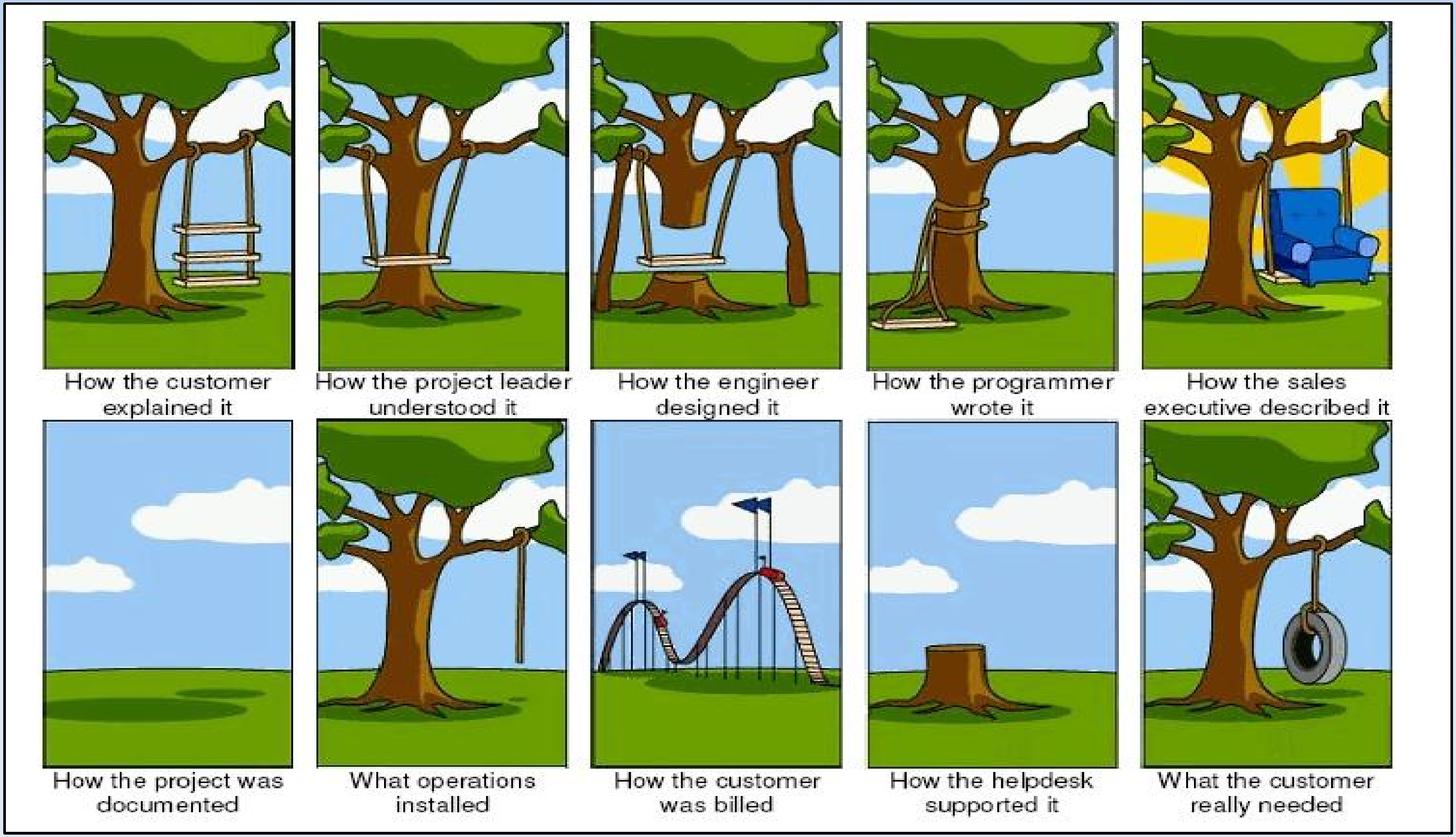
Phew. Break time. :)


How the customer explained it
How the project leader understood it
How the engineer designed it
How the programmer wrote it
How the sales executive described it
How the project was documented
What operations installed
How the customer was billed
How the helpdesk supported it
What the customer really needed

A word about Ask

- Ask is looking at the “big picture”
- That's all good...
- But, hey, it's more fun to dig in to the code!

Ask's shovel

Jay's shovel

A Word of Warning...

The answer to every question will be ....

“It depends.”

Get Your Learn On

- The EXPLAIN command
- Schema and indexing guidelines
- Black-belt SQL coding
- Tuning server settings

The EXPLAIN Command

The EXPLAIN command

- Simply append EXPLAIN before any SELECT statement
- Returns the execution plan chosen
- Each row in output is a set of information
  - A real schema table
  - A “virtual” table (a subquery in the FROM clause)
  - A subquery in the SELECT or WHERE clause
  - A UNIONed resultset

Example EXPLAIN output

```
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
*************************** 1. row ***************************
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16
        Extra:
*************************** 2. row ***************************
  select_type: SIMPLE
        table: fc
         type: ref
possible_keys: PRIMARY,fk_film_category_category
          key: fk_film_category_category
      key_len: 1
          ref: sakila.c.category_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
  select_type: SIMPLE
        table: f
         type: eq_ref
possible_keys: PRIMARY,idx_title
          key: PRIMARY
      key_len: 2
          ref: sakila.fc.film_id
         rows: 1
        Extra: Using where
```

An *estimate* of rows in this set

The “access strategy” chosen

The available indexes, and the one(s) chosen

A **covering index** is used

The `type` column – avoid the “ALL”

- Perhaps the most important column in EXPLAIN's output
- Tells you the *access strategy* which MySQL chose to retrieve the specified rows
- Watch out for the “**ALL**” access type!
- It means you are doing a *full table scan* of the table's records
- Let's see what it looks like...

ALL access type example

```
mysql> EXPLAIN SELECT * FROM rental \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 15258
        Extra:
1 row in set (0.01 sec)
```

Here, we see that a ***full table scan*** is planned. This makes sense, considering we gave MySQL no WHERE clause by which the optimizer could filter the rows or use an index. Also, note the difference between this query, which uses a SELECT * FROM rental, and the next, which selects only the rental_date field...

The `type` column – the “index” scan

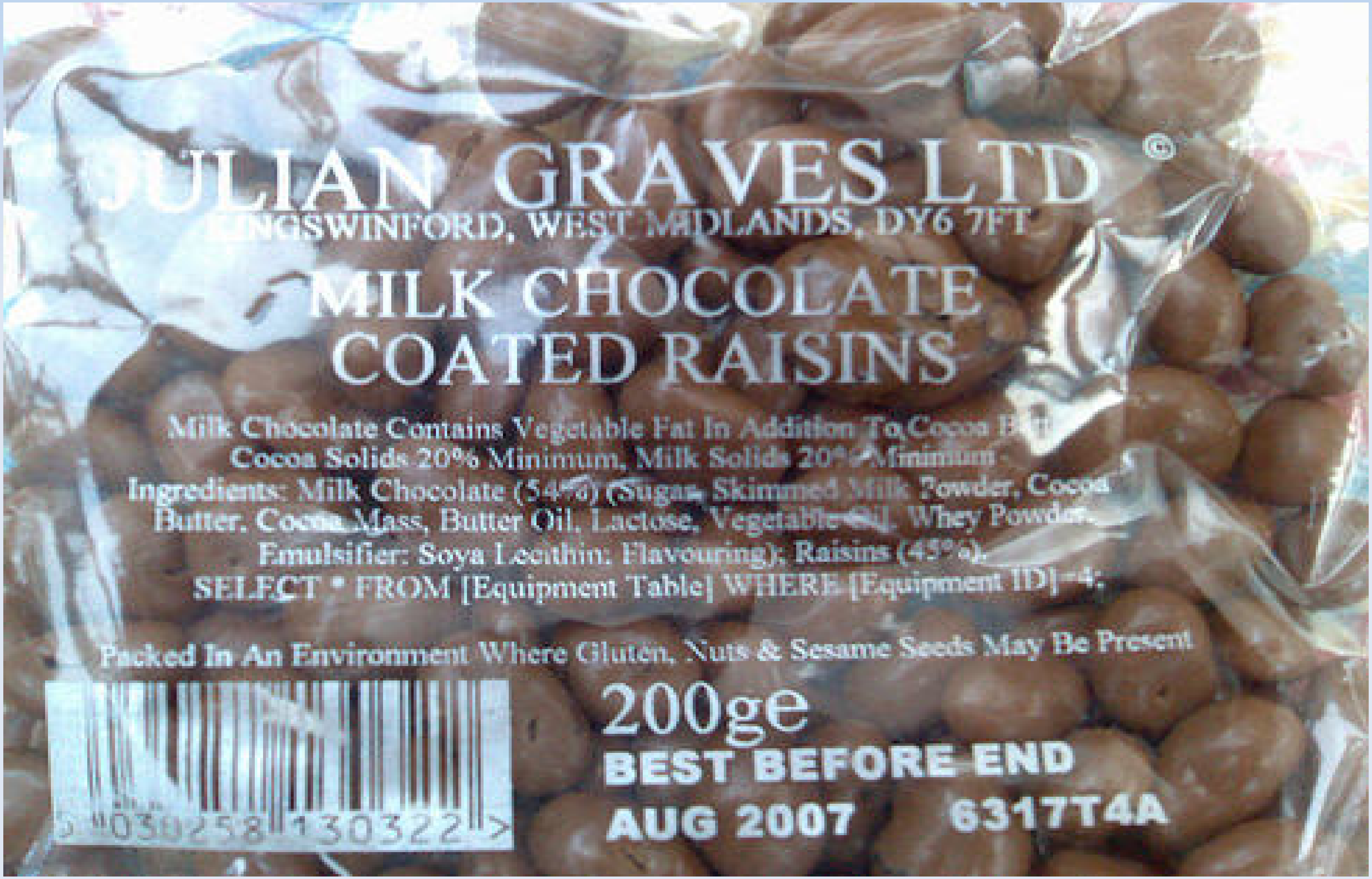
- The “**index**” access type is **NOT** a good thing!
- It means you are doing a full index scan of all the index' records
- Better than a full table scan in *most* cases, but still requires a LOT of resources
- Let's see what it looks like...

index access type example

```
mysql> EXPLAIN SELECT 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: 15258
        Extra: Using index
1 row in set (0.00 sec)
```

Here, we see that a ***full index scan*** is planned. By specifying that we only wish to see the rental_date column, we are essentially informing the query optimizer that if an index contains the rental_date information, there is no need to pull in the rest of the table fields; instead, the index itself can be used to supply all needed data...

Ahhhh... SELECT *



JULIAN GRAVES LTD ©
KINGSWINFORD, WEST MIDLANDS, DY6 7FT

MILK CHOCOLATE
COATED RAISINS

Milk Chocolate Contains Vegetable Fat In Addition To Cocoa B
Cocoa Solids 20% Minimum, Milk Solids 20% Minimum
Ingredients: Milk Chocolate (54%) (Sugar, Skimmed Milk Powder, Cocoa Butter, Cocoa Mass, Butter Oil, Lactose, Vegetable Oil, Whey Powder, Emulsifier: Soya Lecithin: Flavouring); Raisins (45%).
SELECT * FROM [Equipment Table] WHERE [Equipment ID]=4;

Packed In An Environment Where Gluten, Nuts & Sesame Seeds May Be Present

030258 130322

200ge

BEST BEFORE END
AUG 2007 6317T4A

The `type` column – the “range”

- You have specified a **`WHERE`** or **`ON`** clause that uses a range filter
  - The BETWEEN operator
  - The **IN** operator
  - The **>**, **>=**, **<=**, or **<** operators
- MySQL has many optimizations for range queries, which make this a good access type
- But... you **must** have an *index* on the field
- Let's take a look...

range access type example

```
mysql> EXPLAIN SELECT * FROM rental
    -> WHERE rental_date
    -> BETWEEN '2006-01-01' AND '2006-07-01' \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: 2614
        Extra: Using where
1 row in set (0.00 sec)
```

Here, we see that a ***range*** access is planned. The **BETWEEN** operator means we want to access rental records corresponding to a range of rental dates. Note that the possible_keys column shows us that an *index* on rental_date is available for the optimizer to use a range access pattern.

But what would happen if there weren't an index on rental_date?

Ooops. Back to a full table scan

```
mysql> DROP INDEX rental_date ON rental;
Query OK, 16044 rows affected (1.20 sec)
Records: 16044  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT * FROM rental
    -> WHERE rental_date
    -> BETWEEN '2006-01-01' AND '2006-07-01' \G
*************************** 1. row
           id: 1
  select_type: SIMPLE
        table: rental
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16462
        Extra: Using where
1 row in set (0.01 sec)
```

Uh oh. Because there is no index available on the field we are filtering by, MySQL cannot use a range optimization, and resorts to the (horrible) full table scan, doing a filter on each sequential record to find records meeting our criteria... so ***indexes are critically important!***

EXPLAIN Tips

- There is a **huge** difference between “index” in the type column and “Using index” in the Extra column
  - In the type column, it means a full index scan
  - In the Extra column, it means a covering index
- 5.0+ look for the index_merge optimization
  - Prior to 5.0, only one index can be used per table
    - Would have to use a UNION to achieve same results
  - 5.0+ if multiple indexes can be used, can use them

Index Merge Example (5.0+)

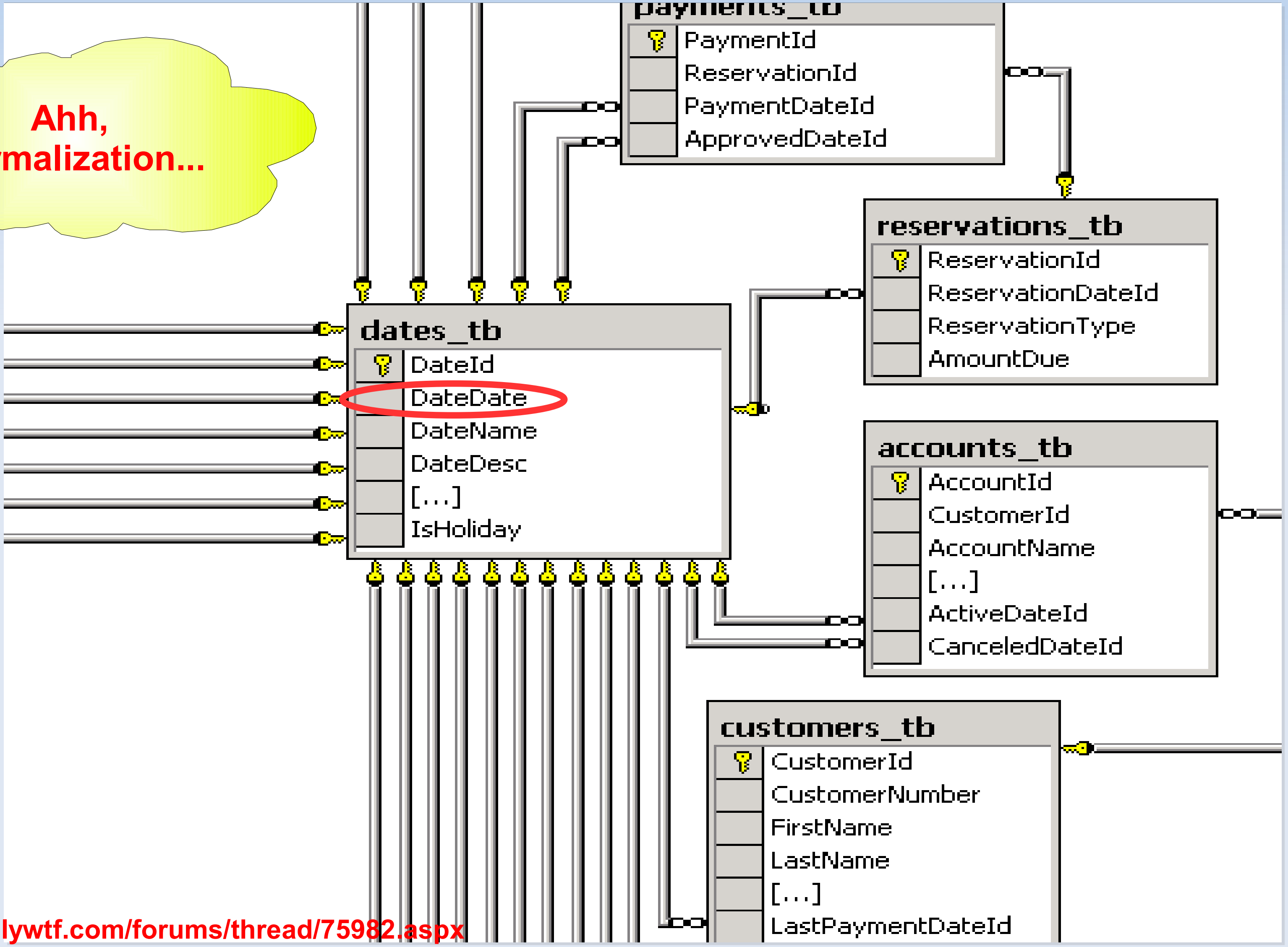
```
mysql> EXPLAIN SELECT * FROM rental
    -> WHERE rental_id IN (10,11,12)
    -> OR rental_date = '2006-02-01' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: index_merge
possible_keys: PRIMARY,rental_date
          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 (0.04 sec)
```

Schema and Index Strategies

Schema

- Poor schema is a great way to shoot yourself in the foot
- Use smallest data types possible (esp. InnoDB)
  - Do you *really* need that BIGINT?
- Smaller the field structure in table or index row, the more records can fit into a single page (so faster accesses!)
- Normalize first, then de-normalize only in extreme cases

Journey to the Center of the Database



payments_tb
- PaymentId
- ReservationId
- PaymentDateId
- ApprovedDateId

reservations_tb
- ReservationId
- ReservationDateId
- ReservationType
- AmountDue

dates_tb
- DateId
- DateDate
- DateName
- DateDesc
- [...]
- IsHoliday

accounts_tb
- AccountId
- CustomerId
- AccountName
- [...]
- ActiveDateId
- CanceledDateId

customers_tb
- CustomerId
- CustomerNumber
- FirstName
- LastName
- [...]
- LastPaymentDateId

Ahh, normalization...

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

Horizontal Partitioning Example

```sql
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
, company TEXT NULL
, PRIMARY KEY (user_id)
, UNIQUE INDEX (email)
) ENGINE=InnoDB;
```

```sql
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
, company TEXT NULL
, PRIMARY KEY (user_id)
) ENGINE=InnoDB;
```

When Horizontal 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
- To use FULLTEXT on your text columns

Counter Table Example

```
CREATE TABLE Products (
  product_id INT NOT NULL AUTO_INCREMENT
, 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 AUTO_INCREMENT
, 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 ProductCountSummary (
  total_products INT UNSIGNED NOT NULL
) ENGINE=MEMORY;
```

When Counter Tables Make Sense

- 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

Identifying Good Field Candidates for Indexes

- Good Selectivity (% distinct values in field)
- Used in WHERE? ON? GROUP BY? ORDER BY?
- How to determine selectivity of current indexes?
  - `SHOW INDEX FROM some_table`
    - Repeat as needed
  - `SELECT COUNT(DISTINCT some_field)/COUNT(*) FROM some_table`
    - Repeat as needed
  - or, use the INFORMATION_SCHEMA ...

INFORMATION_SCHEMA is your friend

```
+--------------+------------------+----------------------+-------------+--------------+---------------+------+-------+-------+
| 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 |
+--------------+------------------+----------------------+-------------+--------------+---------------+------+-------+-------+
```

Effects of Column Order in Indexes

```
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;
```

Random schema and index tips

- Don't store IP addresses as strings
  - Use **INET_ATON()** and **INET_NTOA()** for conversion
- Never do a **GROUP BY** on a non-indexed column
- Use **AUTO_INCREMENT** liberally
- Take advantage of all the storage engines
  - Know which works best for different needs
- Don't reinvent wheels. Don't use **GET_LOCK()** ... use InnoDB (or PBXT or Falcon)

Real World MySQL Tuning

Black-belt SQL Coding

SQL Coding Guidelines

- Change the way you think
  - SQL Programming != Procedural Programming
- No more “for” loop thinking
  - Instead, learn to think in “sets”
- KISS (Keep it Simple and Straightforward)
  - “Chunky” coding
  - If it looks too complex, break it down
- Be **consistent**
  - Helps you *and* your team

Thinking in Sets

“*Show the maximum price that each product was sold, along with the product name for each product*”

- Many programmers think:
  - OK, **for each** product, find the maximum price the product was sold and output that with the product's name (WRONG!)
- Think instead:
  - OK, I have **2 sets** of data here. One set of product names and another set of maximum sold prices

Not everything is as it seems...

```
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)
```

...not what you expected?

```
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)
```

Isolate those Indexed Fields!

```
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)

A Very Common Isolated Index Field Problem

```
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 Field 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'), '%');
```

A word on stored procedures

- Totally different implementation from other RDBMS
  - Cached on *connection thread*!
    - So, little performance benefit if application does lots of different, non-repetitive requests on a page request
- Can increase CPU on database server
  - Fundamental principal in scaling: don't make your database server the point of contention
  - Putting stored procedures in the DB means you will need to scale ***UP***, instead of scaling ***OUT*** (bad...)

Tuning Server Settings

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
- Query Cache is not a panacea
- **key_buffer_size** != **innodb_buffer_pool_size**
- Remember mysql system database is MyISAM
- Memory is cheapest, fastest, easiest way to increase performance

MyISAM

- **key_buffer_size**
  - Main MyISAM key cache (blocks of size 1K)
  - Watch for **Key_blocks_unused** approaching 0
- **table_cache** (InnoDB too...)
  - 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**
  - Building indexes, set this as high as possible

MyISAM

- 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 *key_cache hit ratio*

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

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**

InnoDB (cont'd)

- **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)

Examining Hit Rates (InnoDB and Query Cache)

- 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** > 0

Further Reading

- ***Optimizing Linux Performance***
  - Philip Ezolt, HP Press
- **http://www.mysqlperformanceblog.com/**
  - Peter Zaitsev
- **http://xaprb.com**
  - Baron Schwartz
- **http://planetmysql.org**
  - Planet MySQL
- ***Advanced PHP Programming***
  - George Schlossnagle, Developer's Library

Get involved!

- **http://forge.mysql.com**
  - The MySQL Forge (projects, wiki, code snippets)
- **http://forge.mysql.com/worklog/**
  - The MySQL roadmap (comment on it!)
- **The Quality Contributor Program**
  - Report bugs, submit test cases and/or patches
  - Win free Enterprise software and Über-geek status
- **The IRC Channels (irc.freenode.net)**
  - **#mysql** (General questions and SQL help)
  - **#mysql.hacking** (Contributors, MySQL devs and engineers, hardcore C/C++)