The Legend of Drunken Query Master

The Apprentice’s Journey

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
before we start

• Who am I?
  – Just some dude who works at MySQL (eh...Sun)
  – Oh, I co-wrote a book on MySQL
  – Active PHP/MySQL community member
  – Other than that, semi-normal geek, married, 2 dogs, 2 cats, blah blah

• This talk is about understanding and tuning SQL applications
notes on this presentation

- Got a quick question?
  - Just ask it.

- Got a longer question?
  - Wait until the break or when the speaker forgets what he is talking about.

- Pictures of cute and cuddly animals are used frequently throughout presentation as a blatant attempt to improve your review of the speaker
  - If you hate cute and cuddly animals, you may want to leave now.
drunken query master says...

“...thou shalt not fight with the database, for it is thy friend”

can't we all just get along?

http://www.ringelkater.de/ringel_witzig/cat_dog_mouse.jpg
your friend, the database...

- Recognize the strengths and also the weaknesses of your database
- No database is perfect -- deal with it, you're not perfect either
- Think of both big things and small things
  - **BIG**: Architecture, surrounding servers, caching
  - **SMALL**: SQL coding, join rewrites, server config
expect the unexpected

• Some big things produce small results
  – Hardware changes can produce smaller improvements than many expect

• Some small things produce humungonormous results
  – Rewriting a poorly-constructed query can improve performance or scalability more than you might expect
system architecture of MySQL

Clients → Query Cache → "Packaging" → Net I/O → Parser → Optimizer

Pluggable Storage Engine API

- MyISAM
- InnoDB
- MEMORY
- Falcon
- Archive
- PBXT
- SolidDB
- Cluster (Ndb)
keys to MySQL system architecture

- Understand storage engine abilities and weaknesses
- Understand how the query cache and important buffers works
- Understand optimizer's limitations
- Understand what should and should not be done at the application level
- If you understand the above, you'll start to see the database as a friend and not an enemy
drunken query master says...

“...poor skills in schema-kido shall lead to crushing defeat”

Image: http://www.cold-moon.com/images/Motivators/GMs/defeat.jpg

DEFEAT

Sometimes you just should have seen it coming.
the schema

- Basic foundation of performance
- Everything else depends on it
- Choose your data types wisely
- Conquer the schema through partitioning

The Leaning Tower of Pisa from Wikipedia:

“Although intended to stand vertically, the tower began leaning to the southeast soon after the onset of construction in 1173 due to a poorly laid foundation and loose substrate that has allowed the foundation to shift direction.”
smaller, smaller, smaller

The Pygmy Marmoset
world's smallest monkey

This picture is a cheap stunt intended to induce kind feelings for the presenter.

Oh, and I totally want one of these guys for a pet.

The more records you can fit into a single page of memory/disk, the faster your seeks and scans will be

- Do you really need that **BIGINT**?
- Use **INT UNSIGNED** for IPv4 addresses
- Use **VARCHAR** carefully
  - Converted to **CHAR** when used in a temporary table
- Use **TEXT** sparingly
  - Consider separate tables
- Use **BLOBs** very sparingly
  - Use the filesystem for what it was intended
handling IPv4 addresses

```sql
CREATE TABLE Sessions (
    session_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    ip_address INT UNSIGNED NOT NULL // Compare to CHAR(15)...
    , session_data TEXT NOT NULL,
    PRIMARY KEY (session_id),
    INDEX (ip_address)
) ENGINE=InnoDB;

// Insert a new dummy record
INSERT INTO Sessions VALUES (NULL, INET_ATON('192.168.0.2'), 'some session data');

// Find all sessions coming from a local subnet
SELECT
    session_id,
    ip_address as ip_raw,
    INET_NTOA(ip_address) as ip,
    session_data
FROM Sessions
WHERE ip_address BETWEEN INET_ATON('192.168.0.1') AND INET_ATON('192.168.0.255');

mysql> SELECT session_id, ip_address as ip_raw, INET_NTOA(ip_address) as ip, session_data
    -> FROM Sessions
    -> WHERE ip_address BETWEEN INET_ATON('192.168.0.1') AND INET_ATON('192.168.0.255');

+------------+------------+-------------+-------------------+
| session_id | ip_raw     | ip          | session_data      |
+------------+------------+-------------+-------------------+
|          1 | 3232235522 | 192.168.0.2 | some session data |
+------------+------------+-------------+-------------------+
```

// Insert INTO Session VALUES (NULL, 3232235522, 'some session data');

```sql
// Find all sessions coming from a local subnet
SELECT
    session_id,
    ip_address as ip_raw,
    INET_NTOA(ip_address) as ip,
    session_data
FROM Sessions
WHERE ip_address BETWEEN 3232235521 AND 3232235775;
```
SETs and ENUMs

- Drunken query master is not a big fan of SET and ENUM
- Sign of poor schema design
- Changing the definition of either will require a full rebuild of the table
- Search functions like FIND_IN_SET() are inefficient compared to index operations on a join

Rumour has it that the Mandelbrot Set will be a full-fledged column type in MySQL 9.1, making for some very interesting application uses in database graphics processing...
how much storage space is consumed?

- With this definition, how many bytes will the “a” column consume per row?

```sql
CREATE TABLE t1 (  
a INT(1) UNSIGNED NOT NULL );
```

- The number in parentheses is the \textit{ZEROFILL} argument, not the storage space

- INT takes 4 bytes of space
  - Regardless of the UNSIGNED or NOT NULL
drunken query master says...

“...table-kee-do shall show thee the Path of (De)Normalization”

fifth-degree black-belt in join-fu -->

Edgar F. Codd
from Wikipedia:

“...while working for IBM, invented the relational model for database management, the theoretical basis for relational databases.”
taking normalization way too far

Hmm......

ToDate?
divide et impera

- Vertical partitioning
  - Split tables with many columns into multiple tables

- Horizontal partitioning
  - Split table with many rows into multiple tables

- Partitioning in MySQL 5.1 is transparent horizontal partitioning within the DB...

...and it's got issues at the moment.

Niccolò Machiavelli
*The Art of War*, (1519-1520):

“A Captain ought, among all the other actions of his, endeavor with every art to divide the forces of the enemy, either by making him suspicious of his men in whom he trusted, or by giving him cause that he has to separate his forces, and, because of this, become weaker.”
vertical partitioning

- Mixing frequently and infrequently accessed attributes in a single table?
- Space in buffer pool at a premium?
  - Splitting the table allows main records to consume the buffer pages without the extra data taking up space in memory
- Need **FULLTEXT** on your text columns?
drunken query master says...

“...thou shalt employ table-keeper in order to avoid the Fury of the Query Cache”

Think kittens get angry? Wait until you see what the query cache can do.
the MySQL query cache

- You must understand your application's read/write patterns
- Internal query cache design is a compromise between CPU usage and read performance
- Stores the MYSQL_RESULT of a SELECT along with a hash of the SELECT SQL statement
- Any modification to any table involved in the SELECT invalidates the stored result
- Write applications to be aware of the query cache
  - Use SELECT SQL_NO_CACHE
vertical partitioning ... continued

- Mixing static attributes with frequently updated fields in a single table?
  - Thrashing occurs with query cache. Each time an update occurs on any record in the table, all queries referencing the table are invalidated in the query cache

- Doing \texttt{COUNT(*)} with no \texttt{WHERE} on an indexed field on an InnoDB table?
  - Complications with versioning make full table counts very slow
“...thou shalt not be afraid of SQL, for it is thy strongest weapon”
coding like a join-fu master

- Be consistent (for crying out loud)
- Use ANSI SQL coding style
- Stop thinking in terms of iterators, for loops, while loops, etc
- Instead, think in terms of sets
- Break complex SQL statements (or business requests) into smaller, manageable chunks

Did you know?
from Wikipedia:

Join-fu is a close cousin to Jun Fan Gung Fu, the method of martial arts Bruce Lee began teaching in 1959.

OK, not really.
SQL coding consistency

• Tabs and spacing
• Upper and lower case
• Keywords, function names
• Some columns aliased, some not

SELECT
  a.first_name, a.last_name, COUNT(*) as num_rentals
FROM actor a
INNER JOIN film f
  ON a.actor_id = fa.actor_id
GROUP BY a.actor_id
ORDER BY num_rentals DESC, a.last_name, a.first_name
LIMIT 10;

vs.

select first_name, a.last_name,
  count(*) AS num_rentals
FROM actor a join film f on a.actor_id = fa.actor_id
  group by a.actor_id order by
num_rentals DESC, a.last_name, a.first_name
LIMIT 10;

• Consider your teammates
• Like your programming code, SQL is meant to be read, not written

Nothing pisses off the query master like inconsistent SQL code!
join-fu guidelines

- Always try variations on a theme
- Beware of join hints
  - Can get “out of date”
- Just because it *can* be done in a single SQL statement doesn't mean it should
- Always test and benchmark your solutions
  - Use `http_load` (simple and effective for web stuff)

See, even bears practice join-fu.
ANSI vs. Theta SQL coding style

ANSI STYLE
Explicitly declare JOIN conditions using the ON clause

```
SELECT  
a.first_name, a.last_name, COUNT(*) as num_rentals  
FROM actor a 
    INNER JOIN film f ON a.actor_id = fa.actor_id  
    INNER JOIN film_actor fa ON fa.film_id = f.film_id  
    INNER JOIN inventory I ON f.film_id = i.film_id  
    INNER JOIN rental r ON r.inventory_id = i.inventory_id  
GROUP BY a.actor_id  
ORDER BY num_rentals DESC, a.last_name, a.first_name  
LIMIT 10;
```

Theta STYLE
Implicitly declare JOIN conditions in the WHERE clause

```
SELECT  
a.first_name, a.last_name, COUNT(*) as num_rentals  
FROM actor a, film f, film_actor fa, inventory i, rental r  
WHERE a.actor_id = fa.actor_id  
AND fa.film_id = f.film_id  
AND f.film_id = i.film_id  
AND r.inventory_id = i.inventory_id  
GROUP BY a.actor_id  
ORDER BY num_rentals DESC, a.last_name, a.first_name  
LIMIT 10;
```
why ANSI style's join-fu kicks Theta style's ass

- MySQL only supports the INNER and CROSS join for the Theta style
  - But, MySQL supports the INNER, CROSS, LEFT, RIGHT, and NATURAL joins of the ANSI style
  - Mixing and matching both styles can lead to hard-to-read SQL code

- It is supremely easy to miss a join condition with Theta style
  - especially when joining many tables together
  - Leaving off a join condition in the WHERE clause will lead to a cartesian product (not a good thing!)
drunken query master says...

“...Without the strength of explain-jitsu, thou shall perish in the Meadow of Misunderstanding”

chased by the Evil Army of Correlated Subqueries through the Meadow of Misunderstanding -->

EXPLAIN Basics

- Provides the execution plan chosen by the MySQL optimizer for a specific SELECT statement
- Simply append the word EXPLAIN to the beginning of your SELECT statement
- Each row in output represents a set of information used in the SELECT
  - A real schema table
  - A virtual table (derived table) or temporary table
  - A subquery in SELECT or WHERE
  - A unioned set
EXPLAIN columns

- **select_type** - type of “set” the data in this row contains
- **table** - alias (or full table name if no alias) of the table or derived table from which the data in this set comes
- **type** - “access strategy” used to grab the data in this set
- **possible_keys** - keys available to optimizer for query
- **keys** - keys chosen by the optimizer
- **rows** - *estimate* of the number of rows in this set
- **Extra** - information the optimizer chooses to give you
- **ref** - shows the column used in join relations
mysql> EXPLAIN SELECT f.film_id, f.title, c.name
    > FROM film f INNER JOIN film_category fc
    > ON f.film_id=fc.film_id INNER JOIN category c
    > ON fc.category_id=c.category_id WHERE f.title LIKE 'T%' \G

An estimate of rows in this set

The available indexes, and the one(s) chosen

A covering index is used
Example #1 - the const access type

```
EXPLAIN SELECT * FROM rental WHERE rental_id = 13\G
*************************** 1. row ***************************
    id: 1
  select_type: SIMPLE
     table: rental
      type: const
possible_keys: PRIMARY
       key: PRIMARY
   key_len: 4
      ref: const
     rows: 1
    Extra: 1 row in set (0.00 sec)
```
Constants in the optimizer

- A field indexed with a *unique non-nullable* key
- The access strategy of *system* is related to *const* and refers to when a table with only a single row is referenced in the SELECT
- Can be propagated across joined columns
Example #2 - constant propagation

```sql
EXPLAIN SELECT r.*, c.first_name, c.last_name
FROM rental r INNER JOIN customer c
ON r.customer_id = c.customer_id WHERE r.rental_id = 13\G
```

```
*************************** 1. row ***************************
  id: 1
  select_type: SIMPLE
  table: r
  type: const
  possible_keys: PRIMARY,idx_fk_customer_id
  key: PRIMARY
  key_len: 4
  ref: const
  rows: 1
  Extra:

*************************** 2. row ***************************
  id: 1
  select_type: SIMPLE
  table: c
  type: const
  possible_keys: PRIMARY
  key: PRIMARY
  key_len: 2
  ref: const /* Here is where the propagation occurs...*/
  rows: 1
  Extra:
```

2 rows in set (0.00 sec)
Example #3 - the range access type

```
SELECT * FROM rental
WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-16'\G
```

```
*************************** 1. row ***************************
   id: 1
  select_type: SIMPLE
    table: rental
    type: range
possible_keys: rental_date
  key: rental_date
key_len: 8
  ref: NULL
rows: 364
Extra: Using where
1 row in set (0.00 sec)
```
Considerations with range accesses

- Index must be available on the field operated upon by a range operator
- If too many records are estimated to be returned by the condition, the range optimization won't be used
  - index or full table scan will be used instead
- The indexed field must not be operated on by a function call! (Important for all indexing)
The scan vs. seek dilemma

- A **seek** operation, generally speaking, jumps into a random place -- either on disk or in memory -- to fetch the data needed.
  - Repeat for each piece of data needed from disk or memory
- A **scan** operation, on the other hand, will jump to the start of a chunk of data, and sequentially read data -- either from disk or from memory -- until the end of the chunk of data
- For large amounts of data, scan operations tend to be more efficient than multiple seek operations
Example #4 - Full table scan

```
EXPLAIN SELECT * FROM rental
WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-21'

*************** 1. row ***************
  id: 1
  select_type: SIMPLE
  table: rental
  type: ALL
  possible_keys: rental_date /* larger range forces scan choice */
    key: NULL
    key_len: NULL
    ref: NULL
    rows: 16298
  Extra: Using where
1 row in set (0.00 sec)
```
Why full table scans pop up

- No WHERE condition (duh.)
- No index on any field in WHERE condition
- Poor selectivity on an indexed field
- Too many records meet WHERE condition
- < MySQL 5.0 and using OR in a WHERE clause
- Using SELECT * FROM
Example #5 - Full index scan

```sql
EXPLAIN SELECT rental_id, rental_date FROM rental
```

```
*************** 1. row ***************
  id: 1
  select_type: SIMPLE
  table: rental
  type: index
  possible_keys: NULL
  key: rental_date
  key_len: 13
  ref: NULL
  rows: 16325
  Extra: Using index
1 row in set (0.00 sec)
```

**CAUTION!**

Extra="Using index" is **NOT** the same as type="index"
Example #6 - eq_ref strategy

```
EXPLAIN SELECT r.*, c.first_name, c.last_name
FROM rental r INNER JOIN customer c ON r.customer_id = c.customer_id
WHERE r.rental_date BETWEEN '2005-06-14' AND '2005-06-16'
```

```
*************************** 1. row ***************************
   id: 1
   select_type: SIMPLE
   table: r
     type: range
   possible_keys: idx_fk_customer_id,rental_date
     key: rental_date
   key_len: 8
   ref: NULL
   rows: 364
   Extra: Using where
```

```
*************************** 2. row ***************************
   id: 1
   select_type: SIMPLE
   table: c
     type: eq_ref
   possible_keys: PRIMARY
     key: PRIMARY
   key_len: 2
     ref: sakila.r.customer_id
   rows: 1
   Extra: 
```

2 rows in set (0.00 sec)
When eq_ref pops up

• Joining two sets on a field where
  – One side has unique, non-nullable index
  – Other side has at least a non-nullable index

• In example #6, an eq_ref access strategy was chosen because a unique, non-nullable index is available on customer.customer_id and an index is available on the rental.customer_id field
Nested loops join algorithm

• For each record in outermost set
  – Fetch a record from the next set via a join column condition
  – Repeat until done with outermost set

• Main algorithm in optimizer
  – Main work in 5.1+ is in the area of subquery optimization and additional join algorithms like semi- and merge joins
Example #7 - ref strategy

EXPLAIN SELECT * FROM rental WHERE rental_id IN (10,11,12) AND rental_date = '2006-02-01' \G

*************************** 1. row ***************************

id: 1
select_type: SIMPLE
table: rental
type: ref
possible_keys: PRIMARY,rental_date
key: rental_date
key_len: 8
ref: const
rows: 1
Extra: Using where
1 row in set (0.02 sec)
OR conditions and the index merge

- Index merge best thing to happen in optimizer for 5.0
- Allows optimizer to use more than one index to satisfy a join condition
  - Prior to MySQL 5.0, only one index
  - In case of OR conditions in a WHERE, MySQL <5.0 would use a *full table scan*
Example #8 - index_merge strategy

EXPLAIN SELECT * FROM rental
WHERE rental_id IN (10,11,12)
OR rental_date = '2006-02-01' \G
*************************** 1. row ***************************
    id: 1
   select_type: SIMPLE
    table: rental
     type: index_merge
possible_keys: PRIMARY,rental_date
    key: rental_date,PRIMARY
key_len: 8,4
    ref: NULL
   rows: 4
Extra: Using sort_union(rental_date,PRIMARY); Using where
1 row in set (0.02 sec)
tired? break time...

http://jimburrussdesign.com/comics/images/news_pics/passed_out_corgi.jpg
ok, now that you've had a quick snack...
drunken query master says...

“...thou shall befriend the Index, for it is a Master of Joint fu and will protect thee from the Ravages of the Table Scan”

best coffee table. ever. -->

http://technabob.com/blog/wp-content/uploads/2008/05/nes_coffee_table.jpg
indexes - your schema's phone book

- Speed up SELECTs, but slow down modifications
- Ensure indexes on columns used in WHERE, ON, GROUP BY clauses
- Always ensure JOIN conditions are indexed (and have identical data types)
- Be careful of the column order
- Look for covering indexes
  - Occurs when all fields in one table needed by a SELECT are available in an index record

The Yellow Pages
from Wikipedia:

"The name and concept of "Yellow Pages" came about in 1883, when a printer in Cheyenne, Wyoming working on a regular telephone directory ran out of white paper and used yellow paper instead"
selectivity - the key to good, er...keys

- **Selectivity**
  - % of distinct values in a column
  - \( S = \frac{d}{n} \)
  - Unique/primary always 1.0

- **If column has a low selectivity**
  - It may still be put in a multi-column index
    - As a prefix?
    - As a suffix?
    - Depends on the application
SELECT t.TABLE_SCHEMA AS `db`, t.TABLE_NAME AS `table`, s.INDEX_NAME AS `index name`, s.COLUMN_NAME AS `field name`, s.SEQ_IN_INDEX AS `seq in index`, s2.max_columns AS `# cols`, s.CARDINALITY AS `card`, t.TABLE_ROWS AS `est rows`, ROUND(((s.CARDINALITY / IFNULL(t.TABLE_ROWS, 0.01)) * 100), 2) AS `sel %` FROM INFORMATION_SCHEMA.STATISTICS s INNER JOIN INFORMATION_SCHEMA.TABLES t ON s.TABLE_SCHEMA = t.TABLE_SCHEMA AND s.TABLE_NAME = t.TABLE_NAME INNER JOIN (SELECT TABLE_SCHEMA, TABLE_NAME, INDEX_NAME, MAX(SEQ_IN_INDEX) AS max_columns FROM INFORMATION_SCHEMA.STATISTICS WHERE TABLE_SCHEMA != 'mysql' GROUP BY TABLE_SCHEMA, TABLE_NAME, INDEX_NAME ) AS s2 ON s.TABLE_SCHEMA = s2.TABLE_SCHEMA AND s.TABLE_NAME = s2.TABLE_NAME AND s.INDEX_NAME = s2.INDEX_NAME WHERE t.TABLE_SCHEMA != 'mysql' /* Filter out the mysql system DB */ AND t.TABLE_ROWS > 10 /* Only tables with some rows */ AND s.CARDINALITY IS NOT NULL /* Need at least one non-NULL value in the field */ AND (s.CARDINALITY / IFNULL(t.TABLE_ROWS, 0.01)) < 1.00 /* unique indexes are perfect anyway */) ORDER BY `sel %`, s.TABLE_SCHEMA, s.TABLE_NAME /* DESC for best non-unique indexes */ LIMIT 10;

<table>
<thead>
<tr>
<th>TABLE_SCHEMA</th>
<th>TABLE_NAME</th>
<th>INDEX_NAME</th>
<th>COLUMN_NAME</th>
<th>SEQ_IN_INDEX</th>
<th>COLS_IN_INDEX</th>
<th>CARD</th>
<th>ROWS</th>
<th>SEL %</th>
</tr>
</thead>
<tbody>
<tr>
<td>worklog</td>
<td>amendments</td>
<td>text</td>
<td>text</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>33794</td>
<td>0.00</td>
</tr>
<tr>
<td>planetmysql</td>
<td>entries</td>
<td>categories</td>
<td>categories</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4171</td>
<td>0.02</td>
</tr>
<tr>
<td>planetmysql</td>
<td>entries</td>
<td>categories</td>
<td>categories</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4171</td>
<td>0.02</td>
</tr>
<tr>
<td>planetmysql</td>
<td>entries</td>
<td>categories</td>
<td>categories</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4171</td>
<td>0.02</td>
</tr>
<tr>
<td>sakila</td>
<td>inventory</td>
<td>idx_store_id_film_id</td>
<td>idx_store_id_film_id</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4673</td>
<td>0.02</td>
</tr>
<tr>
<td>sakila</td>
<td>rental</td>
<td>title</td>
<td>title</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>16291</td>
<td>0.02</td>
</tr>
<tr>
<td>worklog</td>
<td>tasks</td>
<td>title</td>
<td>description</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3567</td>
<td>0.03</td>
</tr>
<tr>
<td>worklog</td>
<td>tasks</td>
<td>title</td>
<td>staff_id</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>3567</td>
<td>0.03</td>
</tr>
<tr>
<td>sakila</td>
<td>payment</td>
<td>idx_fk_staff_id</td>
<td>idx_fk_staff_id</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>15422</td>
<td>0.04</td>
</tr>
<tr>
<td>mysqlforge</td>
<td>mw_recentchanges</td>
<td>rc_ip</td>
<td>rc_ip</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>996</td>
<td>0.20</td>
</tr>
</tbody>
</table>

indexed columns and functions don't mix

```
mysql> EXPLAIN SELECT * FROM film WHERE title LIKE 'Tr%

*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: film
type: range
possible_keys: idx_title
key: idx_title
key_len: 767
ref: NULL
rows: 15
Extra: Using where
```

• A fast range access strategy is chosen by the optimizer, and the index on title is used to winnow the query results down

```
mysql> EXPLAIN SELECT * FROM film WHERE LEFT(title,2) = 'Tr'

*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: film
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 951
Extra: Using where
```

• A slow full table scan (the ALL access strategy) is used because a function (LEFT) is operating on the title column
solving multiple issues in a SELECT query

```
SELECT * FROM Orders WHERE TO_DAYS(CURRENT_DATE()) - TO_DAYS(order_created) <= 7;
```

- First, we are operating on an indexed column (order_created) with a function - let's fix that:

```
SELECT * FROM Orders WHERE order_created >= CURRENT_DATE() - INTERVAL 7 DAY;
```

- Although we rewrote the WHERE expression to remove the operating function, we still have a non-deterministic function in the statement, which eliminates this query from being placed in the query cache - let's fix that:

```
SELECT * FROM Orders WHERE order_created >= '2008-01-11' - INTERVAL 7 DAY;
```

- We replaced the function with a constant (probably using our application programming language). However, we are specifying SELECT * instead of the actual fields we need from the table.

- What if there is a TEXT field in Orders called order_memo that we don't need to see? Well, having it included in the result means a larger result set which may not fit into the query cache and may force a disk-based temporary table

```
SELECT order_id, customer_id, order_total, order_created
FROM Orders WHERE order_created >= '2008-01-11' - INTERVAL 7 DAY;
```
“...join-fu is thy best defense against the Evil Army of Correlated Subqueries”

general in the evil army -->
set-wise problem solving

“Show the last payment information for each customer”

CREATE TABLE `payment` (  `payment_id` smallint(5) unsigned NOT NULL auto_increment,  `customer_id` smallint(5) unsigned NOT NULL,  `staff_id` tinyint(3) unsigned NOT NULL,  `rental_id` int(11) default NULL,  `amount` decimal(5,2) NOT NULL,  `payment_date` datetime NOT NULL,  `last_update` timestamp NOT NULL ... on update CURRENT_TIMESTAMP,  PRIMARY KEY (`payment_id`),  KEY `idx_fk_staff_id` (`staff_id`),  KEY `idx_fk_customer_id` (`customer_id`),  KEY `fk_payment_rental` (`rental_id`),  CONSTRAINT `fk_payment_rental` FOREIGN KEY (`rental_id`) REFERENCES `rental` (`rental_id`),  CONSTRAINT `fk_payment_customer` FOREIGN KEY (`customer_id`) REFERENCES `customer` (`customer_id`),  CONSTRAINT `fk_payment_staff` FOREIGN KEY (`staff_id`) REFERENCES `staff` (`staff_id`) ) ENGINE=InnoDB DEFAULT CHARSET=utf8
thinking in terms of **foreach** loops...

OK, *for each* customer, find the maximum date the payment was made and get that payment record(s)

```sql
mysql> EXPLAIN SELECT
       -> p.*
       -> FROM payment p
       -> WHERE p.payment_date =
       -> ( SELECT MAX(payment_date)
       -> FROM payment
       -> WHERE customer_id=p.customer_id
       -> )
```

```
*************************** 1. row ***************************
 id: 1
 select_type: PRIMARY
 table: p
 type: ALL
 rows: 16567
 Extra: Using where

*************************** 2. row ***************************
 id: 2
 select_type: DEPENDENT SUBQUERY
 table: payment
 type: ref
 possible_keys: idx_fk_customer_id
 key: idx_fk_customer_id
 key_len: 2
 ref: sakila.p.customer_id
 rows: 15
 2 rows in set (0.00 sec)
```

- A correlated subquery in the **WHERE** clause is used
- It will be re-executed *for each* row in the primary table (payment)
- Produces 623 rows in an average of **1.03s**
what about adding an index?

Will adding an index on (customer_id, payment_date) make a difference?

```sql
dranken@localhost> EXPLAIN SELECT
-> p.*
-> FROM payment p
-> WHERE p.payment_date =
-> ( SELECT MAX(payment_date)
-> FROM payment
-> WHERE customer_id=p.customer_id
-> )
```

```
*************************** 1. row ***************************
id: 1
select_type: PRIMARY
table: p
type: ALL
rows: 16567
Extra: Using where
```

```
*************************** 2. row ***************************
id: 2
select_type: DEPENDENT SUBQUERY
table: payment
type: ref
possible_keys: idx_fk_customer_id
key: idx_fk_customer_id
key_len: 2
ref: sakila.p.customer_id
rows: 15
```

`2 rows in set (0.00 sec)`

```sql
dranken@localhost> EXPLAIN SELECT
-> p.*
-> FROM payment p
-> WHERE p.payment_date =
-> ( SELECT MAX(payment_date)
-> FROM payment
-> WHERE customer_id=p.customer_id
-> )
```

```
*************************** 1. row ***************************
id: 1
select_type: PRIMARY
table: p
type: ALL
rows: 15485
Extra: Using where
```

```
*************************** 2. row ***************************
id: 2
select_type: DEPENDENT SUBQUERY
table: payment
type: ref
possible_keys: idx_fk_customer_id,ix_customer_paydate
key: ix_customer_paydate
key_len: 2
ref: sakila.p.customer_id
rows: 14
Extra: Using index
```

`2 rows in set (0.00 sec)`

- Produces 623 rows in an average of 1.03s
- Produces 623 rows in an average of 0.45s
thinking in terms of sets...

OK, I have one set of last payments dates and another set containing payment information (so, how do I join these sets?)

```
mysql> EXPLAIN SELECT
    -> p.*
    -> FROM (
        -> SELECT customer_id, MAX(payment_date) as last_order
        -> FROM payment
        -> GROUP BY customer_id
        -> ) AS last_orders
    -> INNER JOIN
        -> payment p
    -> ON p.customer_id = last_orders.customer_id
    -> AND p.payment_date = last_orders.last_order
G
```

- **A derived table**, or subquery in the `FROM` clause, is used.
- The derived table represents a set: last payment dates of customers.
- Produces 623 rows in an average of 0.03s.
drunken query master says...

“...join-fu shall assist you in your N:M relationships”

...but it won't help your other relationships

working with “mapping” or N:M tables

CREATE TABLE Project {
  project_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  name VARCHAR(50) NOT NULL,
  url TEXT NOT NULL,
  PRIMARY KEY (project_id)
} ENGINE=MyISAM;

CREATE TABLE Tag2Project {
  tag INT UNSIGNED NOT NULL,
  project INT UNSIGNED NOT NULL,
  PRIMARY KEY (tag, project),
  INDEX rv_primary (project, tag)
} ENGINE=MyISAM;

CREATE TABLE Tags {
  tag_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  tag_text VARCHAR(50) NOT NULL,
  PRIMARY KEY (tag_id),
  INDEX (tag_text)
} ENGINE=MyISAM;

- The next few slides will walk through examples of querying across the above relationship
  - dealing with OR conditions
  - dealing with AND conditions
dealing with OR conditions

Grab all project names which are tagged with “mysql” OR “php”

```sql
mysql> SELECT p.name FROM Project p
   -> INNER JOIN Tag2Project t2p
   -> ON p.project_id = t2p.project
   -> INNER JOIN Tag t
   -> ON t2p.tag = t.tag_id
   -> WHERE t.tag_text IN ('mysql','php');
```

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpMyAdmin</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>MySQL Stored Procedures Auto Generator</td>
</tr>
</tbody>
</table>

90 rows in set (0.05 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>t</td>
<td>range</td>
<td>PRIMARY,uix_tag_text</td>
<td>uix_tag_text</td>
<td>52</td>
<td>NULL</td>
<td>2</td>
<td>Using where</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>t2p</td>
<td>ref</td>
<td>PRIMARY,rv_primary</td>
<td>PRIMARY</td>
<td>4</td>
<td>t.tag_id</td>
<td>10</td>
<td>Using index</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>p</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>t2p.project</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

3 rows in set (0.00 sec)

• Note the order in which the optimizer chose to join the tables is exactly the opposite of how we wrote our SELECT
dealing with AND conditions

Grab all project names which are tagged with “storage engine” AND “plugin”

- A little more complex, let’s grab the project names which match both the “mysql” tag and the “php” tag
- Here is perhaps the most common method - using a HAVING COUNT(*) against a GROUP BY on the relationship table
- EXPLAIN on next page...

```mysql
SELECT p.name FROM Project p
INNER JOIN (SELECT t2p.project FROM Tag2Project t2p
INNER JOIN Tag t ON t2p.tag = t.tag_id
ON t2p.tag = t.tag_id
WHERE t.tag_text IN ('plugin', 'storage engine')
GROUP BY t2p.project
HAVING COUNT(*) = 2
) AS projects_having_all_tags
ON p.project_id = projects_having_all_tags.project;
```

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic data revision</td>
</tr>
<tr>
<td>memcache storage engine for MySQL</td>
</tr>
</tbody>
</table>

2 rows in set (0.01 sec)
The EXPLAIN plan shows the execution plan using a derived table containing the project IDs having records in the Tag2Project table with both the “plugin” and “storage engine” tags.

Note that a filesort is needed on the Tag table rows since we use the index on tag_text but need a sorted list of tag_id values to use in the join.

```
*************************** 1. row ***************************
  id: 1
  select_type: PRIMARY
  table: <derived2>
  type: ALL
  rows: 2

*************************** 2. row ***************************
  id: 1
  select_type: PRIMARY
  table: p
  type: eq_ref
  possible_keys: PRIMARY
  key: PRIMARY
  key_len: 4
  ref: projects_having_all_tags.project
  rows: 1

  Extra: Using where; Using index; Using temporary; Using filesort

*************************** 3. row ***************************
  id: 2
  select_type: DERIVED
  table: t
  type: range
  possible_keys: PRIMARY,uix_tag_text
  key: uix_tag_text
  key_len: 52
  rows: 2

  Extra: Using where; Using index; Using temporary; Using filesort

*************************** 4. row ***************************
  id: 2
  select_type: DERIVED
  table: t2p
  type: ref
  possible_keys: PRIMARY
  key: PRIMARY
  key_len: 4
  ref: mysqlforge.t.tag_id
  rows: 1
  Extra: Using index
4 rows in set (0.00 sec)
```
removing the filesort using CROSS JOIN

- We can use a CROSS JOIN technique to remove the filesort
  - We winnow down two copies of the Tag table (t1 and t2) by supplying constants in the WHERE condition
- This means no need for a sorted list of tag IDs since we already have the two tag IDs available from the CROSS JOINs...so no more filesort

```
mysql> EXPLAIN SELECT p.name
    -> FROM Project p
    -> CROSS JOIN Tag t1
    -> CROSS JOIN Tag t2
    -> INNER JOIN Tag2Project t2p
    -> ON p.project_id = t2p.project
    -> AND t2p.tag = t1.tag_id
    -> INNER JOIN Tag2Project t2p2
    -> ON t2p.project = t2p2.project
    -> AND t2p2.tag = t2.tag_id
    -> WHERE t1.tag_text = "plugin"
    -> AND t2.tag_text = "storage engine";
```

<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>t1</td>
<td>const</td>
<td>PRIMARY,uix_tag_text</td>
<td>uix_tag_text</td>
<td>52</td>
<td>const</td>
<td>1</td>
<td>Using index</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>t2</td>
<td>const</td>
<td>PRIMARY,uix_tag_text</td>
<td>uix_tag_text</td>
<td>52</td>
<td>const</td>
<td>1</td>
<td>Using index</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>t2p</td>
<td>ref</td>
<td>PRIMARY,rv_primary</td>
<td>PRIMARY</td>
<td>8</td>
<td>const,mysqlforge.t2p.project</td>
<td>1</td>
<td>Using index</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>t2p2</td>
<td>eq_ref</td>
<td>PRIMARY,rv_primary</td>
<td>PRIMARY</td>
<td>4</td>
<td>const,mysqlforge.t2p2.project</td>
<td>1</td>
<td>Using index</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>p</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>mysqlforge.t2p2.project</td>
<td>1</td>
<td>Using where</td>
</tr>
</tbody>
</table>

5 rows in set (0.00 sec)
another technique for dealing with ANDs

- Do two separate queries - one which grabs tag_id values based on the tag text and another which does a self-join after the application has the tag_id values in memory

**Benefit #1**
- If we assume the Tag2Project table is updated 10X more than the Tag table is updated, the first query on Tag will be cached more effectively in the query cache

**Benefit #2**
- The EXPLAIN on the self-join query is much better than the HAVING COUNT(*) derived table solution
understanding LEFT-join-fu

- Get the tag phrases which are not related to any project
- Get the tag phrases which are not related to any project OR the tag phrase is related to project #75
- Get the tag phrases not related to project #75
LEFT join-fu: starting simple...the NOT EXISTS

- Get the tag phrases which are not related to any project

- LEFT JOIN ... WHERE x IS NULL

- WHERE x IS NOT NULL would yield tag phrases that are related to a project

  - But, then, you'd want to use an INNER JOIN
LEFT join-fu: a little harder

Get the tag phrases which are not related to any project OR the tag phrase is related to project #75

No more NOT EXISTS optimization :(

But, isn't this essentially a UNION?
LEFT join-fu: a UNION returns us to optimization

```sql
mysql> EXPLAIN SELECT
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project IS NULL
-> GROUP BY t.tag_text
-> UNION ALL
-> SELECT
-> FROM Tag t
-> INNER JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project = 75

id: 1
select_type: PRIMARY
table: t
type: index
key: uix_tag_text
key_len: 52
rows: 1126
Extra: Using index

id: 1
select_type: PRIMARY
table: t2p
type: ref
key: PRIMARY
key_len: 4
ref: mysqlforge.t.tag_id
rows: 1
Extra: Using where; Using index; Not exists

id: 2
select_type: UNION
possible_keys: PRIMARY,rv_primary
table: t2p
type: ref
key: rv_primary
key_len: 4
ref: mysqlforge.t.tag_id
rows: 1
Extra: Using index

id: NULL
select_type: UNION RESULT
table: <union1,2>
5 rows in set (0.00 sec)
```

```sql
mysql> SELECT
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project IS NULL
-> GROUP BY t.tag_text
-> UNION ALL
-> SELECT
-> FROM Tag t
-> INNER JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project = 75;

+--------------------------------------+
<table>
<thead>
<tr>
<th>tag_text</th>
</tr>
</thead>
</table>
+--------------------------------------+
184 rows in set (0.00 sec)
```
LEFT join-fu: the trickiest part...

Get the tag phrases which are not related to project #75

Shown to the left is the most common mistake made with LEFT JOINs

The problem is where the filter on project_id is done...
LEFT join-fu: the trickiest part...fixed

- Filters on the LEFT joined set must be placed in the ON clause
- Filter is applied before the LEFT JOIN and NOT EXISTS is evaluated, resulting in fewer rows in the evaluation, and better performance
Practical examples, but meant to show techniques of SQL problem solving

- Handling hierarchical queries
  - Adjacency lists
  - Nested sets
- Reporting query techniques
  - Running sums and aggregates
  - Ranking return results
drunken query master says...

“...join-fu and the Nested Sets Model shall shall deliver thee from the Adjacency List Model”
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?

```sql
CREATE TABLE People (
    person_id INT UNSIGNED NOT NULL,
    name VARCHAR(50) NOT NULL,
    parent INT UNSIGNED 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)
```
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 //
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
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
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...

```
mysql> SELECT p1.person_id, p1.name, COUNT(*) AS depth
    -> FROM People p1
    -> INNER JOIN People p2
    -> ON p1.left_side BETWEEN p2.left_side AND p2.right_side
    -> GROUP BY p1.person_id;
+-----------+-------------------+-------+
<table>
<thead>
<tr>
<th>person_id</th>
<th>name</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Great grandfather</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Grandfather</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Great Uncle</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Father</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Uncle</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Me</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>Brother</td>
<td>4</td>
</tr>
</tbody>
</table>
+-----------+-------------------+-------+
```

```
ALTER TABLE People ADD UNIQUE INDEX ix_nsm (left_side, right_side);
```
“...thou shalt build queries based on results you already know are correct”
find the max depth of the whole tree

• How do I find the max depth of the tree?
  – If the last query showed 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”
### attacking unnecessary filesorts

```
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
```

<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>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>&lt;derived2&gt;</td>
<td>ALL</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DERIVED</td>
<td>p1</td>
<td>index</td>
<td>ix_nsm</td>
<td>ix_nsm</td>
<td>8</td>
<td>7</td>
<td>Using index; Using temporary;</td>
</tr>
<tr>
<td>3</td>
<td>DERIVED</td>
<td>p2</td>
<td>index</td>
<td>ix_nsm</td>
<td>ix_nsm</td>
<td>8</td>
<td>7</td>
<td>Using where; Using index</td>
</tr>
</tbody>
</table>

- **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;
```

```plaintext
+---------+
| 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;
```

```plaintext
*************************** 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
Extra: Using where
```

```plaintext
*************************** 2. row ***************************
id: 1
select_type: SIMPLE
table: p1
type: range
possible_keys: PRIMARY,ix_nsm
key: PRIMARY
key_len: 4
rows: 4
```
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?

```sql
mysql> SELECT p2.name
    -> FROM People p1
    -> INNER JOIN People p2
    -> ON p1.left_side
    -> BETWEEN p2.left_side AND p2.right_side
    -> WHERE p1.person_id = @to_find
    -> AND p2.person_id <> @to_find;
+-------------------+
| name              |
+-------------------+
| Great grandfather |
+-------------------+
1 row in set (0.00 sec)
```

```sql
SELECT p2.name
FROM People p1
INNER JOIN People p2
ON p1.left_side
BETWEEN p2.left_side AND p2.right_side
WHERE p1.person_id = @to_find
AND p2.person_id <> @to_find;
```
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
drunken query master says...

“...thou shalt study the practice of set-based formula replacement”
formula replacement

- Take a formula you know works, and replace the variables with known sets
- Reduces errors significantly
- Forces you to think in terms of sets, instead of those darn FOR loops
- Examples:
  - Running aggregates
  - Ranking of results
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,

```sql
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>
<tr>
<td>----------</td>
<td>-------</td>
</tr>
</tbody>
</table>

6 rows in set (0.00 sec)

To get a running total, you can use a window function like this:

```sql
SELECT
    MONTHNAME(created) AS Month,
    COUNT(*) AS Added,
    COUNT(*) OVER () 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 >= predicate
  - ON x1.key >= 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

```sql
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.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!)
• 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...
ranking with user variables

- 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...

```sql
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>
+---------+----------------------+-------------+------+
10 rows in set (0.00 sec)
```

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 \( \geq \) predicate in the join condition.

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

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

- **PlanetMySQL**
  - 300+ writers on MySQL topics
  - [http://planetmysql.com](http://planetmysql.com)

- **MySQL Forge**
  - Code snippets, project listings, wiki, worklog
  - [http://forge.mysql.org](http://forge.mysql.org)

Baron Schwartz
http://xaprb.com

MySQL performance guru and co-author of High Performance MySQL, 2nd Edition (O'Reilly, 2008)

“xarpb” is Baron spelled on a Dvorak keyboard...