Join-fu: The Art of SQL

Part II - Intermediate Join-Fu

Jay Pipes
Community Relations Manager
MySQL
jay@mysql.com
http://jpipes.com

These slides released under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 License
Practical examples, but meant to show techniques of SQL problem solving

- Handling hierarchical queries
  - Adjacency lists
  - Nested sets
- Exploring GIS calculations in SQL
  - Distance between two points
  - Points within a given radius
- Reporting query techniques
  - Running sums and aggregates
  - Ranking return results
a word about fear...

Don't be afraid of SQL.

Remember...

SQL is your friend.
querying hierarchical structures

- Graphs and trees don't fit the relational model well
- Common solutions tend to use either of two techniques
  - Recursion (yuck.)
  - Application layer coding (ok.)
- A good solution blends two common tree-storage models
  - Adjacency list
  - Nested sets
adjacency list model

- Very common but doesn't scale
- Easy to query for:
  - Who is my parent?
  - Who are my children?
- Difficult to query for:
  - How many levels are in my tree?
  - Who are ALL the descendants of my grandfather's brother?
adjacency list model - easy stuff

- Who is my parent?

- Who are my father's children?

- Who are my father's father's grandchildren?
adjacency list model - hard stuff

- How many levels in my hierarchy?
  - Told you. Yuck.

- Find all descendants of a specific person
  - Double yuck.

- Basic join-fu how not to do SQL?
  - Avoid cursors, iterators, etc

```sql
DELIMITER //
CREATE PROCEDURE get_max_levels()
BEGIN
  SET @lowest_parent :=
    (SELECT MAX(parent) FROM People WHERE parent IS NOT NULL);
  SET @levels := 1;

  SET @current_parent = @lowest_parent;
  WHILE @current_parent IS NOT NULL DO
    SET @current_parent :=
      (SELECT parent FROM People WHERE person_id = @current_parent);
    SET @levels := @levels + 1;
  END WHILE;

  SELECT @levels;
END //
```

```sql
DELIMITER //
CREATE PROCEDURE get_node_descendants(IN to_find INT)
BEGIN
  DROP TEMPORARY TABLE IF EXISTS child_ids;
  CREATE TEMPORARY TABLE child_ids (child_id INT UNSIGNED NOT NULL);
  ...
  WHILE @last_count_children > @new_count_children DO
    ...
    INSERT INTO child_ids
    SELECT person_id FROM new_children WHERE blah blah...;
    SET @new_count_children := (SELECT COUNT(*) FROM child_ids);
  END WHILE;

  SELECT p.* FROM People
  INNER JOIN child_ids
  ON person_id = child_id;
END //
```
nested sets model

• Uncommon because it is hard to grasp at first, but it really scales
• Easy to query for:
  – How many levels are in my tree?
  – Who are ALL the descendants of my grandfather's brother?
  – Various complex queries that would be impossible for the adjacency list model

```
CREATE TABLE People (  
  person_id INT UNSIGNED NOT NULL,  
  name VARCHAR(50) NOT NULL,  
  left_side INT UNSIGNED NOT NULL,  
  right_side INT UNSIGNED NOT NULL,  
  PRIMARY KEY (person_id),  
  INDEX (parent)  
) ENGINE=InnoDB;
```

```
mysql> SELECT * FROM People;
+-----------+-------------------+--------+
| person_id | name              | parent |
+-----------+-------------------+--------+
|         1 | Great grandfather |   NULL |
|         2 | Grandfather       |      1 |
|         3 | Great Uncle       |      1 |
|         4 | Father            |      2 |
|         5 | Uncle             |      2 |
|         6 | Me                |      4 |
|         7 | Brother           |      4 |
+-----------+-------------------+--------+
7 rows in set (0.00 sec)
```
nested sets model

- Each node in tree stores info about its location
  - Each node stores a “left” and a “right”
    - For the root node, “left” is always 1, “right” is always $2^n$, where $n$ is the number of nodes in the tree
    - For all other nodes, “right” is always equal to the “left” + $(2^n) + 1$, where $n$ is the total number of child nodes of this node
      - So...all “leaf” nodes in a tree have a “right” = “left” + 1
- Allows SQL to “walk” the tree's nodes

- OK, got all that? :)
• For the root node, “left” is always 1, “right” is always $2^n$, where $n$ is the number of nodes in the tree.

• For all other nodes, “right” is always equal to the “left” + $(2^n) + 1$, where $n$ is the total number of child nodes of this node.
so, how is this easier?

- Easy to query for:
  - How many levels are in my tree?
  - Who are ALL the descendants of my grandfather’s brother?
  - Various complex queries that would be impossible for the adjacency list model

- Efficient processing via set-based logic
  - Versus inefficient iterative/recursive model

- Basic operation is a BETWEEN predicate in a self join condition
  - Hey, you said you wanted advanced stuff…
nested list model - sets, not procedures...

- What is the depth of each node?
  - Notice the BETWEEN predicate in use

- What about the EXPLAIN output?
  - Oops
  - Add an index...

```sql
mysql> SELECT p1.person_id, p1.name, COUNT(*) AS level
    -> FROM People p1
    -> INNER JOIN People p2
    -> ON p1.left_side BETWEEN p2.left_side AND p2.right_side
    -> GROUP BY p1.person_id;
+-----------+-------------------+-------+
| person_id | name              | level |
+-----------+-------------------+-------+
|         1 | Great grandfather |     1 |
|         2 | Grandfather       |     2 |
|         3 | Great Uncle       |     3 |
|         4 | Father            |     4 |
|         5 | Uncle             |     4 |
|         6 | Me                |     3 |
|         7 | Brother           |     2 |
+-----------+-------------------+-------+
```

```
ALTER TABLE People ADD UNIQUE INDEX ix_nsm (left_side, right_side);
```
How do I find the max depth of the tree?

- If the last query shows the depth of each node...then we build on the last query

```sql
mysql> SELECT MAX(level) AS max_level FROM (  
-> SELECT p1.person_id, COUNT(*) AS level  
-> FROM People p1  
-> INNER JOIN People p2  
-> ON p1.left_side BETWEEN p2.left_side AND p2.right_side  
-> GROUP BY p1.person_id  
-> ) AS derived;

+-----------+
| max_level |
+-----------+
| 4         |
+-----------+
1 row in set (0.00 sec)
```

Use this technique when solving set-based problems

- Build on a known correct set and then intersect, union, aggregate, etc against that set
good, but could be better...

- Using covering indexes for everything
  - “Using index”
- Unfortunately, we've got a filesort
  - “Using filesort”

```sql
mysql> EXPLAIN SELECT MAX(level) AS max_level FROM (  
    -> SELECT p1.person_id, COUNT(*) AS level  
    -> FROM People p1  
    -> INNER JOIN People p2  
    -> ON p1.left_side BETWEEN p2.left_side AND p2.right_side  
    -> GROUP BY p1.person_id  
    -> ) AS derived\G
```

1. row
   - id: 1
   - select_type: PRIMARY
   - table: <derived2>
   - type: ALL
   - rows: 7

2. row
   - id: 2
   - select_type: DERIVED
   - table: p1
   - type: index
   - possible_keys: ix_nsm
   - key: ix_nsm
   - key_len: 8
   - rows: 7
   - Extra: Using index; Using temporary; Using filesort

3. row
   - id: 2
   - select_type: DERIVED
   - table: p2
   - type: index
   - possible_keys: ix_nsm
   - key: ix_nsm
   - key_len: 8
   - rows: 7
   - Extra: Using where; Using index
attacking unnecessary filesorts

```sql
mysql> EXPLAIN SELECT MAX(level) AS max_level FROM ( 
    -> SELECT p1.person_id, COUNT(*) AS level 
    -> FROM People p1 
    -> INNER JOIN People p2 
    -> ON p1.left_side BETWEEN p2.left_side AND p2.right_side 
    -> GROUP BY p1.person_id 
    -> ORDER BY NULL 
    -> ) AS derived
G
```

```
*************************** 1. row ***************************
  id: 1
  select_type: PRIMARY
  table: <derived2>
  type: ALL
  rows: 7

*************************** 2. row ***************************
  id: 2
  select_type: DERIVED
  table: p1
  type: index
  possible_keys: ix_nsm
    key: ix_nsm
    key_len: 8
    rows: 7
    Extra: Using index; Using temporary;

*************************** 3. row ***************************
  id: 2
  select_type: DERIVED
  table: p2
  type: index
  possible_keys: ix_nsm
    key: ix_nsm
    key_len: 8
    rows: 7
    Extra: Using where; Using index
```

- **GROUP BY** implicitly orders the results
- If you don't need that sort, remove it using ORDER BY NULL
finding a node's descendants

- Who are ALL my grandfather's descendants?
  - Remember the nasty recursive solution we had?

```sql
mysql> SELECT p1.name
          -> FROM People p1
          -> INNER JOIN People p2
          -> ON p1.left_side BETWEEN p2.left_side AND p2.right_side
          -> WHERE p2.person_id = @to_find
          -> AND p1.person_id <> @to_find;
+---------+
| name    |
+---------+
| Father  |
| Uncle   |
| Me      |
| Brother |
+---------+
4 rows in set (0.00 sec)
```

```sql
mysql> EXPLAIN SELECT p1.name
          -> FROM People p1
          -> INNER JOIN People p2
          -> ON p1.left_side BETWEEN p2.left_side AND p2.right_side
          -> WHERE p2.person_id = @to_find
          -> AND p1.person_id <> @to_find;
1. row
id: 1
select_type: SIMPLE
table: p2
type: const
possible_keys: PRIMARY,ix_nsm
key: PRIMARY
key_len: 4
 ref: const
rows: 1

2. row
id: 1
select_type: SIMPLE
table: p1
type: range
possible_keys: PRIMARY,ix_nsm
key: PRIMARY
key_len: 4
rows: 4
Extra: Using where
```
finding all nodes *above* a specific node

- Who are ALL my grandfather's *predecessors*?
- Look familiar to the last query?
  - What has changed?

- What about now?
summarizing trees and graphs

- Lots more we could do with trees
  - How to insert/delete/move a node in the tree
  - How to connect the tree to aggregate reporting results
  - But not right now...

- Best practice
  - Use both adjacency list and nested sets for various query types
    - Little storage overhead
    - Best of both worlds
reporting techniques

- Running aggregates
  - Without user variables
  - Running sums and averages

- Ranking of results
  - Using user variables
  - Using JOINs!
When we want to have a column which “runs” a sum during the result set

```
SELECT
    MONTHNAME(created) AS Month,
    COUNT(*) AS Added
FROM feeds
WHERE created >= '2007-01-01'
GROUP BY MONTH(created);
```

<table>
<thead>
<tr>
<th>Month</th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>February</td>
<td>1</td>
</tr>
<tr>
<td>March</td>
<td>11</td>
</tr>
<tr>
<td>April</td>
<td>8</td>
</tr>
<tr>
<td>May</td>
<td>18</td>
</tr>
<tr>
<td>June</td>
<td>3</td>
</tr>
</tbody>
</table>

6 rows in set (0.00 sec)

```
SELECT
    MONTHNAME(created) AS Month,
    COUNT(*) AS Added,
    SUM(COUNT(*)) OVER (ORDER BY MONTH(created)) AS Total
FROM feeds
WHERE created >= '2007-01-01'
GROUP BY MONTH(created);
```

<table>
<thead>
<tr>
<th>Month</th>
<th>Added</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>January</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>February</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>March</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>April</td>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td>May</td>
<td>18</td>
<td>39</td>
</tr>
<tr>
<td>June</td>
<td>3</td>
<td>42</td>
</tr>
</tbody>
</table>

6 rows in set (0.00 sec)
basic formula for running aggregates

- Join a set (table) to itself using a $\geq$ predicate
  - ON x1.key $\geq$ x2.key

- Problem, though, when we are working with *pre-aggregated* data
  - Obviously, you can't do two GROUP BYs...
replacing sets in the running aggregate formula

• Stick to the formula, but replace sets $x_1$ and $x_2$ with your pre-aggregated sets as derived tables
  – The right shows replacing $x$ with derived

```
SELECT
  x1.key
, x1.some_column
, AGGREGATE_FN(x2.some_column)
FROM x AS x1
INNER JOIN x AS x2
ON x1.key >= x2.key
GROUP BY x1.key;
```

```
SELECT
  x1.key
, x1.some_column
, AGGREGATE_FN(x2.some_column)
FROM (SELECT
  MONTH(created) AS MonthNo
, MONTHNAME(created) AS MonthName
, COUNT(*) AS Added
FROM feeds
WHERE created >= '2007-01-01'
GROUP BY MONTH(created)
) AS x1
INNER JOIN (SELECT
  MONTH(created) AS MonthNo
, MONTHNAME(created) AS MonthName
, COUNT(*) AS Added
FROM feeds
WHERE created >= '2007-01-01'
GROUP BY MONTH(created)
) AS x2
ON x1.key >= x2.key
GROUP BY x1.key;
```
finally, replace SELECT, ON and outer GROUP BY

• Replace the greyed-out area with the correct fields

```sql
SELECT
  x1.key
, x1.some_column
, AGGREGATE_FN(x2.some_column)
FROM (SELECT
  MONTH(created) AS MonthNo
, MONTHNAME(created) AS MonthName
, COUNT(*) AS Added
FROM feeds
WHERE created >= '2007-01-01'
GROUP BY MONTH(created)
) AS x1
INNER JOIN (SELECT
  MONTH(created) AS MonthNo
, MONTHNAME(created) AS MonthName
, COUNT(*) AS Added
FROM feeds
WHERE created >= '2007-01-01'
GROUP BY MONTH(created)
) AS x2
ON x1.key >= x2.key
GROUP BY x1.key;
```

```sql
SELECT
  x1.MonthNo
, x1.MonthName
, x1.Added
, SUM(x2.Added) AS RunningTotal
FROM (SELECT
  MONTH(created) AS MonthNo
, MONTHNAME(created) AS MonthName
, COUNT(*) AS Added
FROM feeds
WHERE created >= '2007-01-01'
GROUP BY MONTH(created)
) AS x1
INNER JOIN (SELECT
  MONTH(created) AS MonthNo
, MONTHNAME(created) AS MonthName
, COUNT(*) AS Added
FROM feeds
WHERE created >= '2007-01-01'
GROUP BY MONTH(created)
) AS x2
ON x1.MonthNo >= x2.MonthNo
GROUP BY x1.MonthNo;
```
and the running results...

<table>
<thead>
<tr>
<th>MonthNo</th>
<th>MonthName</th>
<th>Added</th>
<th>RunningTotal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>January</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>February</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>March</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>April</td>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>May</td>
<td>18</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>June</td>
<td>3</td>
<td>42</td>
</tr>
</tbody>
</table>

6 rows in set (0.00 sec)

- Easy enough to add running averages
  - Simply add a column for AVG(x2.Added)
- Lesson to learn: stick to a known formula, then replace formula elements with known sets of data (Keep it simple!)
ranking of results

• Using user variables
  – We set a @rank user variable and increment it for each returned result

• Very easy to do in both SQL and in your programming language code
  – But, in SQL, you can use that produced set to join with other results...
• Easy enough
  – But what about ties in the ranking?

• Notice that some of the films have identical prices, and so should be tied...
  – Go ahead and try to produce a clean way of dealing with ties using user variables...

```
mysql> SET @rank = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT film_id, LEFT(title, 30) as title, rental_rate, (@rank:= @rank + 1) as rank
FROM film
ORDER BY rental_rate DESC
LIMIT 10;
```

<table>
<thead>
<tr>
<th>film_id</th>
<th>title</th>
<th>rental_rate</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>243</td>
<td>DOORS PRESIDENT</td>
<td>7.77</td>
<td>1</td>
</tr>
<tr>
<td>93</td>
<td>BRANNIGAN SUNRISE</td>
<td>7.70</td>
<td>2</td>
</tr>
<tr>
<td>321</td>
<td>FLASH WARS</td>
<td>7.50</td>
<td>3</td>
</tr>
<tr>
<td>938</td>
<td>VELVET TERMINATOR</td>
<td>7.50</td>
<td>4</td>
</tr>
<tr>
<td>933</td>
<td>VAMPIRE WHALE</td>
<td>7.49</td>
<td>5</td>
</tr>
<tr>
<td>246</td>
<td>DOUBTFIRE LABYRINTH</td>
<td>7.45</td>
<td>6</td>
</tr>
<tr>
<td>253</td>
<td>DRIFTER COMMANDMENTS</td>
<td>7.44</td>
<td>7</td>
</tr>
<tr>
<td>676</td>
<td>PHILADELPHIA WIFE</td>
<td>7.44</td>
<td>8</td>
</tr>
<tr>
<td>961</td>
<td>WASH HEAVENLY</td>
<td>7.41</td>
<td>9</td>
</tr>
<tr>
<td>219</td>
<td>DEEP CRUSADE</td>
<td>7.40</td>
<td>10</td>
</tr>
</tbody>
</table>
```

Hmm, I have to wonder what “Deep Crusade” is about...
Again, we use a formula to compute ranked results.

Technique: use a known formulaic solution and replace formula values with known result sets.

The formula takes ties into account with the `>=` predicate in the join condition.

```
SELECT x1.key_field, x1.other_field, COUNT(*) AS rank
FROM x AS x1
INNER JOIN x AS x2
  ON x1.rank_field <= x2.rank_field
GROUP BY x1.key_field
ORDER BY x1.rank_field DESC;
```
replace variables in the formula

```
SELECT
  x1.key_field,
  x1.other_field,
  COUNT(*) AS rank
FROM x AS x1
INNER JOIN x AS x2
  ON x1.rank_field <= x2.rank_field
GROUP BY
  x1.key_field
ORDER BY
  x1.rank_field DESC
LIMIT 10;
```

```
SELECT
  x1.film_id,
  x1.title,
  x1.rental_rate,
  COUNT(*) AS rank
FROM film AS x1
INNER JOIN film AS x2
  ON x1.rental_rate <= x2.rental_rate
GROUP BY
  x1.film_id
ORDER BY
  x1.rental_rate DESC
LIMIT 10;
```

- Ties are now accounted for
- Easy to grab a “window” of the rankings
  - Just change LIMIT and OFFSET

<table>
<thead>
<tr>
<th>film_id</th>
<th>title</th>
<th>rental_rate</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>243</td>
<td>DOORS PRESIDENT</td>
<td>7.77</td>
<td>1</td>
</tr>
<tr>
<td>93</td>
<td>BRANNIGAN SUNRISE</td>
<td>7.70</td>
<td>2</td>
</tr>
<tr>
<td>938</td>
<td>VELVET TERMINATOR</td>
<td>7.50</td>
<td>4</td>
</tr>
<tr>
<td>321</td>
<td>FLASH WARS</td>
<td>7.50</td>
<td>4</td>
</tr>
<tr>
<td>933</td>
<td>VAMPIRE WHALE</td>
<td>7.49</td>
<td>5</td>
</tr>
<tr>
<td>246</td>
<td>DOUBTFIRE LABYRINTH</td>
<td>7.45</td>
<td>6</td>
</tr>
<tr>
<td>676</td>
<td>PHILADELPHIA WIFE</td>
<td>7.44</td>
<td>8</td>
</tr>
<tr>
<td>253</td>
<td>DRIFTER COMMANDMENTS</td>
<td>7.44</td>
<td>8</td>
</tr>
<tr>
<td>961</td>
<td>WASH HEAVENLY</td>
<td>7.41</td>
<td>9</td>
</tr>
<tr>
<td>219</td>
<td>DEEP CRUSADE</td>
<td>7.40</td>
<td>10</td>
</tr>
</tbody>
</table>
refining the performance...

- **EXPLAIN produces:**

<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>x2</td>
<td>ALL</td>
<td>PRIMARY</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>952</td>
<td>Using temporary; Using filesort</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>x1</td>
<td>ALL</td>
<td>PRIMARY</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>952</td>
<td>Using where</td>
</tr>
</tbody>
</table>

- And the query ran in **1.49s** (that's bad, mkay...)
- No indexes being used
  - We add an index on **film (film_id, rental_rate)**

<table>
<thead>
<tr>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>x2</td>
<td>index</td>
<td>ix_film_id</td>
<td>ix film_id_rate</td>
<td>4</td>
<td>NULL</td>
<td>967</td>
<td>Using index; Using temporary; Using filesort</td>
</tr>
<tr>
<td>x1</td>
<td>ALL</td>
<td>ix_rate_film_id</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>967</td>
<td>Using where</td>
</tr>
</tbody>
</table>

- Results: slightly better performance of **0.80s**
  - But, different GROUP and ORDER BY makes it slow
querying GIS data

- Without using the spatial extensions
  - Although you could.
- Without using stored functions
  - Although you could.
- Without using user variables
  - Although you could.
- But, heck, it's more fun this way...
  - And performs faster in a lot of cases!
GIS data basics

- The world is not *flat*
  - Duh.
  - But the MySQL spatial extensions until recently thought the world *was* flat
    - Spatial extensions prior to MySQL 5.1.something-recent used Euclidean geometry
    - Spherical calculations are different - they use Hadrian geometry which takes into account the fact that distances between longitudinal lines converge towards the poles

- GIS calculations are done in *radians*, not degrees

\[ \text{radians} = \text{degrees} \times \left( \frac{\pi}{180} \right) \]
important formulas

- **Great circle distance**
  - Between two points \((x_1, x_2)\) and \((y_1, y_2)\)
  
  \[
  d = \text{acos} \left( \sin(x_1) \times \sin(x_2) + \cos(x_1) \times \cos(x_2) \times \cos(y_2 - y_1) \right) \times r
  \]
  - Where \(r\) is the radius of the Earth (~3956 miles)

- **Haversine formula**
  - Builds on the GCD formula but adds an additional conditioning factor in order to make smaller distance calculations more accurate
  
  \[
  d = r \times \text{asin} \left( \sqrt{\sin((x_2 - y_2) / 2)^2 + \cos(x_1) \times \sin((y_2 - y_1) / 2)^2} \right) 
  \]

- Don't need extreme accuracy or don't have high-accuracy coordinate data? GCD is good enough
CREATE TABLE ZCTA (  
zcta CHAR(5) NOT NULL PRIMARY KEY  ,  
lat_degrees DECIMAL(9,6) NOT NULL  ,  
long_degrees DECIMAL(9,6) NOT NULL  ) ENGINE=MyISAM;

CREATE TABLE Store (  
store_id INT UNSIGNED NOT NULL  ,  
zipcode CHAR(5) NOT NULL  ,  
street_address VARCHAR(100) NOT NULL  ,  
PRIMARY KEY (store_id)  ,  
INDEX (zipcode)  ) ENGINE=InnoDB;

- Data from the US Census Bureau for zip code tabulation areas (ZCTAs)
  - Roughly equivalent to the zip code
  - GIS coordinates provided in degrees
  - So we convert to radians

ALTER TABLE ZCTA  
ADD COLUMN lat_radians DECIMAL(12,9) NOT NULL , ADD COLUMN long_radians DECIMAL(12,9) NOT NULL;

UPDATE ZCTA  
SET lat_radians = lat_degrees * (PI() / 180) , long_radians = long_degrees * (PI() / 180);
finding the distance between two points

• So, how far did I travel today?
  – Downtown Columbus, Ohio: 43206
  – Provo, Utah: 84601

```sql
mysql> SELECT ROUND(
    -> ACOS(SIN(orig.lat_radians) * SIN(dest.lat_radians)
    -> + COS(orig.lat_radians) * COS(dest.lat_radians)
    -> * COS(dest.long_radians - orig.long_radians)) * 3956
    -> , 2) AS distance
    -> FROM ZCTA orig, ZCTA dest
    -> WHERE orig.zcta = '43206'
    -> AND dest.zcta = '84601';

+----------+
| distance |
+----------+
| 1509.46  |
+----------+
1 row in set (0.00 sec)

+----+-------------+-------+-------+---------------+---------+---------+-------+------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+
|  1 | SIMPLE      | orig  | const | PRIMARY       | PRIMARY | 6       | const | 1    |
|  1 | SIMPLE      | dest  | const | PRIMARY       | PRIMARY | 6       | const | 1    |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+
radius searches

- Imagine drawing a circle on a piece of paper using a contractor...

```sql
mysql> SELECT orig.zcta, dest.zcta, ROUND(
    -> ACOS(SIN(orig.lat_radians) * SIN(dest.lat_radians)
    -> + COS(orig.lat_radians) * COS(dest.lat_radians)
    -> * COS(dest.long_radians - orig.long_radians)) * 3956
    -> , 2) AS distance
    -> FROM ZCTA orig, ZCTA dest
    -> WHERE orig.zcta = '43206';
```

| 43206 | 00976 | 1801.56 |
| 43206 | 00979 | 1796.26 |
| 43206 | 00982 | 1798.26 |
| 43206 | 00983 | 1798.53 |
| 43206 | 00985 | 1801.85 |
| 43206 | 00987 | 1801.48 |

- Think of the SQL above as a producing a giant graph that looks like a Koosh® ball
radius searches

• If we remove the WHERE clause from below, what do we get?

```sql
mysql> SELECT orig.zcta, dest.zcta, ROUND(
    -> ACOS(SIN(orig.lat_radians) * SIN(dest.lat_radians)
    -> + COS(orig.lat_radians) * COS(dest.lat_radians)
    -> * COS(dest.long_radians - orig.long_radians)) * 3956
    -> , 2) AS distance
    -> FROM ZCTA orig, ZCTA dest
    -> WHERE orig.zcta = '43206';
```

• A cartesian product of course...
  
  – But a *useful* cartesian product of distances between all points in the US
  
  – Don't try to do this just yet

  • $32,038^2 = 1,026,433,444$ records

• Can we make use of this result?
radius searches - expanding our distance formula

- Get all zips within 35 miles of “43206” (Downtown, Columbus, Ohio)

```sql
mysql> SELECT dest.zcta, ROUND(ACOS(SIN(orig.lat_radians) * SIN(dest.lat_radians) + COS(orig.lat_radians) * COS(dest.lat_radians) * COS(dest.long_radians - orig.long_radians)) * 3956, 9) AS "Distance"
FROM ZCTA orig, ZCTA dest
WHERE orig.zcta = '43206'
AND ACOS(SIN(orig.lat_radians) * SIN(dest.lat_radians) + COS(orig.lat_radians) * COS(dest.lat_radians) * COS(dest.long_radians - orig.long_radians)) * 3956 <= 35
ORDER BY Distance;
```

<table>
<thead>
<tr>
<th>zcta</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>43206</td>
<td>0.000000000</td>
</tr>
<tr>
<td>43205</td>
<td>1.181999017</td>
</tr>
<tr>
<td>43215</td>
<td>1.886507824</td>
</tr>
<tr>
<td>43149</td>
<td>34.895068055</td>
</tr>
</tbody>
</table>

108 rows in set (0.10 sec)

---

<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orig</td>
<td>const</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>6</td>
<td>const</td>
<td>1</td>
<td>Using filesort</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>dest</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>32038</td>
<td>Using where</td>
</tr>
</tbody>
</table>
tie in radius with our store locations

- Find all HomeDepot stores within 35 miles of me

```sql
mysql> SELECT
  ->  LEFT(street_address, 30) AS address
  -> , zipcode
  -> , ROUND(ACOS(SIN(orig.lat_rad) * SIN(dest.lat_rad))
  -> + COS(orig.lat_rad) * COS(dest.lat_rad)
  -> * COS(dest.long_rad - orig.long_rad)) * 3956, 9) AS "Distance"
  -> FROM ZCTA orig, ZCTA dest
  -> INNER JOIN Store s
  -> ON dest.zcta = s.zipcode
  -> WHERE orig.zcta = '43206'
  -> AND ACOS(SIN(orig.lat_rad) * SIN(dest.lat_rad))
  -> + COS(orig.lat_rad) * COS(dest.lat_rad)
  -> * COS(dest.long_rad - orig.long_rad)) * 3956 <= 35
  -> ORDER BY Distance;

+--------------------------+---------+--------------+
| address                  | zipcode | Distance     |
+--------------------------+---------+--------------+
| Grove City #6954 - 1680 String | 43123   | 6.611091045  |
| West Broad #3819 100 South Gr | 43228   | 7.554534005  |
| East Columbus #3828 5200 Hami | 43230   | 8.524457137  |
| Cleveland Ave #3811 6333 Clev | 43229   | 9.726193043  |
| Hilliard #3872 4101 Trueman B | 43026   | 10.304498649 |
| Canal Winchester #3885 6035 G | 43110   | 11.039675381 |
| Sawmill #3831 5858 Sawmill Rd | 43017   | 13.764803511 |
| Westerville #3825 6017 Maxtow | 43082   | 14.534428656 |
| Orange Township #3836 8704 Ow | 43065   | 15.554864931 |
| Marysville #3889 880 Coleman | 43040   | 29.52885252  |
| Newark #3887 1330 N 21st St  | 43055   | 32.063414509 |
+--------------------------+---------+--------------+
11 rows in set (0.00 sec)