

Join-fu: The Art of SQL

Jay Pipes
Community Relations Manager
MySQL

jay@mysql.com
<http://jpipes.com>

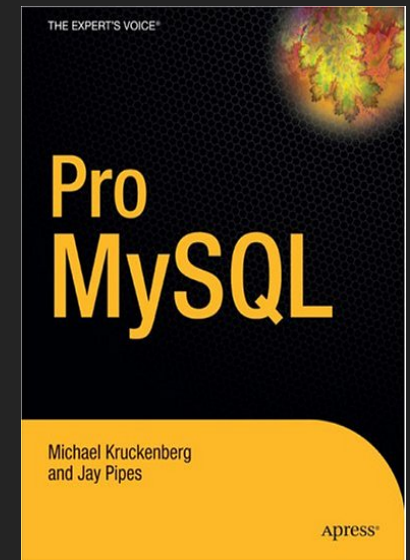


These slides released under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 License



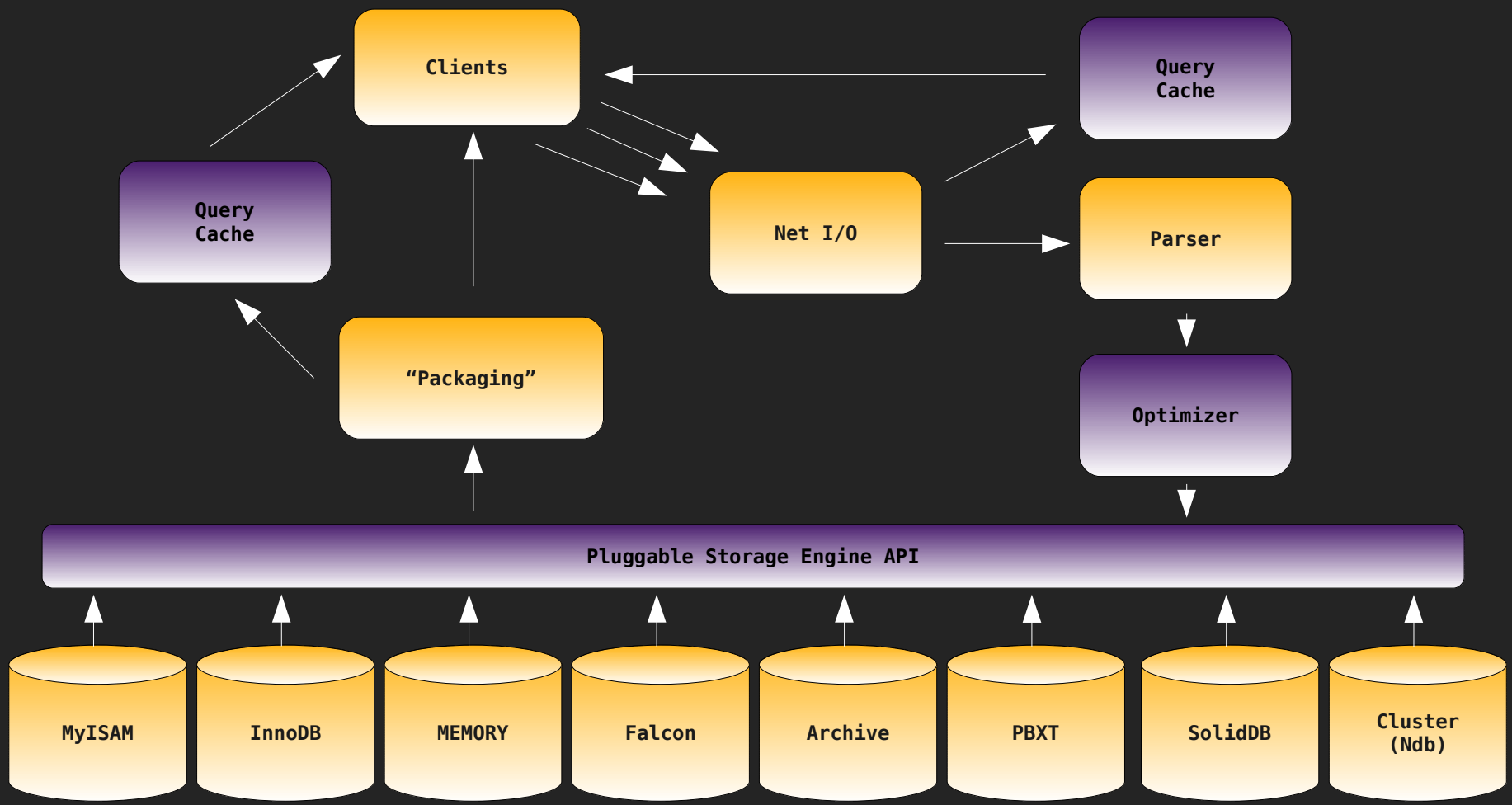
before we start

- 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 how to code your app to get the best performance out of your (My)SQL





system architecture of MySQL





the schema

- Basic foundation of performance
- Normalize first, de-normalize later
- Smaller, smaller, smaller
- Divide and conquer
- Understand benefits and disadvantages of different storage engines

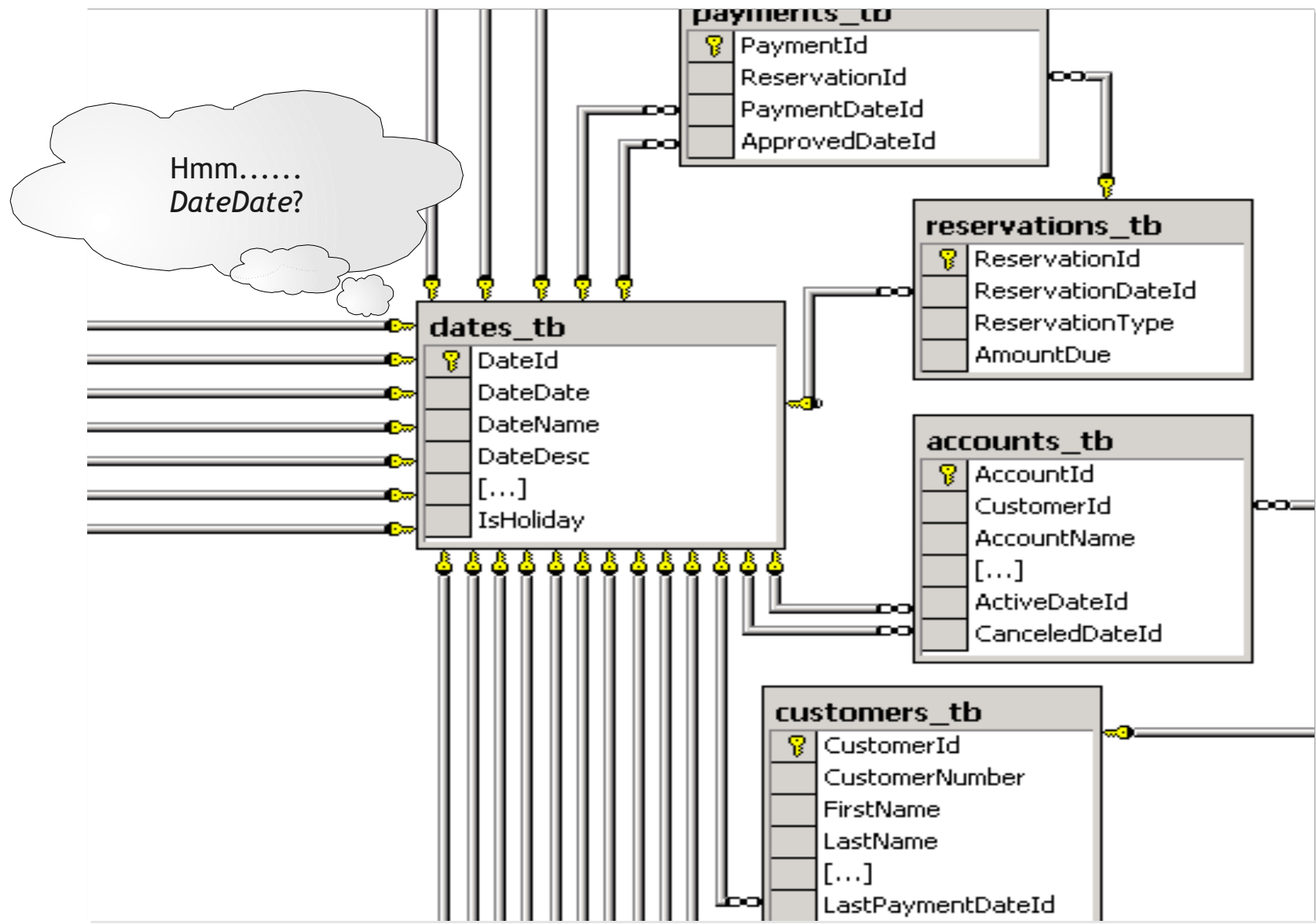


The Leaning Tower of Pisa
from Wikipedia:

“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.”



taking normalization way too far



<http://thedailywtf.com/forums/thread/75982.aspx>



smaller, smaller, smaller



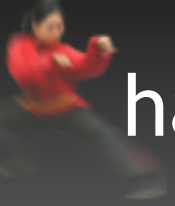
The Pygmy Marmoset
world's smallest monkey

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

```
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');
```

```
INSERT INTO Session VALUES (NULL, 3232235522, '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');
```

```
WHERE ip_address BETWEEN 3232235521 AND 3232235775
```

```
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



how much storage space is consumed?

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

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

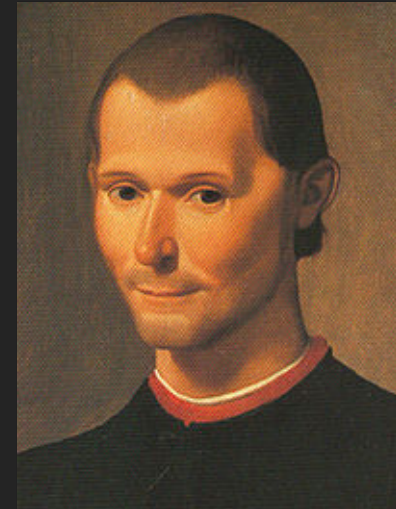
- The number in parentheses is the *ZEROFILL* argument, not the storage space
- INT takes 4 bytes of space
 - Regardless of the UNSIGNED or NOT NULL



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.**”

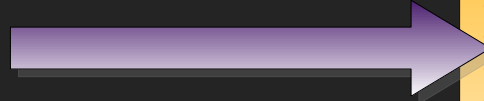


vertical partitioning

```
CREATE TABLE Users (  
  user_id INT NOT NULL AUTO_INCREMENT  
  , email VARCHAR(80) NOT NULL  
  , display_name VARCHAR(50) NOT NULL  
  , password CHAR(41) NOT NULL  
  , first_name VARCHAR(25) NOT NULL  
  , last_name VARCHAR(25) NOT NULL  
  , address VARCHAR(80) NOT NULL  
  , city VARCHAR(30) NOT NULL  
  , province CHAR(2) NOT NULL  
  , postcode CHAR(7) NOT NULL  
  , interests TEXT NULL  
  , bio TEXT NULL  
  , signature TEXT NULL  
  , skills TEXT NULL  
  , PRIMARY KEY (user_id)  
  , UNIQUE INDEX (email)  
  ) ENGINE=InnoDB;
```



```
CREATE TABLE Users (  
  user_id INT NOT NULL AUTO_INCREMENT  
  , email VARCHAR(80) NOT NULL  
  , display_name VARCHAR(50) NOT NULL  
  , password CHAR(41) NOT NULL  
  , PRIMARY KEY (user_id)  
  , UNIQUE INDEX (email)  
  ) ENGINE=InnoDB;
```

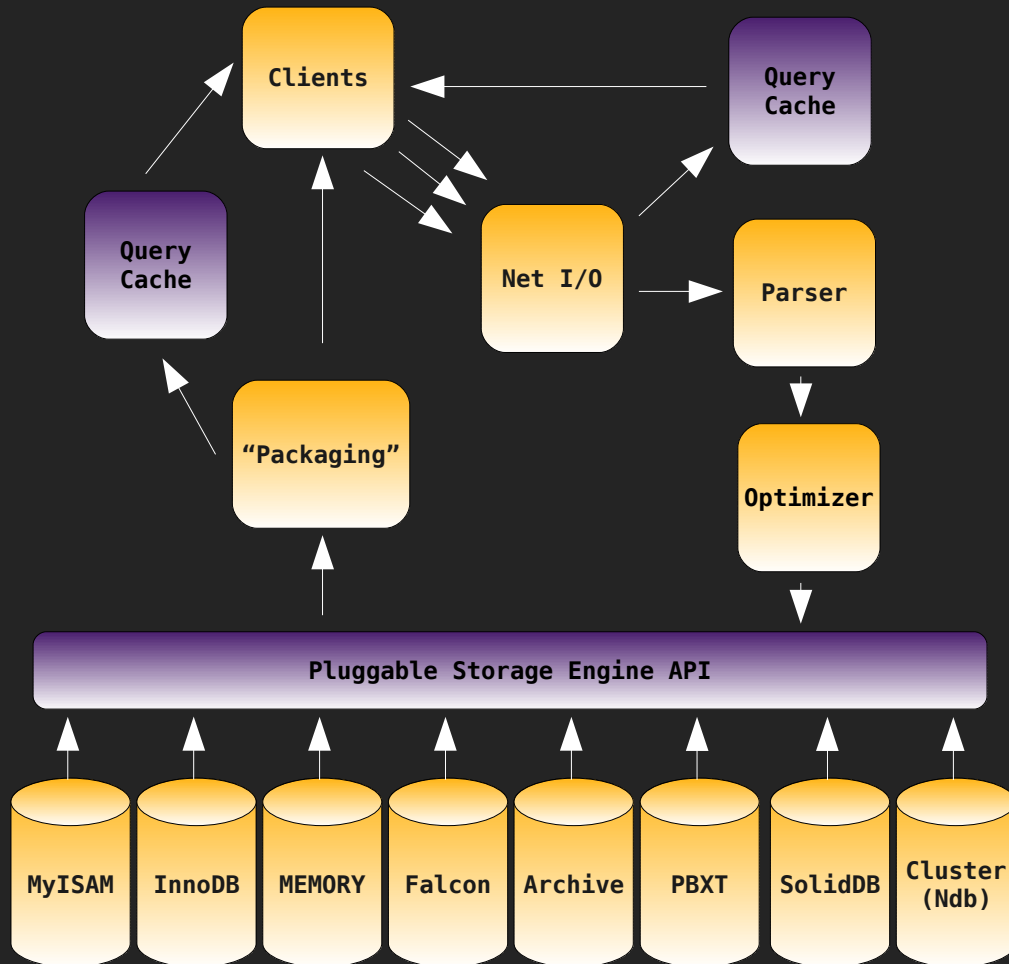


```
CREATE TABLE UserExtra (  
  user_id INT NOT NULL  
  , first_name VARCHAR(25) NOT NULL  
  , last_name VARCHAR(25) NOT NULL  
  , address VARCHAR(80) NOT NULL  
  , city VARCHAR(30) NOT NULL  
  , province CHAR(2) NOT NULL  
  , postcode CHAR(7) NOT NULL  
  , interests TEXT NULL  
  , bio TEXT NULL  
  , signature TEXT NULL  
  , skills TEXT NULL  
  , PRIMARY KEY (user_id)  
  , FULLTEXT KEY (interests, skills)  
  ) ENGINE=MyISAM;
```

- 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?



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

```
CREATE TABLE Products (  
  product_id INT NOT NULL  
  , name VARCHAR(80) NOT NULL  
  , unit_cost DECIMAL(7,2) NOT NULL  
  , description TEXT NULL  
  , image_path TEXT NULL  
  , num_views INT UNSIGNED NOT NULL  
  , num_in_stock INT UNSIGNED NOT NULL  
  , num_on_order INT UNSIGNED NOT NULL  
  , PRIMARY KEY (product_id)  
  , INDEX (name(20))  
 ) ENGINE=InnoDB;  
  
// Getting a simple COUNT of products  
// easy on MyISAM, terrible on InnoDB  
SELECT COUNT(*)  
FROM Products;
```

```
CREATE TABLE Products (  
  product_id INT NOT NULL  
  , name VARCHAR(80) NOT NULL  
  , unit_cost DECIMAL(7,2) NOT NULL  
  , description TEXT NULL  
  , image_path TEXT NULL  
  , PRIMARY KEY (product_id)  
  , INDEX (name(20))  
 ) ENGINE=InnoDB;
```

```
CREATE TABLE ProductCounts (  
  product_id INT NOT NULL  
  , num_views INT UNSIGNED NOT NULL  
  , num_in_stock INT UNSIGNED NOT NULL  
  , num_on_order INT UNSIGNED NOT NULL  
  , PRIMARY KEY (product_id)  
 ) ENGINE=InnoDB;
```

```
CREATE TABLE TableCounts (  
  total_products INT UNSIGNED NOT NULL  
 ) ENGINE=MEMORY;
```

- 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 **COUNT (*)** with no **WHERE** on an indexed field on an InnoDB table?
 - Complications with versioning make full table counts very slow

Don't be afraid
of SQL!

It (usually)
doesn't bite.



FEAR

It'll make you shit your pants.



coding like a join-fu master



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.

- Be consistent (for crying out loud)
- Use ANSI SQL coding style
- Do not think 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



join-fu guidelines



See, even bears practice join-fu.

- 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)

SQL coding consistency

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

Inconsistent SQL code makes baby kittens cry.



```
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



ANSI vs. Theta SQL coding style

```
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;
```

ANSI STYLE

Explicitly declare JOIN conditions using the ON clause

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 by accident in the WHERE clause will lead to a cartesian product



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



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 propogation

```
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 propogation 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'\G
*****
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

Example #5 - Full index scan

```
EXPLAIN SELECT rental_id, rental_date FROM rental\G
***** 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"



indexed columns and functions don't mix

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



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)

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

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

```
mysql> EXPLAIN SELECT
-> p.*
-> FROM payment p
-> WHERE p.payment_date =
-> ( SELECT MAX(payment_date)
-> FROM payment
-> WHERE customer_id=p.customer_id
-> )\G
***** 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
***** 1. row *****
      id: 1
select_type: PRIMARY
      table: <derived2>
       type: ALL
      rows: 599
***** 2. row *****
      id: 1
select_type: PRIMARY
      table: p
       type: ref
possible_keys: idx_fk_customer_id,idx_customer_paydate
      key: idx_customer_paydate
     key_len: 10
      ref: last_orders.customer_id,last_orders.last_order
      rows: 1
***** 3. row *****
      id: 2
select_type: DERIVED
      table: payment
       type: range
      key: ix_customer_paydate
     key_len: 2
      rows: 1107
      Extra: Using index for group-by
3 rows in set (0.02 sec)
```

- 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**



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 Tags (  
  tag_id INT UNSIGNED NOT NULL AUTO_INCREMENT  
  , tag_text VARCHAR(50) NOT NULL  
  , PRIMARY KEY (tag_id)  
  , INDEX (tag_text)  
  ) 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;
```

- 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”

```
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');
+-----+
| name |
+-----+
| phpMyAdmin |
| ... |
| MySQL Stored Procedures Auto Generator |
+-----+
90 rows in set (0.05 sec)
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t | range | PRIMARY,uix_tag_text | uix_tag_text | 52 | NULL | 2 | Using where |
| 1 | SIMPLE | t2p | ref | PRIMARY,rv_primary | PRIMARY | 4 | t.tag_id | 10 | Using index |
| 1 | SIMPLE | p | eq_ref | PRIMARY | PRIMARY | 4 | t2p.project | 1 | |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
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
-> 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;
+-----+-----+
| name                                     |
+-----+-----+
| Automatic data revision                 |
| memcache storage engine for MySQL      |
+-----+-----+
2 rows in set (0.01 sec)
```



the dang filesort

- 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
***** 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";
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	const	PRIMARY, uix_tag_text	uix_tag_text	52	const	1	Using index
1	SIMPLE	t2	const	PRIMARY, uix_tag_text	uix_tag_text	52	const	1	Using index
1	SIMPLE	t2p	ref	PRIMARY, rv_primary	PRIMARY	4	const	9	Using index
1	SIMPLE	t2p2	eq_ref	PRIMARY, rv_primary	PRIMARY	8	const, mysqlforge.t2p.project	1	Using index
1	SIMPLE	p	eq_ref	PRIMARY	PRIMARY	4	mysqlforge.t2p2.project	1	Using where

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

```
mysql> SELECT t.tag_id FROM Tag t
  > WHERE tag_text IN ("plugin","storage engine");
+-----+
| tag_id |
+-----+
|    173 |
|    259 |
+-----+
2 rows in set (0.00 sec)
```

```
mysql> SELECT p.name FROM Project p
-> INNER JOIN Tag2Project t2p
-> ON p.project_id = t2p.project
-> AND t2p.tag = 173
-> INNER JOIN Tag2Project t2p2
-> ON t2p.project = t2p2.project
-> AND t2p2.tag = 259;
+-----+
| name |
+-----+
| Automatic data revision |
| memcache storage engine for MySQL |
+-----+
2 rows in set (0.00 sec)
```

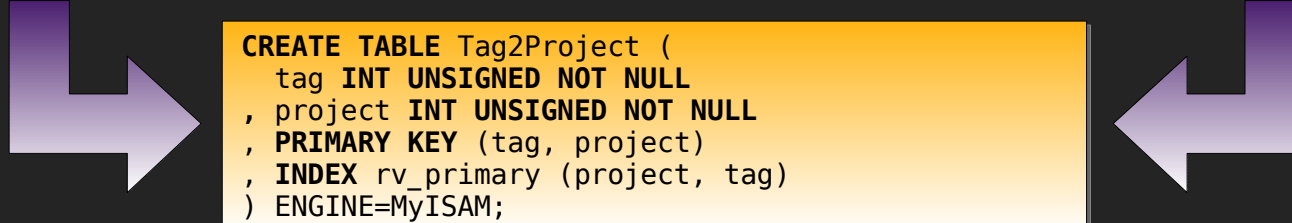
```
mysql> EXPLAIN SELECT p.name FROM Project p
-> INNER JOIN Tag2Project t2p
-> ON p.project_id = t2p.project
-> AND t2p.tag = 173
-> INNER JOIN Tag2Project t2p2
-> ON t2p.project = t2p2.project
-> AND t2p2.tag = 259;
+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+
| 1 | SIMPLE | t2p | ref | PRIMARY,rv_primary | PRIMARY | 4 | const | 9 | Using index |
| 1 | SIMPLE | t2p2 | eq_ref | PRIMARY,rv_primary | PRIMARY | 8 | const,mysqlforge.t2p.project | 1 | Using index |
| 1 | SIMPLE | p | eq_ref | PRIMARY | PRIMARY | 4 | mysqlforge.t2p2.project | 1 | Using where |
+-----+
3 rows in set (0.00 sec)
```



understanding LEFT-join-fu

```
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 Tags (  
  tag_id INT UNSIGNED NOT NULL AUTO_INCREMENT  
  , tag_text VARCHAR(50) NOT NULL  
  , PRIMARY KEY (tag_id)  
  , INDEX (tag_text)  
) 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;
```

- 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

```
mysql> EXPLAIN SELECT
-> t.tag_text
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project IS NULL
-> GROUP BY t.tag_text\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t
         type: index
possible_keys: NULL
          key: uix_tag_text
         key_len: 52
          rows: 1126
        Extra: Using index
***** 2. row *****
      id: 1
  select_type: SIMPLE
        table: t2p
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
         key_len: 4
          ref: mysqlforge.t.tag_id
          rows: 1
        Extra: Using where; Using index; Not exists
2 rows in set (0.00 sec)

mysql> SELECT
-> t.tag_text
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project IS NULL
-> GROUP BY t.tag_text;
+-----+
| tag_text          |
+-----+
<snip>
+-----+
153 rows in set (0.01 sec)
```

- 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

```
mysql> EXPLAIN SELECT
-> t.tag_text
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project IS NULL
-> OR t2p.project = 75
-> GROUP BY t.tag_text\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: t
      type: index
      key: uix_tag_text
      key_len: 52
      ref: NULL
      rows: 1126
      Extra: Using index
***** 2. row *****
      id: 1
select_type: SIMPLE
      table: t2p
      type: ref
possible_keys: PRIMARY
      key: PRIMARY
      key_len: 4
      ref: mysqlforge.t.tag_id
      rows: 1
      Extra: Using where; Using index
2 rows in set (0.00 sec)

mysql> SELECT
-> t.tag_text
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project IS NULL
-> OR t2p.project = 75
-> GROUP BY t.tag_text;
+-----+
| tag_text |
+-----+
<snip>
+-----+
184 rows in set (0.00 sec)
```

- 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

```
mysql> EXPLAIN SELECT
-> t.tag_text
-> 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
-> t.tag_text
-> FROM Tag t
-> INNER JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project = 75\G
***** 1. row *****
      id: 1
select_type: PRIMARY
      table: t
      type: index
      key: uix_tag_text
      key_len: 52
      rows: 1126
      Extra: Using index
***** 2. row *****
      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
***** 3. row *****
      id: 2
select_type: UNION
      table: t2p
      type: ref
possible_keys: PRIMARY,rv_primary
      key: rv_primary
      key_len: 4
      ref: const
      rows: 31
      Extra: Using index
```

```
***** 4. row *****
      id: 2
select_type: UNION
      table: t
      type: eq_ref
possible_keys: PRIMARY
      key: PRIMARY
      key_len: 4
      ref: mysqlforge.t2p.tag
      rows: 1
      Extra:
***** 5. row *****
      id: NULL
select_type: UNION RESULT
      table: <union1,2>
5 rows in set (0.00 sec)
```

```
mysql> SELECT
-> t.tag_text
-> 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
-> t.tag_text
-> FROM Tag t
-> INNER JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.project = 75;
```

```
+-----+
| tag_text |
+-----+
<snip>
+-----+
184 rows in set (0.00 sec)
```



LEFT join-fu: the trickiest part...

```
mysql> SELECT
-> t.tag_text
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.tag IS NULL
-> AND t2p.project= 75
-> GROUP BY t.tag_text;
Empty set (0.00 sec)

mysql> EXPLAIN SELECT
-> t.tag_text
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> WHERE t2p.tag IS NULL
-> AND t2p.project= 75
-> GROUP BY t.tag_text\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: NULL
       type: NULL
possible_keys: NULL
      key: NULL
     key_len: NULL
       ref: NULL
        rows: NULL
  Extra: Impossible WHERE noticed after reading const tables
1 row in set (0.00 sec)
```

- 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

```
mysql> EXPLAIN SELECT
-> t.tag_text
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> AND t2p.project= 75
-> WHERE t2p.tag IS NULL
-> GROUP BY t.tag_text\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t
         type: index
possible_keys: NULL
          key: uix_tag_text
        key_len: 52
         rows: 1126
       Extra: Using index
***** 2. row *****
      id: 1
  select_type: SIMPLE
        table: t2p
         type: eq_ref
possible_keys: PRIMARY,rv_primary
          key: rv_primary
        key_len: 8
         ref: const,mysqlforge.t.tag_id
         rows: 1
       Extra: Using where; Using index; Not exists
2 rows in set (0.00 sec)

mysql> SELECT
-> t.tag_text
-> FROM Tag t
-> LEFT JOIN Tag2Project t2p
-> ON t.tag_id = t2p.tag
-> AND t2p.project= 75
-> WHERE t2p.tag IS NULL
-> GROUP BY t.tag_text;
+-----+
| tag_text |
+-----+
<snip>
+-----+
674 rows in set (0.01 sec)
```

- 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



resources and thank you!

- PlanetMySQL
 - 300+ writers on MySQL topics
 - <http://planetmysql.com>
- MySQL Forge
 - Code snippets, project listings, wiki, worklog
 - <http://forge.mysql.org>



Baron Schwartz
<http://xaprb.com>

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

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