This lecture

This lecture starts formal coverage of SQL, in particular MySQL.

Overview

- **Data Definition Language (DDL)**
  - Creating, deleting and altering *tables* with `CREATE TABLE`, `DROP TABLE` and `ALTER TABLE`.
  - Working with *views*.
- **Data Manipulation Language (DML)**
  - Inserting, updating and deleting *rows* with `INSERT`, `UPDATE` and `DELETE`.
  - *Querying* the database with `SELECT` combining one or more tables.
  - *Summarizing* data with aggregate functions and `GROUP BY`.
- **Advanced functions**
  - Stored procedures, functions and triggers.

Another example

For these lectures we'll use another example based on Paul Dubois's book MySQL with the following specification. The database is to be used to keep student marks while taking a particular unit.

- Students have a first name, a sex and a unique student number
- *Grade Events* are either tests or quizzes and happen on a particular date
- Students *take* tests or quizzes and get a score for that grade event
Creating tables

We have already seen some examples of creating tables simply by listing the attributes and their types inside a `CREATE TABLE` statement.

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

This definition contains only features that we have seen before, but there are several others we will meet later.

Changing tables

Tables can be changed with the `ALTER TABLE` command; notice that this alters the relation `schema` and not an individual row.

```sql
CREATE TABLE grade_event (
    date DATE NOT NULL,
    category INT NOT NULL,
    event_id INT NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (event_id)
) ENGINE = InnoDB;
```

This table has been entered incorrectly because it has `INT` for the category which is either a test or a quiz.
Data Definition Language

Altering a column type

ALTER TABLE grade_event
CHANGE category
category ENUM('T','Q') NOT NULL;

The old column category is named and then an entirely new column
definition (which may include a new name) is given.

Other Alterations

There are many other things that can go after ALTER TABLE
- ALTER TABLE ADD COLUMN column-definition
- ALTER TABLE ADD INDEX...
- ALTER TABLE ADD FOREIGN KEY ...
- ALTER TABLE ADD PRIMARY KEY ...

Anything that you can ADD with ALTER TABLE, you can also DROP in
the same way.

You can also make changes that do not alter the logical structure of the
table, but have some other effect:
ALTER TABLE student ENGINE = MyISAM;

The score table

CREATE TABLE score (
  student_id INT UNSIGNED NOT NULL,
  event_id INT UNSIGNED NOT NULL,
  score INT NOT NULL,
  PRIMARY KEY (event_id, student_id),
  INDEX (student_id),
  FOREIGN KEY (event_id)
    REFERENCES grade_event (event_id),
  FOREIGN KEY (student_id)
    REFERENCES student (student_id)
) ENGINE = InnoDB;

This contains one new feature - an INDEX on the student_id field to
permit rapid searching.

Inserting Data

There are two main ways to insert data into a table
- INSERT INTO table-name VALUES ( values )
- LOAD DATA
The most fundamental database task is querying the database. For this purpose the most important statement is the `SELECT` statement, which can be extremely simple or very complicated due to its many optional parts.

```
SELECT columns
FROM tables
WHERE conditions
GROUP BY group columns
ORDER BY sort columns
HAVING more conditions
LIMIT number
```

(GF Royle 2006-8, N Spadaccini 2008)

**Zero table selections**

In MySQL you can make selections that simply compute results “on-the-fly” and therefore you do not need the `FROM` clause.

```
mysql> SELECT NOW();
+---------------------+
| NOW()              |
+---------------------+
| 2006-08-03 11:47:28 |
+---------------------+
1 row in set (0.00 sec)

mysql> SELECT POW(2,7);
+----------+
| POW(2,7) |
+----------+
| 128      |
+----------+
1 row in set (0.00 sec)
```

(GF Royle 2006-8, N Spadaccini 2008)

**Multiple table selections**

The real power, and hence complexity, of `SELECT` comes from the ability to rapidly extract data from more than one table.

A multiple table `SELECT` statement can become very complex, and (unfortunately) the syntax can often seem somewhat counterintuitive — this is largely because the lack of general programming constructs in SQL.
Combining two tables
The tables score and grade_event are related by the field event_id.

mysql> select * from grade_event;
+------------+----------+----------+
| date       | category | event_id |
+------------+----------+----------+
| 2004-09-03 | Q        | 1        |
| 2004-09-06 | Q        | 2        |
| ...        |          |          |

mysql> select * from score;
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
| 1          | 1        | 20    |
| 3          | 1        | 20    |
| 4          | 1        | 18    |
| ...        |          |       |

Connecting the tables
Suppose we want to determine the scores in the quiz (or test) held on the 9th September 2004. Conceptually this involves
- Consulting the grade_event table to find the event_id of the event of that date
- Consulting the score table to extract the rows that have this event_id

This can be performed in a single SELECT statement:

SELECT * FROM grade_event, score
WHERE grade_event.event_id = score.event_id
AND date = '2004-09-09';

Output from this query

| date       | category | event_id | student_id | event_id | score |
+------------+----------+----------+------------+----------+-------+
| 2004-09-09 | T        | 3        | 1          | 3        | 88    |
| 2004-09-09 | T        | 3        | 2          | 3        | 84    |
| 2004-09-09 | T        | 3        | 3          | 3        | 69    |
| 2004-09-09 | T        | 3        | 4          | 3        | 71    |
| 2004-09-09 | T        | 3        | 5          | 3        | 97    |
| 2004-09-09 | T        | 3        | 6          | 3        | 83    |
| 2004-09-09 | T        | 3        | 7          | 3        | 88    |
| 2004-09-09 | T        | 3        | 8          | 3        | 75    |
| 2004-09-09 | T        | 3        | 9          | 3        | 83    |
| 2004-09-09 | T        | 3        | 10         | 3        | 72    |
| 2004-09-09 | T        | 3        | 11         | 3        | 74    |
| ...        |          |          |            |          |       |

Logically speaking
We can analyse this query in parts
- SELECT * FROM grade_event, score
  On its own, this says to select every combination of rows from grade_event and score — in fact, to form the Cartesian product grade_event \times score.
- WHERE grade_event.event_id = score.event_id
  This now selects only the rows where the event_id in the two tables matches — so each row now refers to an individual score in a specific quiz/test.
- AND date = '2004-09-09'
  And finally this now picks out only the events from this date.

[The data contained in all three tables can be downloaded from the Downloads section of the CITS3240 website]
Qualifying the fields

Notice that the conditions included:

```
grade_event.event_id = score.event_id
and
date = '2004-09-09'
```

where the two occurrences of `event_id` need to be qualified with the table name because both tables contain a field of that name.

However the `date` field only occurs in one table and so it is unambiguous.

Aliasing the tables

We can also alias the table names temporarily:

```
SELECT S.student_id, S.score FROM
grade_event G, score S
WHERE
  G.event_id = S.event_id
AND
  G.date = '2004-09-09';
```

This is convenient but it also semantically critical in certain specific cases.

When both tables are the same

Suppose we want to find out if two presidents have the same birthday — basically we need to form the cross product of two copies of the same `president` table and then pick out the rows where the birthdates fall on the same day of the year.

We can’t start off with

```
SELECT * FROM president, president
WHERE ...
```

because any subsequent occurrence of a field name, like `birth` would be ambiguous.

A better way

```
SELECT P1.last_name, P2.last_name
FROM president P1, president P2
WHERE
  DAYOFYEAR(P1.birth) = DAYOFYEAR(P2.birth);
```

Now `P1` aliases the first copy of the table, while `P2` aliases the second copy of the table, and by comparing which day-of-the-year they were born on we can pick out only those pairs with the same birthday.
Whoops

| last_name | last_name |
+------------+------------+
| Washington | Washington |
| Adams | Adams |
| Jefferson | Jefferson |
| Madison | Madison |
| Monroe | Monroe |
| Adams | Adams |
| Jackson | Jackson |
| Van Buren | Van Buren |

... we forgot that when the two components of the cross product refer to the same president, then obviously their birthdays will match.

(GF Royle 2006-8, N Spadaccini 2008)

Finally...

SELECT P1.last_name, P2.last_name
FROM president P1, president P2
WHERE
  DAYOFYEAR(P1.birth) = DAYOFYEAR(P2.birth)
  AND
  P1.birth <> P2.birth;

| last_name | last_name |
+------------+------------+
| Harding | Polk |
| Polk | Harding |

2 rows in set (0.00 sec)

Really finally

We can ensure that the pair is listed only once by adding an extra condition.

SELECT P1.last_name, P2.last_name
FROM president P1, president P2
WHERE
  DAYOFYEAR(P1.birth) = DAYOFYEAR(P2.birth)
  AND
  P1.birth <> P2.birth
  AND
  P1.last_name < P2.last_name;

| last_name | last_name |
+------------+------------+
| Harding | Polk |

(GF Royle 2006-8, N Spadaccini 2008)