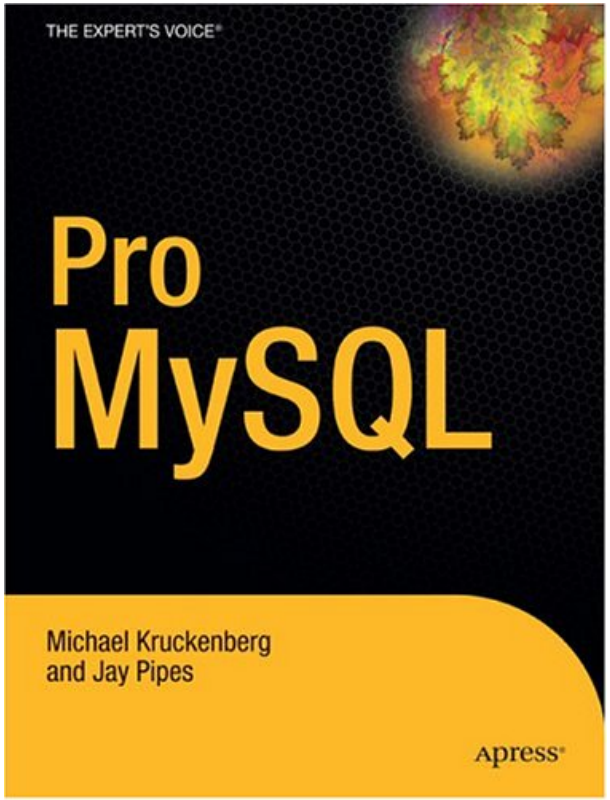
MySQL PERFORMANCE CODING: FROM SOUP TO NUTS

By Jay Pipes
Community Relations Manager, North America
MySQL, Inc.

Who the heck am I?



- Just some dude who works at MySQL
- Working with PHP and MySQL for 6+ years
- Oh, I wrote a book, too...
- Other than that, I'm semi-normal, with wife, two cats, two dogs, blah, blah, blah

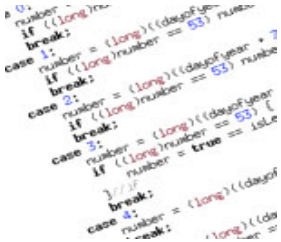
A quick survey...

- 3.23? 4.0? 4.1? 5.0? 5.1?
- MyISAM? InnoDB? Archive? Memory?
- Replication? Cluster?
- PHP 4? PHP 5? PHP 6?
- libmysql? Native Driver for MySQL?
- ext/mysql? ext/mysqli? PDO?
- Oracle? PostgreSQL? DB2? MSSQL? SQLite?

The big three topics

The Schema

The Code



The Server

But first, some general stuff...

- Performance != Scalability
- Benchmarking
- Overview of MySQL System Architecture
- Overview of the MySQL Query Cache
- The Scan vs. Seek Choice

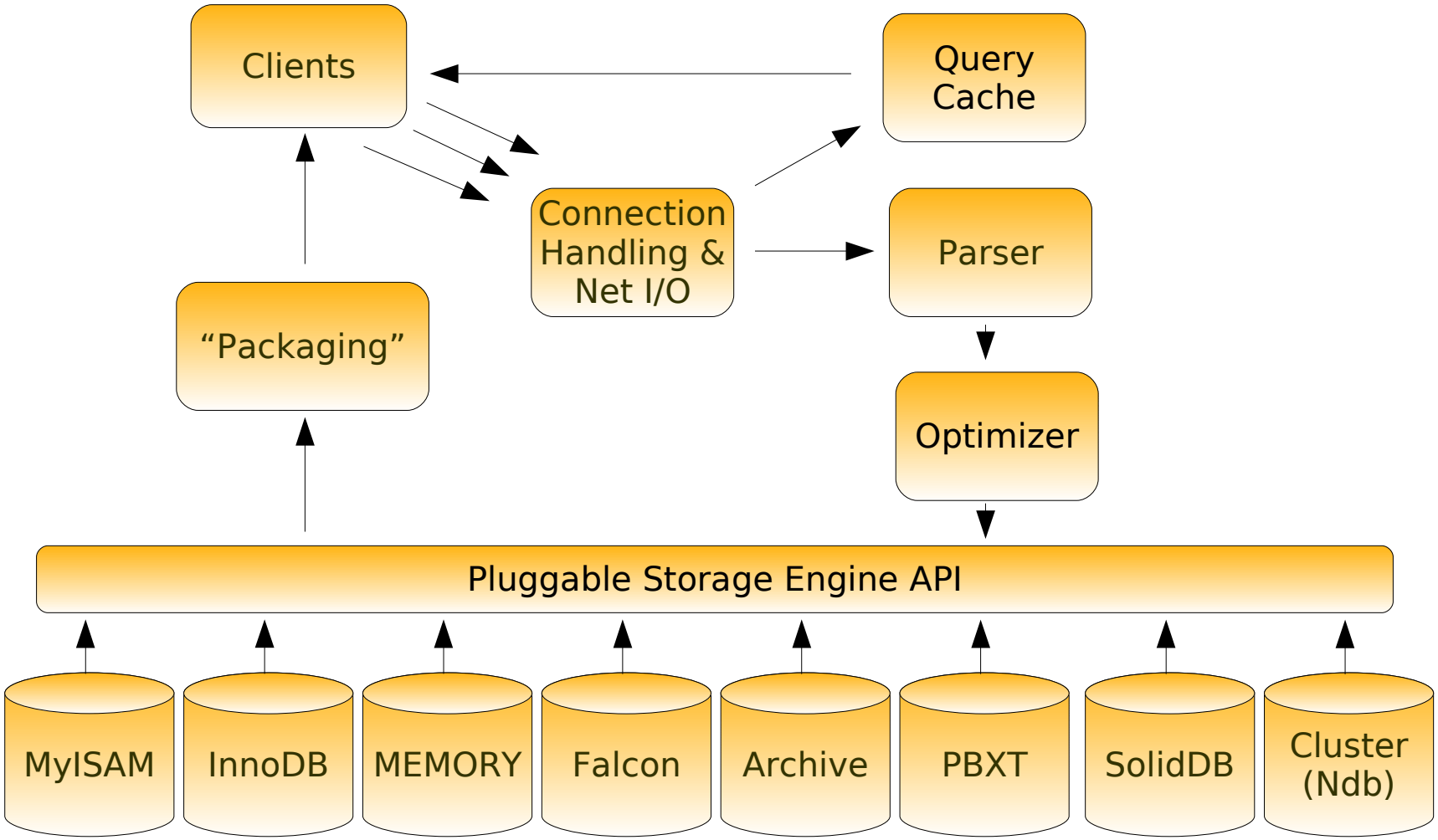
Performance !== Scalability

- Typically, we speak of performance when we talk about *response times* for a web page, an SQL statement, etc
- Scalability comes up when we talk about *throughput*, or the number of concurrent requests a node can serve within a certain timeframe
  - or the size of the *data* increases

Benchmarking

- Very important to benchmark both performance *and* scalability
  - So, test with multiple concurrency levels and varying dataset sizes
- Toolbox
  - mysqlslap
  - Apache Bench (ab)
  - sysbench
  - Custom
    - MyBench, Jmeter/Ant, Shell scripts...

MySQL system architecture



MySQL system architecture notes

- Highly coupled subsystems
- Emphasis on connection-based memory allocation (as opposed to global)
- Caching on many different levels
- Storage engine layer is both blessing and curse
- Optimizer is cost-based, simplistic, and must be guarded against
- Efforts to modularize ongoing

The MySQL query cache

- You ***must*** understand application read/write ratio
- Internal design is a compromise between CPU usage and read performance
- Bigger query cache != better performance, even for heavy read applications
- Not a silver bullet!

The scan vs. seek choice

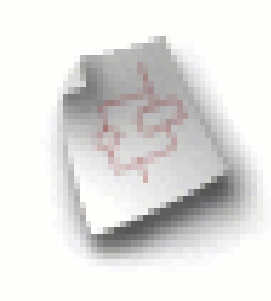
- A seek operation, generally speaking, **jumps to a random place** -- either on disk or in memory -- to fetch the data needed.
  - Repeat for each piece of data needed from disk or memory
- A scan operation, on the other hand, will jump to the start of a chunk of data, and **sequentially read data** -- 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*

And finally...

- Learn to use EXPLAIN!
- Too big a topic for this session, so download my workbook and slides from OSCON
- *http://jpipes.com/presentations/target-practice*
  - */target-practice-workbook.pdf*
  - */target-practice.pdf* *(or .odp)*

MySQL Performance Coding

The Schema



Data types

- Smaller, smaller, smaller!
  - Do you *really* need that BIGINT?
- More records in single page of memory, faster seeks & scans
- AUTO_INCREMENT is a good thing!
  - Generates a “hot spot” on disk and in memory
  - Look at 5.1.21 for InnoDB scaling patch
- Use appropriate data types

Appropriate data types

- INT UNSIGNED for IP 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

IPv4 addresses are INT UNSIGNED

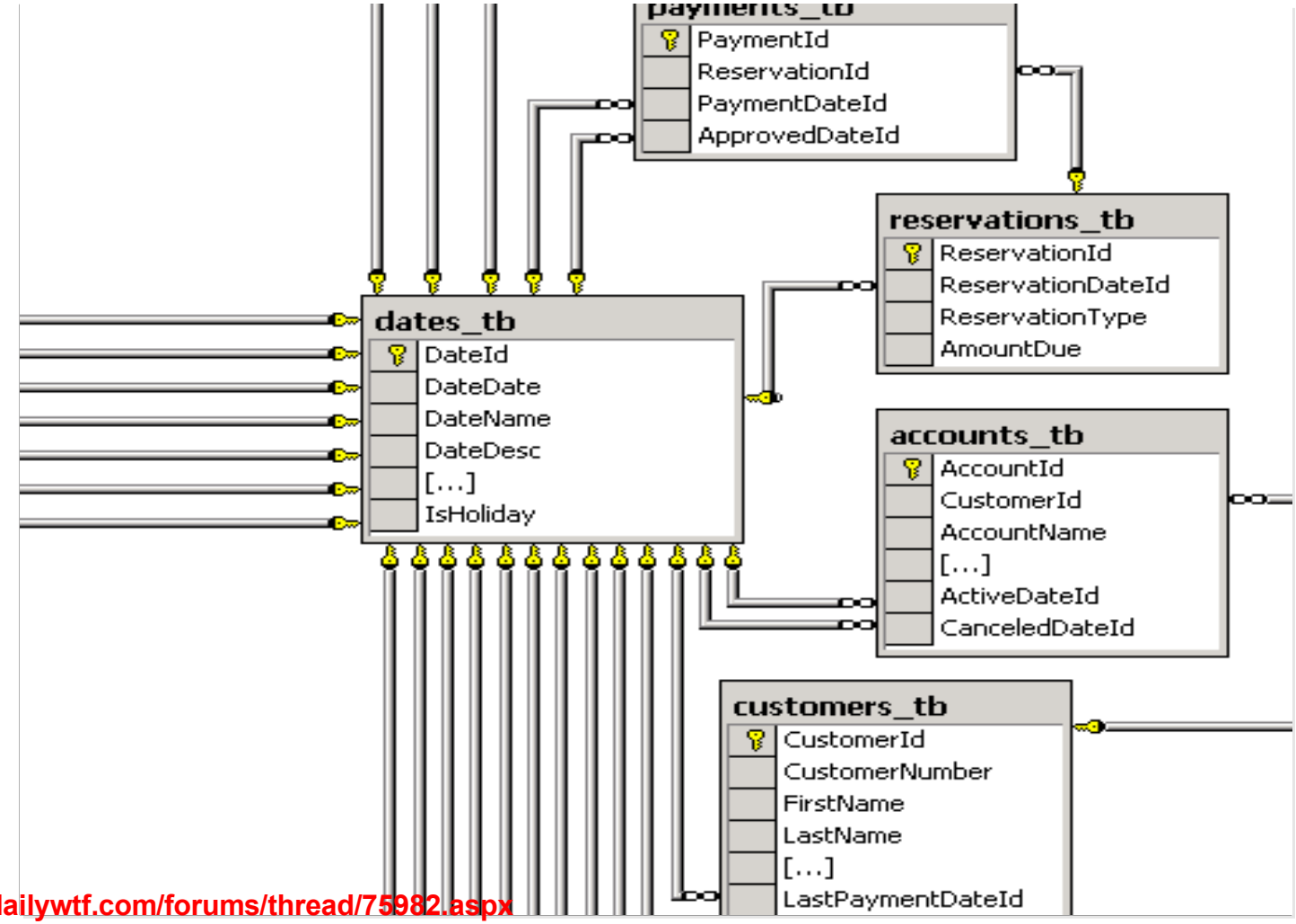
```
CREATE TABLE Sessions (
  session_id INT UNSIGNED NOT NULL AUTO_INCREMENT
, ip_address INT UNSIGNED NOT NULL // Compared to CHAR(15)!!
, session_data TEXT NOT NULL
, PRIMARY KEY (session_id)
, INDEX (ip_address)
) ENGINE=InnoDB;
```

```
// Find all sessions coming from a local subnet
SELECT * FROM Sessions
WHERE ip_address BETWEEN
INET_ATON('192.168.0.1') AND INET_ATON('192.168.0.255');
```

The INET_ATON() function reduces the string to a constant INT and a highly optimized range operation will be performed for:

```
SELECT * FROM Sessions
WHERE ip_address BETWEEN 3232235521 AND 3232235775
```

Normalize first, denormalize later. But...



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

Partitioning

- Vertical partitioning
  - Splits tables with many **columns** into multiple tables
- Horizontal partitioning
  - Splits table with many **rows** into multiple tables
- Both are important for different reasons
- Partitioning in MySQL 5.1 is *horizontal partitioning*

Vertical partioning example

```
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)
) ENGINE=InnoDB;
```

When vertical partitioning makes sense

- “Extra” columns are mostly NULL
- “Extra” columns are infrequently accessed
- When space in buffer pool is at a premium
  - Splitting the table allows main records to consume the buffer pages without the extra data taking up space in memory
  - Many more “main” records can fit into a single 16K InnoDB data page
- Need FULLTEXT on your text columns?

Vertical partioning example #2

```
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; // Or MyISAM

// 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; // Or MyISAM

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;
```

Vertical partitioning solves more problems

- 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

Horizontal partitioning options

- Eli covered this well yesterday
  - (Thanks so much, Eli, I deleted four slides from my talk about it.)
- MySQL 5.1 Partitioning
  - (Will be) good for taking advantage of multiple disks
- Custom partitioning (sharding)
  - Currently, the way to go

Indexes

- 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

What makes a column ideal for indexing?

- Selectivity
  - % of *distinct* values in a column
  - $S = d/n$
  - Unique/primary always be 1.0
- If column has a low selectivity, it can still be put in a multi-column index
  - But, which part? Prefix? Suffix?

Remove redundant or poor indexes

```
SELECT
  t.TABLE_SCHEMA AS `db`
 , t.TABLE_NAME AS `table`
 , s.INDEX_NAME AS `inde name`
 , s.COLUMN_NAME AS `field name`
 , s.SEQ_IN_INDEX `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;
```

http://forge.mysql.com/snippets/view.php?id=85

Whoah. Some crap-ass indexes, huh?

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

Effect of index column order

```
mysql> EXPLAIN SELECT project, COUNT(*) as num_tags
    -> FROM Tag2Project
    -> GROUP BY project;
+-------------+-------+---------+-----------------------------------------------+
| table       | type  | key     | Extra                                         |
+-------------+-------+---------+-----------------------------------------------+
| Tag2Project | index | PRIMARY | Using index; Using temporary; Using filesort |
+-------------+-------+---------+-----------------------------------------------+

mysql> EXPLAIN SELECT tag, COUNT(*) as num_projects
    -> FROM Tag2Project
    -> GROUP BY tag;
+-------------+-------+---------+-------------+
| table       | type  | key     | Extra       |
+-------------+-------+---------+-------------+
| Tag2Project | index | PRIMARY | Using index |
+-------------+-------+---------+-------------+

mysql> CREATE INDEX project ON Tag2Project (project)
Query OK, 701 rows affected (0.01 sec)
Records: 701  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT project, COUNT(*) as num_tags
    -> FROM Tag2Project
    -> GROUP BY project;
+-------------+-------+---------+-------------+
| table       | type  | key     | Extra       |
+-------------+-------+---------+-------------+
| Tag2Project | index | project | Using index |
+-------------+-------+---------+-------------+
```

The Tag2Project Table:

```
CREATE TABLE Tag2Project (
tag INT UNSIGNED NOT NULL
, project INT UNSIGNED NOT NULL
, PRIMARY KEY (tag, project)
) ENGINE=MyISAM;
```

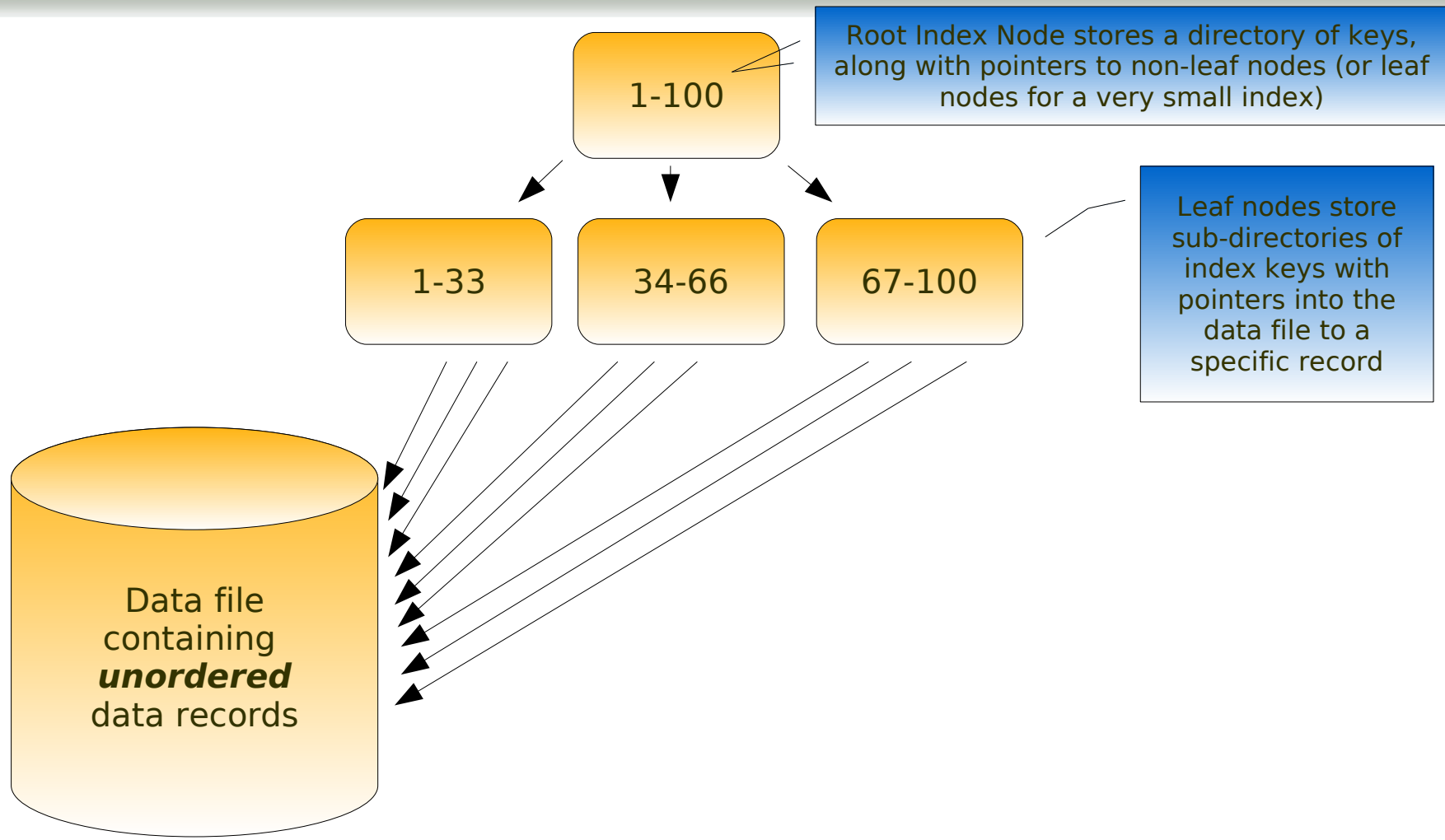
How do you tell if this is happening?

Look for increases in:
Created_tmp_tables and
Created_tmp_disk_tables
status counter variables

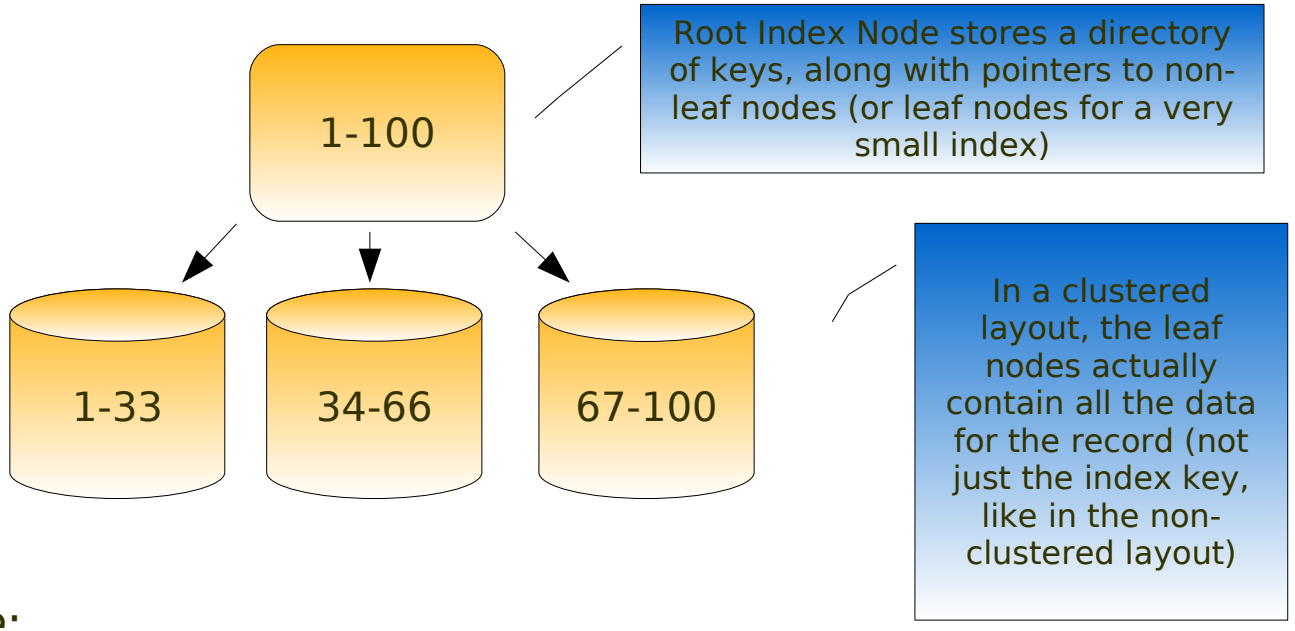
Covering indexes

- When all columns needed from a single table for a SELECT are available in the index
- No need to grab the rest of the columns from the data (file or page)
  - “Bookmark lookup” operation
- Important to know the data to index organization of the storage engine!

Non-clustered organization (MyISAM)



Clustered organization (InnoDB)



So, bottom line:

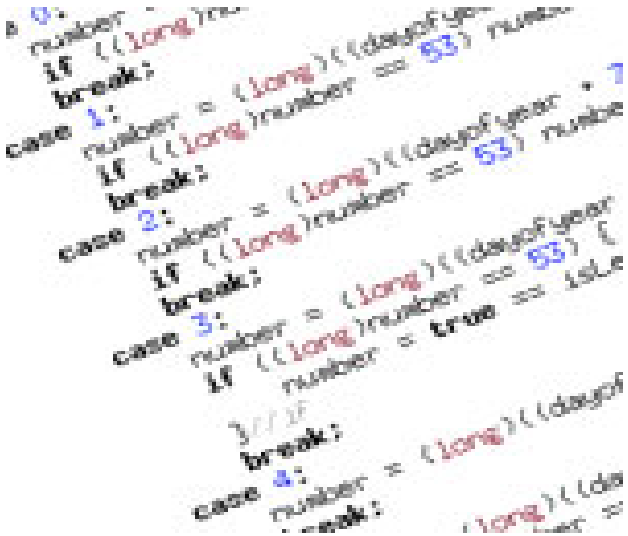
When looking up a record by a primary key, for a clustered layout/organization, the **lookup operation** (following the pointer from the leaf node to the data file) involved in a non-clustered layout **is not needed.**

Clustered indexes

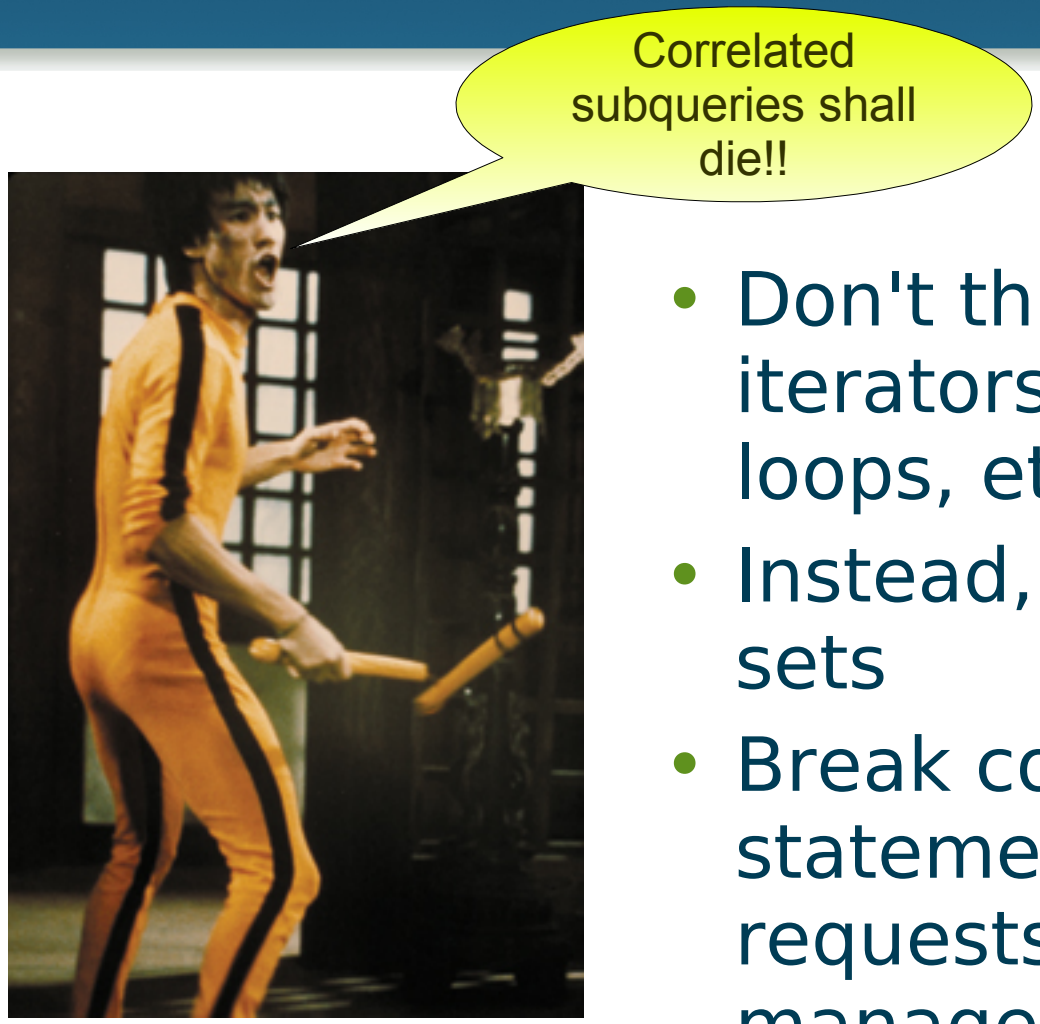
- Very important to have as small a clustering key (primary key) as possible
  - Why? Because every secondary index built on the table will have the primary key appended to each index record
- If you don't pick a primary key (bad idea!), one will be created for you
  - And, you have no control over the key (this is a 6 byte number in InnoDB...)

MySQL Performance Coding

The Code



Be a join-fu master!



- Don't 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

Set-wise thinking

"Show the last payment information for each customer"

- Many programmers think:
  - OK, for each customer, find the maximum date the payment was made get that payment record(s) (bad!)
- Think instead:
  - OK, I have 2 sets of data here. One set of last payments dates and another set containing payment information (so, how do I ***join*** these sets?)

The difference in execution plans?

```
mysql> EXPLAIN SELECT
    ->   p.*
    -> FROM payment p
    -> WHERE p.payment_date =
    -> ( SELECT MAX(payment_date)
    ->   FROM payment
    ->   WHERE customer_id=p.customer_id);
+--------------------+---------+------+---------------------------------+--------------+---------------+-------+-------------+
| select_type        | table   | type | possible_keys                   | key          | ref           | rows  | Extra       |
+--------------------+---------+------+---------------------------------+--------------+---------------+-------+-------------+
| PRIMARY            | p       | ALL  | NULL                            | NULL         | NULL          | 16451 | Using where |
| DEPENDENT SUBQUERY | payment | ref  | idx_fk_customer_id,payment_date | payment_date | p.customer_id |    12 | Using index |
+--------------------+---------+------+---------------------------------+--------------+---------------+-------+-------------+
3 rows in set (0.00 sec)
```

```
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;
+-------------+------------+-------+---------------------------------+--------------------+-------------------------------+-------+
| select_type | table      | type  | possible_keys                   | key                | ref                           | rows  |
+-------------+------------+-------+---------------------------------+--------------------+-------------------------------+-------+
| PRIMARY     | <derived2> | ALL   | NULL                            | NULL               | NULL                          |   599 |
| PRIMARY     | p          | ref   | idx_fk_customer_id,payment_date | payment_date       | customer_id,last_order        |     1 |
| DERIVED     | payment    | index | NULL                            | idx_fk_customer_id | NULL                          | 16451 |
+-------------+------------+-------+---------------------------------+--------------------+-------------------------------+-------+
3 rows in set (0.10 sec)
```

The difference in performance

```
mysql> SELECT
    ->   p.*
    -> FROM payment p
    -> WHERE p.payment_date =
    -> ( SELECT MAX(payment_date)
    ->   FROM payment
    ->   WHERE customer_id=p.customer_id);
+------------+-------------+----------+-----------+--------+---------------------+---------------------+
| payment_id | customer_id | staff_id | rental_id | amount | payment_date        | last_update         |
+------------+-------------+----------+-----------+--------+---------------------+---------------------+
<snip>
|      16049 |         599 |        2 |     15725 |   2.99 | 2005-08-23 11:25:00 | 2006-02-15 19:24:13 |
+------------+-------------+----------+-----------+--------+---------------------+---------------------+
623 rows in set (0.49 sec)
```

```
mysql> 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;
+------------+-------------+----------+-----------+--------+---------------------+---------------------+
| payment_id | customer_id | staff_id | rental_id | amount | payment_date        | last_update         |
+------------+-------------+----------+-----------+--------+---------------------+---------------------+
<snip>
|      16049 |         599 |        2 |     15725 |   2.99 | 2005-08-23 11:25:00 | 2006-02-15 19:24:13 |
+------------+-------------+----------+-----------+--------+---------------------+---------------------+
623 rows in set (0.09 sec)
```

A word on too many joins

- Don't try to do joins on >8 tables
  - Especially with MySQL <5.0
  - Sometimes optimizer can try too hard to find optimal plan
- Use a “temp table reduction” recipe
  - Especially important for AND conditions on many-to-many relation tables
- Or for small, static lookups, use ENUM (or SET for many-to-many)

Operating on indexed column w/ function

```
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
```

Nice. In the top query, we have a fast range access on the indexed field

```
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
```

Oops. In the bottom query, we have a slower full table scan because of the function operating on the indexed field (the LEFT() function)

Operating on indexed column w/ function #2

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

Not a good idea! Lots o' problems with this...

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

Better... Now the index on order_created will be used at least. Still a problem, though...

```
SELECT order_id, order_created, customer
FROM Orders
WHERE order_created
>= '2007-02-11' - INTERVAL 7 DAY;
```

Best. Now the query cache can cache this query, and given no updates, only run it once a day...

replace the CURRENT_DATE() function with a constant string in your programming language du jour... for instance, in PHP, we'd do:

```
$sql= "SELECT order_id, order_created, customer FROM Orders WHERE
order_created >= '" .
date('Y-m-d') . "' - INTERVAL 7 DAY";
```

Calculated fields example

```
CREATE TABLE Customers (
  customer_id INT NOT NULL
, email VARCHAR(80) NOT NULL
// more fields
, PRIMARY KEY (customer_id)
, INDEX (email(40))
) ENGINE=InnoDB;

// Bad idea, can't use index
// on email field
SELECT *
FROM Customers
WHERE email LIKE '%.com';
```

```
// So, we enable fast searching on a reversed field
// value by inserting a calculated field
ALTER TABLE Customers
ADD COLUMN rv_email VARCHAR(80) NOT NULL;

// Now, we update the existing table values
UPDATE Customers SET rv_email = REVERSE(email);

// Then, we create an index on the new field
CREATE INDEX ix_rv_email ON Customers (rv_email);

// Then, we make a trigger to keep our data in sync
DELIMITER ;;
CREATE TRIGGER trg_bi_cust
BEFORE INSERT ON Customers
FOR EACH ROW BEGIN
 SET NEW.rv_email = REVERSE(NEW.email);
END ;;

// same trigger for BEFORE UPDATE...
// Then SELECT on the new field...
WHERE rv_email LIKE CONCAT(REVERSE('.com'), '%');
```

Using stored procedures

- Question: where does the stored procedure compile cache live?
- Don't use stored procedures for simple SELECTs
- Use for:
  - ETL or complex collections of SQL
  - Repeated execution of statement
  - Batch operations

UPDATEs and DELETEs

- Avoid DELETE, especially in MyISAM
  - Use a deleted_rows table
  - Insert rows into the table, then do batched DELETEs
- Have lots of UPDATEs?
  - Insert them into memcache bucket, then periodically update the tables...

MySQL Performance Coding

The Server

SHOW STATUS and SHOW VARIABLES

- SHOW STATUS
  - Counter variables (lots of `em)
  - Count reads, writes, threads, etc.
- SHOW VARIABLES
  - Your configuration variables
- Both take a LIKE clause, for example:

```
mysql> SHOW STATUS LIKE 'Created_tmp%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 499   |
| Created_tmp_files       | 5     |
| Created_tmp_tables      | 1933  |
+-------------------------+-------+
```

Server variable guidelines

- Be aware of what is *global* vs *per thread*
- Make small changes, then test
- Often provide a quick solution, but temporary
- *key_buffer_size* != *innodb_buffer_pool_size*
- Memory is cheapest, fastest, easiest way to increase performance
  - But... bigger buffers aren't always a good thing!

Important settings

- **key_buffer_size** (*global, MyISAM only*)
  - Main MyISAM key cache (blocks of size 1K)
  - Watch for Key_blocks_unused approaching 0
- **table_cache** *(global)*
  - Number of simultaneously open file descriptors
  - < 5.1 contains meta data about tables and file descriptor
  - >= 5.1 Split into table_open_cache
- **myisam_sort_buffer_size** (*global, MyISAM only*)
  - Building indexes?
  - Set this as high as possible

Average table scan and key cache hit ratio

- Examine **Handler_read_rnd_next/Handler_read_rnd** for *average size of table scans*

```
mysql> SHOW STATUS LIKE 'Handler_read_rnd%';
+-----------------------+--------+
| Variable_name         | Value  |
+-----------------------+--------+
| Handler_read_rnd      | 2188   |
| Handler_read_rnd_next | 217247 |
+-----------------------+--------+
```

- Examine **Key_read_requests/Key_reads** for your MyISAM *key cache hit ratio*

```
mysql> SHOW STATUS LIKE 'Key_read%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Key_read_requests | 10063 |
| Key_reads         | 98    |
+-------------------+-------+
```

Important settings for InnoDB

- **innodb_buffer_pool_size**
  - Main InnoDB cache for both data and index pages (16K page)
  - If you have InnoDB-only system, set to 60-80% of total memory
  - Watch for **Innodb_buffer_pool_pages_free** approaching 0
- **innodb_log_file_size**
  - Size of the actual log file
  - Set to 40-50% of **innodb_buffer_pool_size**
  - Bigger means longer recovery, but less disk I/O due to less checkpoint flush activity

Important settings for InnoDB

- **innodb_log_buffer_size**
  - Size of double-write log buffer
  - Set < 16M (recommend 1M to 8M)
- **innodb_flush_method**
  - Determines how InnoDB flushes data and logs
  - defaults to fsync()
  - If getting lots of **Innodb_data_pending_fsyncs**
    - Consider O_DIRECT (Linux only)
  - Other ideas
  - Get a battery-backed disk controller with a write-back cache
  - Set **innodb_flush_log_at_trx_commit**=2 (Risky)

Important settings for InnoDB

- Examine **Innodb_buffer_pool_reads** vs **Innodb_buffer_pool_read_requests** for the *cache hit ratio*

```
mysql> SHOW STATUS LIKE 'Innodb_buffer_pool_read%';
+---------------------------------+---------+
| Variable_name                   | Value   |
+---------------------------------+---------+
| Innodb_buffer_pool_read_requests  | 5415365 |
| Innodb_buffer_pool_reads          | 34260   |
+---------------------------------+---------+
```

```
mysql> SHOW STATUS LIKE 'Qc%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Qcache_free_blocks      | 1     |
| Qcache_hits             | 6     |
| Qcache_inserts          | 12    |
| Qcache_not_cached       | 41    |
| Qcache_lowmem_prunes    | 0     |
| Questions               | 241   |
+-------------------------+-------+
```

- Examine **Qcache_hits/Questions** for the *query cache hit ratio*
- Ensure **Qcache_lowmem_prunes** is low
- Ensure **Qcache_free_blocks = 1**
  - if not, **FLUSH QUERY CACHE**

Yo, we're hiring.

- Dozens of positions open
- Work from anywhere
- Great bennies
- Travel if you want to
- A Swedish company culture (but, sorry, no free massages or hot babes.)
- 5 weeks vacation