SQL Workshop
Session 2

Thursday 18 July 2013

General Assembly

Roadmap

- Where are we
- Review
  - Homework and material
- Conceptual path of query writing
- Concepts of database organization
- Joins
- Subqueries
- More detailed conditions

So Far:

- We have tables, we write queries, we get another table back
- Oriented to basics as we showed that
  - We can do more with WHERE, GROUP BY, ORDER BY
  - Important, useful, boring

Next: Combine Tables, More Complicated Analysis

- JOIN one table with another
  - E.g., match records from one table with records from another
- Nest queries with one another
  - Use “subqueries” to generate values used in queries, or to run particular queries for each record in the returned table

Environment Stuff

- Learn about the databases and tables available to use
- Create new databases and tables
- Load new data
- Revise existing tables, remove or revise data
- Crucial, but boring

Deeper manipulations

- More about
  - Operators (greater than, less than, IN, etc)
  - Logical expressions (AND, OR)
  - Using functions to filter, group, reduce data
- Available functions

Exercise Review

Joins

A basic query:

```
SELECT
   student_id AS Student,
   AVG(score) AS Average,
   COUNT(score) AS "# of Tests"
FROM
   sampdb.score
GROUP BY
   student_id
ORDER BY
   Average DESC
```

- Returns each student's average

Student	Average	# of Tests
1	48.0000	5
27	45.7500	4
5	42.8333	6
18	41.3333	6
17	41.2000	5
2	40.4000	5
11	39.8333	6

- But can we get the student's name rather than their id?

Yes

```
SELECT
   st.name AS          Name,
   scr.student_id AS   Id,
   AVG(scr.score) AS   Average,
   COUNT(scr.score) AS "# Tests"
FROM
   sampdb.score scr
INNER JOIN
   sampdb.student st
ON
   scr.student_id = st.student_id
GROUP BY
   scr.student_id
ORDER BY
   Average DESC
```

Name	Id	Average	# Tests
Megan	1	48.0000	5
Carter	27	45.7500	4
Abby	5	42.8333	6
Max	18	41.3333	6
Will	17	41.2000	5
Joseph	2	40.4000	5

Breaking down the query

```
SELECT
   st.name AS         Name,
   scr.student_id AS   Id,
   AVG(scr.score) AS   Average,
   COUNT(scr.score) AS "# Tests"
FROM
   sampdb.score scr
INNER JOIN
    sampdb.student st
ON
    scr.student_id = st.student_id
GROUP BY
   scr.student_id
ORDER BY
   Average DESC
```

- A join combines rows from different tables
- The "ON" clause specifies conditions for combining records
  - Here, records with the same student_id value are combined into a single record
  - The SELECT clause still specifies which fields are displayed from that combined record
- INNER JOIN specifies which table to join data from
  - The addition of "scr" and "st" in the specification of tables provides alias values for reference by the other clauses
  - Without these the query wouldn't know which table it should find a field in

Break Down Joins Still More

Two Really Simple Tables

- T1

i1	c1
1	a
2	b
3	c

- t2

i2	c2
2	c
3	b
4	a

Inner Join: All Rows Matched

- A simple inner join matches each row in one table with each row in the other
- So joining a 3 row table with a 3 row table produces a 9 row table
- Obviously we don't want all those rows

```
SELECT * FROM join_sample.t1 INNER
JOIN join_sample.t2;
```

i1	c1	i2	c2
1	a	2	c
2	b	2	c
3	c	2	c
1	a	3	b
2	b	3	b
3	c	3	b
1	a	4	a
2	b	4	a
3	c	4	a

Limiting Inner Join Results

```
SELECT
   *
FROM
   join_sample.t1

INNER JOIN
   join_sample.t2

WHERE
   join_sample.t1.i1 = join_sample.t2.i2

;
```

i1	c1	i2	c2
1	a	2	c
2	b	2	c
3	c	2	c
1	a	3	b
2	b	3	b
3	c	3	b
1	a	4	a
2	b	4	a
3	c	4	a

i1	c1	i2	c2
2	b	2	c
3	c	3	b

Left Join

- Get all rows from the “left” table, each with that row from the “right” table that matches on the specified fields

```
SELECT
   *
FROM
   join_sample.t1
LEFT JOIN
   join_sample.t2
   ON
      join_sample.t1.i1 = join_sample.t2.i2
;
```

i1	c1	i2	c2
1	a	NULL	NULL
2	b	2	c
3	c	3	b

Subqueries

Simple but clumsy

```
SELECT au_id, city

 FROM books.authors

 WHERE city IN

  (

     'New York',

     'San Francisco',

     'Hamburg',

     'Berkeley'

  );
```

- Uses the WHERE field IN [list] pattern that we have seen before
- Works, but must type four city names
  - List might be longer
  - List might be wrong
  - List might change
- How did we get those names in the first place?

Databases produce lists

```
SELECT city FROM books.publishers;
```

```
city
New York
San Francisco
Hamburg
Berkeley
```

- Now we have the right list, but we still have to type it.

Better to subquery

```
SELECT au_id, city
  FROM books.authors
  WHERE city IN
    (SELECT city FROM books.publishers);
```

- We let the database figure out the list
- If the data supporting the list changes – the query still works
- And we can expand the logic used to specify the list . . .

```
SELECT au_id, au_fname, au_lname, state
  FROM authors
  WHERE state IN
    (SELECT state
      FROM authors
      WHERE au_id = 'A04');
```

- Now we can introduce logic into our specification of the WHERE clause list

Subqueries inject query results into queries

```
SELECT au_id, city

 FROM books.authors

 WHERE city IN

  (SELECT city FROM books.publishers);
```

Exploring Data Available

Exploring databases and tables

SHOW DATABASES

- Dull but important

```
+--------------------+
| Database           |
+--------------------+
| books              |
| information_schema |
| join_sample        |
| mysql              |
| sampdb             |
| test               |
+--------------------+
```

SHOW TABLES IN books

```
+-----------------+
| Tables_in_books |
+-----------------+
| au_orders       |
| authors         |
| dups            |
| employees       |
| empsales        |
| hier            |
| publishers      |
| roadtrip        |
| royalties       |
| telephones      |
| temps           |
| time_series     |
| title_authors   |
| titles          |
+-----------------+
```

Finding fields

DESCRIBE books.author

- 'Type' tells the database how to handle data in a field
  - For example, numbers, strings and dates all sort in different ways
- Null, Key and Default set controls on values entered into the table

Field	Type	Null	Key	Default	Extra
au_id	char(3)	NO	PRI	NULL	
au_fname	varchar(15)	NO		NULL	
au_lname	varchar(15)	NO		NULL	
phone	varchar(12)	YES		NULL	
address	varchar(20)	YES		NULL	
city	varchar(15)	YES		NULL	
state	char(2)	YES		NULL	
zip	char(5)	YES		NULL	

Setting and Changing These

- We will look at how to creating and loading these things
  - These values are set at table creation
  - They can be altered, but it takes some trouble