brands
         where
             brand like '%o%'
         )

+----+--------+
| ID | brand  |
+----+--------+
|  2 | Ford   |
|  7 | Ford   |
|  1 | Toyota |
|  5 | Toyota |
|  8 | Toyota |
|  9 | Toyota |
| 11 | Toyota |
+----+--------+
7 rows in set (0.00 sec)
```

As you can see, even though the subquery was returning the three brand IDs, our cars table only had entries for two of them.

In this case, for further detail, the subquery is working as if we wrote the following code:

```
select
    a.ID,
    b.brand
from
    cars a
        join brands b
            on a.brand=b.ID
where
    a.brand in (1,2,6)

+----+--------+
| ID | brand  |
+----+--------+
|  1 | Toyota |
|  2 | Ford   |
|  5 | Toyota |
|  7 | Ford   |
|  8 | Toyota |
|  9 | Toyota |
| 11 | Toyota |
+----+--------+
7 rows in set (0.00 sec)
```

Again, you can see how a subquery vs manual inputs has changed the order of the rows when returning from the database.

While we are discussing subqueries, lets see what else we can do with a subquery:

- You can place a subquery within another subquery, and so on and so on. There is a limit which depends on your database, but short of recursive functions of some insane and maniacal programmer, most folks will never hit that limit.
- You can place a number of subqueries into a single query, a few in the `select` clause, some in the `from` clause and a couple more in the `where` clause - just remember that each one you put in is making your query more complex and likely to take longer to execute.

If you need to write some efficient code, it can be beneficial to write the query a number of ways and see (either by timing it or by using an explain plan) which is the optimal query to get your results. The first way that works may not always be the best way of doing it.

Share Improve this answer Follow

answered Sep 19, 2012 at 9:47

community wiki
Fluffeh

1 Very important for new devs: subqueries probably run once for every result *unless* you can use the sub-query as a join (shown above). – Xeoncross Jan 15, 2019 at 20:12

60

Part 3 - Tricks and Efficient Code

MySQL in() efficiency

I thought I would add some extra bits, for tips and tricks that have come up.

One question I see come up a fair bit, is *How do I get non-matching rows from two tables* and I see the answer most commonly accepted as something like the following (based on our cars and brands table - which has *Holden* listed as a brand, but does not appear in the cars table):

```
select
    a.ID,
    a.brand
```

```
from
    brands a
where
    a.ID not in(select brand from cars)
```

And *yes* it will work.

```
+----+--------+
| ID | brand  |
+----+--------+
|  6 | Holden |
+----+--------+
1 row in set (0.00 sec)
```

However it is *not* efficient in some database. Here is a link to a Stack Overflow question asking about it, and here is an excellent in depth article if you want to get into the nitty gritty.

The short answer is, if the optimiser doesn't handle it efficiently, it may be much better to use a query like the following to get non matched rows:

```
select
    a.brand
from
    brands a
        left join cars b
            on a.id=b.brand
where
    b.brand is null

+--------+
| brand  |
+--------+
| Holden |
+--------+
1 row in set (0.00 sec)
```

Update Table with same table in subquery

Ahhh, another oldie but goodie - the old *You can't specify target table 'brands' for update in FROM clause*.

MySQL will not allow you to run an `update...` query with a subselect on the same table. Now, you might be thinking, why not just slap it into the where clause right? But what if you want to update only the row with the `max()` date amoung a bunch of other rows? You can't exactly do that in a where clause.

```
update
    brands
set
    brand='Holden'
where
    id=
        (select
            id
        from
            brands
        where
            id=6);
ERROR 1093 (HY000): You can't specify target table 'br
for update in FROM clause
```

So, we can't do that eh? Well, not exactly. There is a sneaky workaround that a surprisingly large number of users don't know about - though it does include some hackery that you will need to pay attention to.

You can stick the subquery within another subquery, which puts enough of a gap between the two queries so that it will work. However, note that it might be safest to stick the query within a transaction - this will prevent any other changes being made to the tables while the query is running.

```
update
    brands
set
    brand='Holden'
where id=
    (select
        id
    from
        (select
            id
        from
            brands
        where
            id=6
        )
    as updateTable);

Query OK, 0 rows affected (0.02 sec)
Rows matched: 1  Changed: 0  Warnings: 0
```

Share Improve this answer Follow

edited May 23, 2017 at 11:47

community wiki
7 revs, 5 users 94%
Fluffeh

4 Just want to note that the WHERE NOT EXISTS() construction is pretty much identical from an 'efficiency point of view' but in my opinion much easier to read/understand. Then again, my knowledge is limited to MSSQL and I can't vow if the same is true on other platforms. – deroby Sep 24, 2012 at 21:21

I just tried this type of comparison the other day, where the NOT IN() had a list about several hundred ID's and there was no difference between it and the join version of the query. Perhaps it makes a difference when you get up into the thousands or billions. – Buttle Butkus Sep 27, 2012 at 4:19

19

You can use the concept of multiple queries in the FROM keyword. Let me show you one example:

```
SELECT DISTINCT e.id,e.name,d.name,lap.lappy LAPTOP_MA
FROM  (
          SELECT c.id cnty,l.name
          FROM   county c, location l
          WHERE  c.id=l.county_id AND l.end_Date IS NO
      ) c_loc, emp e
      INNER JOIN dept d ON e.deptno =d.id
      LEFT JOIN
      (
         SELECT l.id lappy, c.name cmpy
         FROM   laptop l, company c
         WHERE l.make = c.name
      ) lap ON e.cmpy_id=lap.cmpy
```

You can use as many tables as you want to. Use outer joins and union where ever it's necessary, even inside table subqueries.

That's a very easy method to involve as many as tables and fields.

Share Improve this answer Follow

edited Sep 27, 2012 at 4:27

community wiki
6 revs, 2 users 76%
prashant1988

10

Hopes this makes it find the tables as you're reading through the thing:

jsfiddle

```
mysql> show columns from colors;
+-------+-------------+------+-----+---------+--------
| Field | Type        | Null | Key | Default | Extra
+-------+-------------+------+-----+---------+--------
| id    | int(3)      | NO   | PRI | NULL    | auto_in
| color | varchar(15) | YES  |     | NULL    |
| paint | varchar(10) | YES  |     | NULL    |
+-------+-------------+------+-----+---------+--------
```

Share Improve this answer Follow

edited May 19, 2016 at 9:33

community wiki
2 revs, 2 users 85%
Anton Chan

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.