

MySQL Introduction

Scott D. Anderson

Plan

- Part 1: Simple Queries
- Part 2: Creating a database
- Part 3: Joining tables
- Part 4: complex queries:
 - grouping
 - aggregate functions
 - subqueries
 - sorting
- Reference: <https://cs.wellesley.edu/~cs304/downloads/>



Files for this Part

```
$ cd ~/cs304
```

```
$ cp -r ~/cs304/pub/downloads/part4 .
```

```
$ cd part4
```



Grouping and Subqueries

Our topics for this part:

- Grouping: reporting or computing values for groups of rows, defined by whatever criteria we want
- Subqueries: a query can return a set of rows, which we can use as part of a more complex query

These can be combined in interesting ways.

We'll also look at sorting



The CGM data

People with type 1 diabetes sometimes have an implanted sensor that reads their blood glucose and can have that data wirelessly transmitted to a database in the cloud.

The CGM table in the WEBDB database is some sanitized data of that kind:

- the user is fake (Sonia Sotomayor has type 1 diabetes; I used her name.)
- the timestamps are accurate
- the `mgdl` value is the original value with a random value added



Counting Rows

It's useful to see how many rows are in our tables. Here's a way:

```
mysql> use wmdb;  
mysql> select count(*) from movie;  
mysql> select count(*) from person;  
mysql> select count(*) from credit;  
mysql> use webdb;  
mysql> select count(*) from cgm;
```



Search time

MySQL reports how long a query takes. Mostly we will not worry about performance, but here's an interesting example:

```
mysql> select * from cgm where rec_num = 700001;
```

```
mysql> select * from cgm where date = '2017-03-10 12:37:00';
```

You can see that these queries look up exactly the same row.

Why the difference in execution time?



The DISTINCT keyword

We often have repeated values in a database. It's nice to filter out repeats. Try:

```
mysql> use wmdb;  
mysql> select `release` from movie limit 10;  
mysql> select distinct `release` from movie;  
mysql> select distinct `release` from movie  
    -> where `release` > 2010;
```



Grouping movies

How many movies are there in WMDB released after 2010? How many each year?

Run `group1.sql`:

```
select `release`, count(*)  
from movie  
where `release` > '2010'  
group by `release`;
```

Omit the WHERE clause to see counts for all years.



The GROUP BY clause

- the GROUP BY clause causes the rows to be grouped by some criterion
- the GROUP BY clause goes **after** the WHERE clause.
- there is **one** result for **each** distinct value of the grouping criterion
- here, it's one for each distinct value of the release year
- You can also group by a boolean. Let's look at `group2.sql`
 - We can specify a column header, like 'number of movies'
 - a boolean value of 1 means true, and 0 means false
 - a NULL value for the release year is **neither true nor false**
 - that is, `NULL >= '2000'` is NULL



Uniformity across a Group

Let's run `group3.sql`, look at the code, and discuss:

- If you ask for a value that **varies** across the group,
 - You might get a value from some arbitrary row in the group, or
 - You might get an error, because
- This query makes no sense.
- Why? Because **each** movie has a title, but the **group** of movies does **not** have a title, and yet we have asked for the title column, even though it varies across the group.



Aggregate Functions

One useful thing to do is to compute some property of the group, using an aggregate function or GROUP BY function. The most popular:

- `count()`
- `sum()`
- `max()`
- `min()`
- `avg()`



Exercise

Switch to the **webdb** database and search the **cgm** table to find:

- how many different users are in the table?
- how many different years are in the table?
- how many data values in each year?

How could we get counts for each year in an easy way?



Solution

See `group4.sql`



Groups having a property

With grouping and aggregate functions, we can compute **properties** of a group.

We can report **only** groups having a particular property.

- readings with the same timestamp: `having1.sql`
- min/max readings with same timestamp: `having2.sql`
- readings that are literally redundant: `having3.sql`



the HAVING clause

```
SELECT ... FROM table  
GROUP BY something  
HAVING predicate involving aggregates
```

the HAVING clause filters *groups*, not rows.



Exercise on Having

In the **WMDB**, find movies having at least 8 actors



Exercise: finding lows

Switch to the **webdb** database and search the **cgm** table to find:

- the lowest value of mgdl



Solution

Did you try these (from select1.sql):

```
select user,count(*) from cgm group by user;  
select year(date),count(*) from cgm group by year(date);  
select min(mgdl) from cgm;  
select min(mgdl) from cgm where mgdl <> '';
```

Did you find the low of 29?



Explanation

The low was hard to find because the mgdl is a string and

`'100' < '29'`

for the same reason that `'add' < 'by'`

Strings are compared in dictionary order.

So, let's convert the mgdl to integer and do a few other things



A better data representation

```
create table cgm2 (  
    date datetime,  
    mgdl mediumint,  
    rec_num int,  
    primary key (rec_num),  
    index(date)  
);
```

The index will make lookup by date faster.



Migrating the data

```
insert into cgm2  
select date, mgdl, rec_num  
from cgm where mgdl <> '';
```

The datatypes are automatically transformed if possible and the index on date is built.

See `cgm2-table.sql` for the code. I ran that batch file.



Index speed

Compare the speed for looking up `rec_num = 700,001` in both `cgm` and `cgm2`.

Compare the speed for looking up the same record by date:

`2017-03-10 12:37:00`

Also, compute the min `mgdl`, as in `select2.sql`



When did the low happen?

```
-- this works fine  
select min(mgd1) from cgm2;
```

```
-- this does *not* return what we want  
select date_time, min(mgd1) from cgm2;
```

```
-- see?  
select * from cgm2 where date_time = '2014-05-27 22:48:00';
```

From select3.sql



Searching for the low

This works, but assumes we know the value that we are looking for:

```
select * from cgm2  
where mgdl = 29;
```

See `select4.sql`

What we need is to replace that 29 with a query to find the value of the low.



Subqueries to the rescue!

This works great:

```
select * from cgm2  
where mgdl = (select min(mgdl) from cgm2);
```

Try it using `select5.sql`



More on Subqueries

- subqueries can return **one** value. See also `peter-jackson-movies.sql`
- subqueries can return multiple values, searched using IN or NOT IN. See `movies-with-actors.sql`, `movies-with-n-actors.sql`, `movies-without-actors.sql`
- subqueries can return multiple columns. We can give these temporary names and use joins. See `scott-movies.sql` and `count-dups.sql`




Sorting

Among your final clauses, you can sort the results:

```
SELECT .. FROM table ORDER BY col1 ASC, col2 DESC;
```

See `popular-years.sql`



Summary of Part 4

- Rows can be **grouped** by column values.
- All values must be uniform over the group.
- Groups can be filtered using a **HAVING** clause.
- Subqueries allow SQL statements to be nested
- The inner query can return
 - a constant
 - a column, searched by IN or NOT IN
 - several columns, used as a temporary table
- Results can be sort using the **ORDER BY** clause



Wildcard expressions

- Sometimes, strict equality is too strict. You want to match a **set** of strings
- The set can have wildcard characters that match many characters:
 - `_` to match exactly one character, but any character
 - `%` to match zero or more of any character
- Examples:

```
... where name like 'P%'; -- starts with P
```

```
... where name like '%son'; -- ends with "son"
```

```
... where name like 'P%son'; -- starts with P, ends in  
"son"
```