Databases - SQL3

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

This lecture

This lecture focuses on the *summary* or *aggregate* features provided in MySQL.

These functions all have the following effect:

- The "candidate rows"[1] are collected into *groups*
- Each *group* contributes *just one row* to the output table
- Used to provide *summary data* about the group in some way

[1] In other words, the rows of the intermediate table that results from all the joins and selections in the `FROM` clause

Summary Functions

One of the main uses of a database is to *summarize* the data it contains, in particular to provide *statistical data*.

The main summary functions are

- `COUNT` – to *count* rows
- `SUM` – to *add* the values in a column
- `MIN` – to find the *minimum* value in a column
- `MAX` – to find the *maximum* value in a column
- `AVG` – to find the *average* value in a column
- `STD` – to find the *standard deviation* of the values in a column

Structure of a summary

If *any* of the output columns use any of the aggregate functions, then either

- There is a `GROUP BY` attribute specified
  The rows are formed into groups according to that attribute, and the output table will contain one row per group
- There is no `GROUP BY` attribute specified
  The entire table is treated as one group, and the output table will contain just one row

Just one group

If there is an aggregate function present (in this case `SUM`) and no `GROUP BY`, then there will be one row output.

```
SELECT SUM(Population)
FROM City;
+-----------------+
| SUM(Population) |
+-----------------+
|      1429559884 |
+-----------------+
1 row in set (0.00 sec)
```

How to think about this?

Behind the scenes

The `FROM` table is `City`, which we recall looks something like this

```
SELECT * FROM City;
+----+----------------+-------------+------------+
| ID | Name           | CountryCode | Population |
+----+----------------+-------------+------------+
|  1 | Kabul          | AFG         |    1780000 |
|  2 | Qandahar       | AFG         |     237500 |
|  3 | Herat          | AFG         |     186800 |
|  4 | Mazar-e-Sharif | AFG         |     127800 |
...
```

What was selected?

The presence of the `SUM(Population)` had the effect of

- Forming all the rows into one large group
- Adding up the `Population` field of each row
- Producing a single row as output

More than one aggregate function

The data can be summarised in several ways at once

```
SELECT MIN(Population), MAX(Population)
FROM City;
+-----------------+-----------------+
| MIN(Population) | MAX(Population) |
+-----------------+-----------------+
|              42 |        10500000 |
+-----------------+-----------------+
1 row in set (0.00 sec)
```

Renaming

The usual rules apply for renaming

```
SELECT MIN(Population) AS smallest,
       MAX(Population) AS biggest
FROM City;
+----------+----------+
| smallest | biggest  |
+----------+----------+
|       42 | 10500000 |
+----------+----------+
1 row in set (0.00 sec)
```

French cities

```
SELECT MIN(Population),
       MAX(Population)
FROM City
WHERE CountryCode = 'FRA';
+-----------------+-----------------+
| MIN(Population) | MAX(Population) |
+-----------------+-----------------+
|           90674 |         2125246 |
+-----------------+-----------------+
```

German cities

```
SELECT MIN(Population),
       MAX(Population)
FROM City
WHERE CountryCode = 'DEU';
+-----------------+-----------------+
| MIN(Population) | MAX(Population) |
+-----------------+-----------------+
|           89667 |         3386667 |
+-----------------+-----------------+
```

What do we really want?

What we *really* want is to be able to do is:

- summarise the data for *each country individually*, but
- get the results for *all the countries at once*.

This is the purpose of the `GROUP BY` statement.

Grouping

```
SELECT MIN(Population),
     MAX(Population)
FROM City
GROUP BY CountryCode;
+-----------------+-----------------+
| MIN(Population) | MAX(Population) |
+-----------------+-----------------+
|           29034 |           29034 |
|          127800 |         1780000 |
|          118200 |         2022000 |
|             595 |             961 |
|          270000 |          270000 |
|           21189 |           21189 |
|            2345 |            2345
....
```

What is happening?

First the rows are grouped by `CountryCode` — we can "simulate" this grouping by using the `ORDER BY` statement.

```
SELECT * FROM City
ORDER BY CountryCode;
+-----+-------------------+-------------+------------+
| ID  | Name              | CountryCode | Population |
+-----+-------------------+-------------+------------+
| 129 | Oranjestad        | ABW         |      29034 |
|   1 | Kabul             | AFG         |    1780000 |
|   4 | Mazar-e-Sharif    | AFG         |     127800 |
|   3 | Herat             | AFG         |     186800 |
|   2 | Qandahar          | AFG         |     237500 |
|  58 | Lobito            | AGO         |     130000 |
|  59 | Benguela          | AGO         |     128300 |
|  57 | Huambo            | AGO         |     163100 |
|  56 | Luanda            | AGO         |    2022000 |
|  60 | Namibe            | AGO         |     118200 |
|  62 | The Valley        | AIA         |        595 |
|  61 | South Hill        | AIA         |        961 |
```

Grouping

So the first group is

```
+-----+-------------------+-------------+------------+
| ID  | Name              | CountryCode | Population |
+-----+-------------------+-------------+------------+
| 129 | Oranjestad        | ABW         |      29034 |
+-----+-------------------+-------------+------------+
```

and so the requested summary data for *that group* is the first row of the output.

Grouping 2

The second group is

ID	Name	CountryCode	Population
1	Kabul	AFG	1780000
4	Mazar-e-Sharif	AFG	127800
3	Herat	AFG	186800
2	Qandahar	AFG	237500

and so the summary line for that group is

MIN(Population)	MAX(Population)
127800	1780000

But we want more

Ideally though, we want each summary line to be labelled so that the *group* can be identified.

```
SELECT CountryCode,
       MIN(Population),
       MAX(Population)
FROM City
GROUP BY CountryCode;
+-------------+-----------------+-----------------+
| CountryCode | MIN(Population) | MAX(Population) |
+-------------+-----------------+-----------------+
| ABW         |           29034 |           29034 |
| AFG         |          127800 |         1780000 |
| AGO         |          118200 |         2022000 |
| AIA         |             595 |             961 |
| ALB         |          270000 |          270000 |
| AND         |           21189 |           21189 |
| ANT         |            2345 |            2345 |
| ARE         |          114395 |          669181
```

How to understand this

The `SELECT` statement specified three output columns — two were aggregate functions, but one was *not* an aggregate function.

This only makes sense if the non-aggregate output columns are *constant* on the groups — in particular, this will be true if the non-aggregate output columns are all `GROUP BY` columns.

However, `MySQL` does *not enforce* this rule.

This lecture

We continue coverage of the aggregate functions of SQL

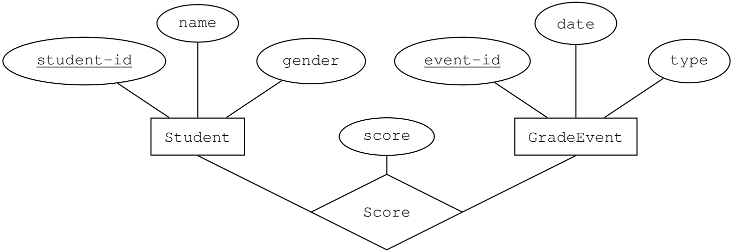
Another example

For this lecture we'll use an example based on Paul Dubois's book *MySQL*.

The database is to be used to keep student marks while taking a particular unit.

- `Students` have a first name, a gender and a unique student number
- `GradeEvents` are either *tests* or *quizzes* and happen on a particular date
- Students tests or quizzes and get a *score* for that particular "event"

The ER diagram



Creating the tables

```
CREATE TABLE Student (
  name          VARCHAR(20) NOT NULL,
  gender        ENUM('F','M') NOT NULL,
  student_id    INT NOT NULL AUTO_INCREMENT,
  PRIMARY KEY(student_id)
) ENGINE = InnoDB;
```

This contains a few things we have already seen, but a couple of new ones, a `PRIMARY KEY` and the statement `NOT NULL`.

Keys

A *key* for a relation / table is an attribute that cannot contain *repeated values*.

We think of it as a value that is enough to *uniquely identify* a row in the table.

For example, a student number uniquely identifies a student, so a table containing two rows with the same student number is probably corrupt and likely to be problematic.

A key can be a single attribute, a combination of attributes, or an artificial identifier (like student number).

The table

```
SELECT *
FROM Student;
+-----------+--------+------------+
| name      | gender | student_id |
+-----------+--------+------------+
| Megan     | F      |          1 |
| Joseph    | M      |          2 |
| Kyle      | M      |          3 |
| Katie     | F      |          4 |
...
| Gabrielle | F      |         29 |
| Grace     | F      |         30 |
| Emily     | F      |         31 |
+-----------+--------+------------+
31 rows in set (0.00 sec)
```

Enforcing key constraints

Attempting to insert a row with a duplicate value will fail.

```
INSERT INTO Student
   VALUES('James','M',31);
ERROR 1062 (23000): Duplicate entry '31' for key 'PRIMARY'
```

This is an instance of the many ways in which `SQL` attempts to ensure *data integrity* — that the data is the database is internally consistent.

“Don’t care” values

We don’t actually care *which* student number is given to James, so declare the field to be `AUTO_INCREMENT`.

```
INSERT INTO Student
  VALUES('James','M',NULL);
Query OK, 1 row affected (0.00 sec)
```

Hmm, what `student_id` has James been given?

```
+-----------+--------+------------+
| name      | gender | student_id |
+-----------+--------+------------+
| Megan     | F      |          1 |
| Joseph    | M      |          2 |
....
| Emily     | F      |         31 |
| James     | M      |         32 |
+-----------+--------+------------+
```

Creating the tables

```
CREATE TABLE GradeEvent (
    date        DATE NOT NULL,
    category    ENUM('T','Q') NOT NULL,
    event_id    INT NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (event_id)
) ENGINE = InnoDB;
```

The score table

```
CREATE TABLE Score (
 student_id      INT  NOT NULL,
 event_id        INT  NOT NULL,
 score           INT NOT NULL,
 PRIMARY KEY (event_id, student_id),
 FOREIGN KEY (event_id)
   REFERENCES GradeEvent (event_id),
 FOREIGN KEY (student_id)
   REFERENCES Student (student_id)
) ENGINE = InnoDB;
```

This contains one major new feature — the `FOREIGN KEY` constraints on the attributes `event_id` and `student_id`.

The data

```
SELECT *
FROM GradeEvent;
+------------+----------+----------+
| date       | category | event_id |
+------------+----------+----------+
| 2014-09-03 | Q        |        1 |
| 2014-09-06 | Q        |        2 |
| 2014-09-09 | T        |        3 |
| 2014-09-16 | Q        |        4 |
| 2014-09-23 | Q        |        5 |
| 2014-10-01 | T        |        6 |
+------------+----------+----------+
6 rows in set (0.01 sec)
```

The data

```
mysql> SELECT * FROM Score;
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          1 |        1 |    20 |
|          3 |        1 |    20 |
|          4 |        1 |    18 |
...
...
|         28 |        6 |    77 |
|         29 |        6 |    66 |
|         30 |        6 |    68 |
|         31 |        6 |    76 |
+------------+----------+-------+
173 rows in set (0.00 sec)
```

Counting students

How many students are in the class?

```
SELECT COUNT(*) FROM student;
+----------+
| COUNT(*) |
+----------+
|       31 |
+----------+
```

The `COUNT` function says to count the number of rows that are returned by the `SELECT` statement.

This syntax is strange at first sight, but interpreting `COUNT` as just another *summary function* makes it seem much more logical

How many men and women?

Use the `WHERE` clause to limit the chosen rows.

```
SELECT COUNT(*)
FROM student
WHERE gender = 'M';
+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+

SELECT
COUNT(*) FROM student
WHERE gender = 'F';
+----------+
| COUNT(*) |
+----------+
|       15 |
+----------+
```

With one statement

We can count both men and women in a single statement by using the `GROUP BY` clause — recall that this first *groups the rows* and then summarises each group into a single summary row.

```
SELECT COUNT(*)
FROM   student
GROUP  BY gender;
+----------+
| COUNT(*) |
+----------+
|       15 |
|       16 |
+----------+
```

But which is which

As it stands, we don't know which value is associated with which gender!

```
SELECT gender, COUNT(*)
FROM student
GROUP BY gender;
+-----+----------+
| gender | COUNT(*) |
+-----+----------+
| F    |       15 |
| M    |       16 |
+-----+----------+
```

The `GROUP BY` clause says to *first* group the rows according to the distinct values of the specified attribute(s) and *then* do the counting.

Statistical Data

Now let's try and find statistical data about the quizzes and tests.

```
SELECT event_id,
       MIN(score),
       MAX(score),
       AVG(score)
FROM   score
GROUP  BY event_id;
+----------+------------+------------+------------+
| event_id | MIN(score) | MAX(score) | AVG(score) |
+----------+------------+------------+------------+
|        1 |          9 |         20 |    15.1379 |
|        2 |          8 |         19 |    14.1667 |
|        3 |         60 |         97 |    78.2258 |
|        4 |          7 |         20 |    14.0370 |
|        5 |          8 |         20 |    14.1852 |
|        6 |         62 |        100 |    80.1724 |
+----------+------------+------------+------------+
```

Counting tests and quizzes

How many of the events were tests and how many were quizzes?

```
SELECT G.category, COUNT(*)
  FROM GradeEvent G
  GROUP BY G.category;

+----------+----------+
| category | COUNT(*) |
+----------+----------+
| T        |        2 |
| Q        |        4 |
+----------+----------+
```

Separating tests and quizzes

Can we get separate summary data for the quizzes and the tests? To do this we will need to do a *multi-table query* because `Score` does not know what type each event is.

```
SELECT G.category,
       AVG(S.score)
FROM   GradeEvent G,
       Score S
WHERE  G.event_id = S.event_id
GROUP  BY G.category;
+----------+--------------+
| category | AVG(S.score) |
+----------+--------------+
| T        |      79.1667 |
| Q        |      14.3894 |
+----------+--------------+
```

Separating males and females

```
SELECT G.category,
       S.gender,
       AVG(M.score)
FROM   GradeEvent G,
       Student S,
       Score M
WHERE  G.event_id = M.event_id
       AND M.student_id = S.student_id
GROUP  BY G.category,
          S.gender;

+----------+--------+--------------+
| category | gender | AVG(M.score) |
+----------+--------+--------------+
| T        | F      |      77.5862 |
| T        | M      |      80.6452 |
| Q        | F      |      14.6981 |
| Q        | M      |      14.1167 |
+----------+--------+--------------+
4 rows in set (0.00 sec)
```

Nested aggregation

Now we want to do multi-level aggregation!

```
SELECT G.category,
       S.gender,
       AVG(M.score)
FROM   GradeEvent G,
       Student S,
       Score M
WHERE  G.event_id = M.event_id
       AND M.student_id = S.student_id
GROUP  BY G.category,
          S.gender WITH ROLLUP;
```

What does ROLLUP do?

Rollup

```
+----------+--------+--------------+
| category | gender | AVG(M.score) |
+----------+--------+--------------+
| Q        | F      |      14.6981 |
| Q        | M      |      14.1167 |
| Q        | NULL   |      14.3894 |
| T        | F      |      77.5862 |
| T        | M      |      80.6452 |
| T        | NULL   |      79.1667 |
| NULL     | NULL   |      36.8555 |
+----------+--------+--------------+
```

What `ROLLUP` does

The `ROLLUP` clause generates "summaries of summaries" that are inserted at appropriate places in the table.

The `GROUP BY G.category, S.gender` clause summarises the data according to the four groups `(Q,F)`, `(Q,M)`, `(T,F)`, `(T,M)`.

Rollup causes these groups to be further grouped together into `(Q, M/F)` and `(T, M/F)` and then finally combined into a single group.

The fields where multiple values have been counted together are displayed in the result set by using `NULL` for that field.

Adding the names

At the end of semester, the lecturer needs to know how many marks each *person* got in their quizzes and tests.

```
SELECT S.name,
       G.category,
       COUNT(*),
       SUM(M.score)
FROM   GradeEvent G,
       Student S,
       Score M
WHERE  G.event_id = M.event_id
       AND S.student_id = M.student_id
GROUP  BY S.name,
          G.category WITH ROLLUP;
```

The output

```
+-----------+----------+-----------+---------------+
| name      | category | COUNT(*)  | SUM(M.score)  |
+-----------+----------+-----------+---------------+
| Abby      | Q        |         4 |            63 |
| Abby      | T        |         2 |           194 |
| Abby      | NULL     |         6 |           257 |
| Aubrey    | Q        |         4 |            58 |
| Aubrey    | T        |         2 |           137 |
| Aubrey    | NULL     |         6 |           195 |
| Avery     | Q        |         3 |            40 |
| Avery     | T        |         2 |           138 |
| Avery     | NULL     |         5 |           178 |
| Becca     | Q        |         4 |            60 |
| Becca     | T        |         2 |           176 |
```

Filtering on aggregate values

Suppose we want to find the student who got the highest average quiz mark.

```
SELECT S.name, COUNT(*), AVG(M.score)
 FROM GradeEvent G, student S, score M
 WHERE G.category = 'Q'
 AND G.event_id = M.event_id
 AND S.student_id = M.student_id
 GROUP BY S.name
 ORDER BY AVG(M.score) DESC;
```

```
+------------+-----------+---------------+
| name       | COUNT(*)  | AVG(M.score)  |
+------------+-----------+---------------+
| Megan      |         3 |       17.3333 |
| Gabrielle  |         3 |       17.0000 |
| Michael    |         4 |       16.7500 |
| Teddy      |         4 |       16.2500 |
```

Using HAVING

But the quiz-prize can only go to a student who sat *all* of the quizzes.

```
SELECT S.name, COUNT(*), AVG(M.score)
 FROM GradeEvent G, student S, score M
 WHERE G.category = 'Q'
 AND G.event_id = M.event_id
 AND S.student_id = M.student_id
 GROUP BY S.name
 HAVING COUNT(*) = 4
 ORDER BY AVG(M.score) DESC;
```

```
 +---------+----------+--------------+
| name    | COUNT(*) | AVG(M.score) |
+---------+----------+--------------+
| Michael |        4 |      16.7500 |
| Teddy   |        4 |      16.2500 |
```

Summary

The `HAVING` clause behaves exactly like a `WHERE` clause except that it operates on the *summarized* data, so the whole process is as follows:

- The named columns are extracted from the Cartesian product of all the tables listed in the `FROM` clause.
- All of these rows are then filtered according to the `WHERE` clause.
- The filtered rows are then grouped together according to the `GROUP BY` clause.
- The *aggregate* functions are applied to the rows in each *group*, forming one *summary row* per group.
- The resulting rows are then *filtered* by the `HAVING` clause.
- The filtered, aggregated rows are then *ordered* by the `ORDER BY` clause.