15 Ways to Kill Your MySQL Application Performance

Jay Pipes
Community Relations Manager, North America
MySQL, Inc.
jay@mysql.com

MySQL

Before we get started...a quick poll

- 3.23? 4.0? 4.1? 5.0? 5.1? 5.2/6.0?
- PostgreSQL? Oracle? SQL Server? DB2? SQLite? Others?
- OLAP? OLTP? Mix?
- MyISAM? InnoDB? Others? (Falcon or PBXT, anyone?)
- Developer? DBA? Mix?

Oh, and one more thing...

The answer to every question will be...

It depends.

Get your learn on.

- 15 tips of what ***not*** to do
- Some may surprise you
- Others won't (but you probably still do them)
- Have a short question? Just ask it
- Longer questions, save to the end

#1: Thinking too small

If you need to move some serious data or deal with massive scale, you need to think about the ecosystem in which MySQL lives.

The dolphin swims in a big sea

- Surrounded by web servers, application servers, DNS servers, etc
- Proxies and caching at every level
- No major website exists without caching heavily
- See Ask Hansen's slides (develooper.com) and Ilia's great tutorial

Architect for scale *out* from the start

- ✓ Detach components and application pieces from each other
- ✓ Never rely on a single “big box” architecture
- ✓ Plan for replication and/or partitioning early
- ✓ Keep session data for transient, small data sets (oh, and don't use file-based sessions)

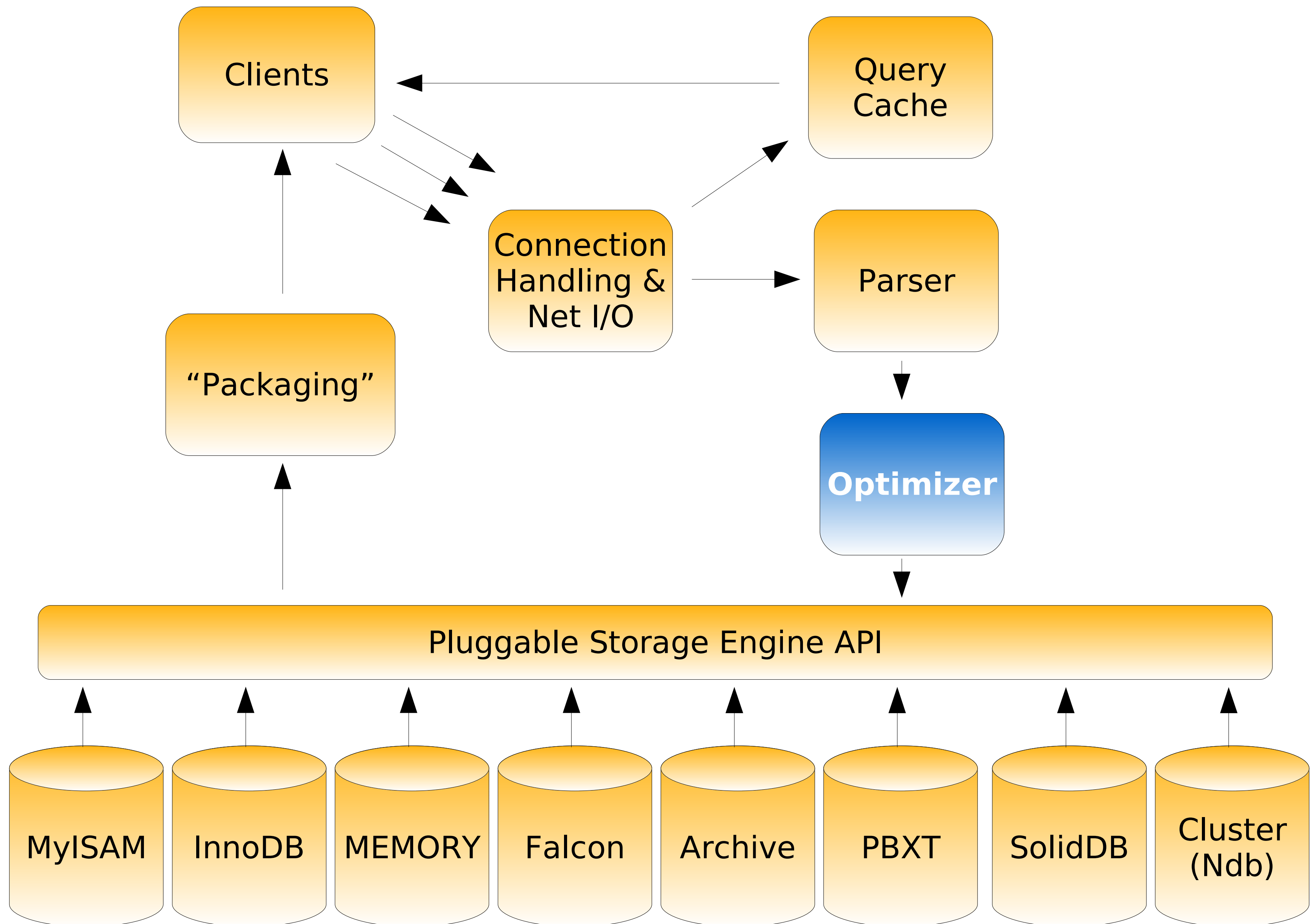
But wait! Don't think *too* big

The biggest performance gains will come from changes in the way you write your SQL code, design your schema, and apply indexing strategies

Remember,

performance != scalability

#2: Not using EXPLAIN



Explaining EXPLAIN

- Simply append EXPLAIN before any SELECT statement
- Returns the execution plan chosen by the optimizer
- Each row in output represents a set of information used in the SELECT
  - A real schema table
  - A virtual table (derived table)
  - A subquery in SELECT or WHERE
  - A unioned set

Sample 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

Tips on using EXPLAIN

- 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 (bad!)
  - In the Extra column, it means a covering index was found (good!)
- 5.0+ look for the index_merge optimization
  - Prior to 5.0, only one index used, even if more than one were useful

index_merge example

```
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)
```

Prior to 5.0, the optimizer would have to choose which index would be best for winnowing the overall result and then do a secondary pass to determine the OR condition, or, more likely, perform a full table scan and perform the WHERE condition on each row

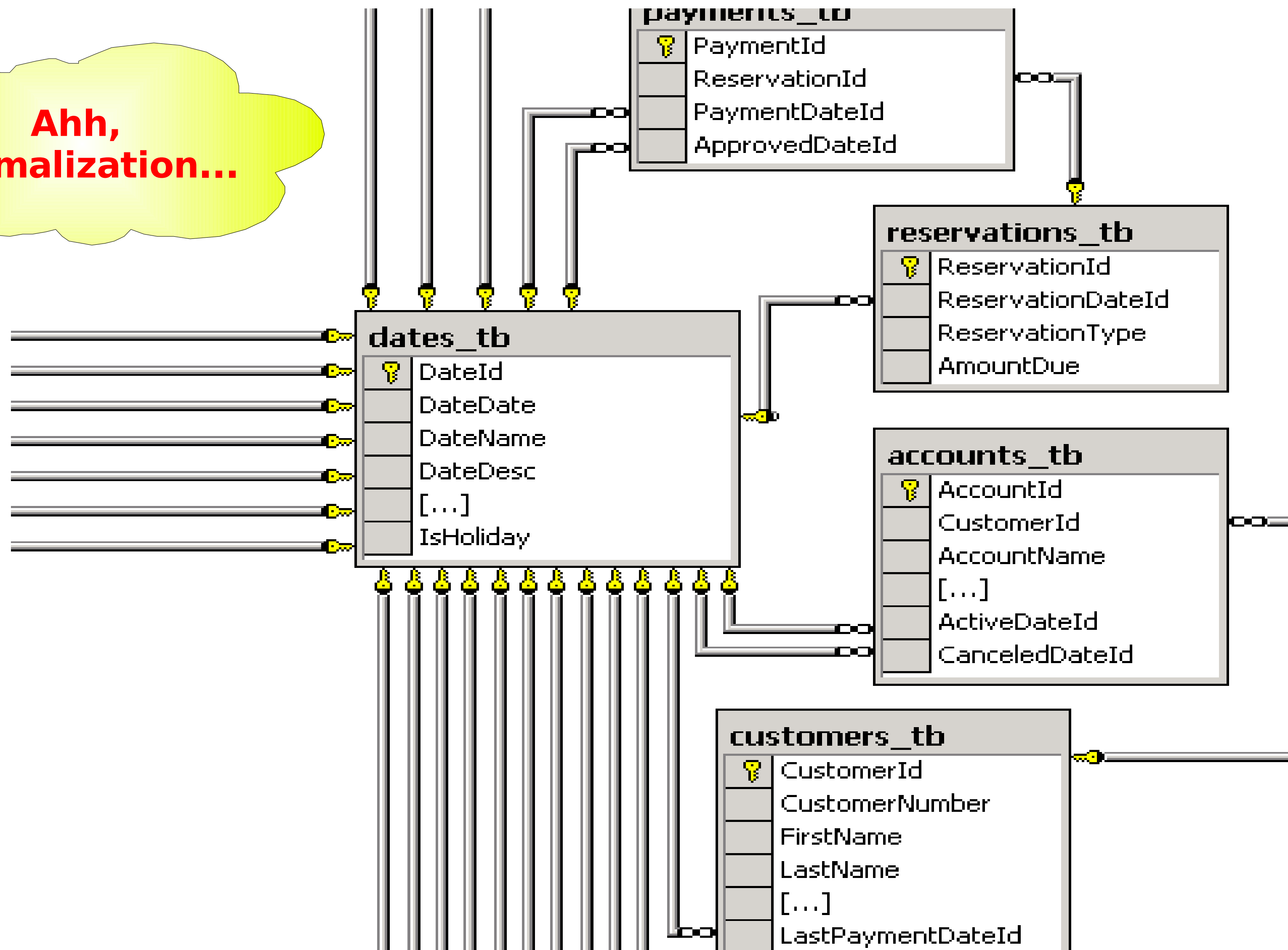
#3: Choosing the wrong data types

A concept to remember:

The more index (and data) records can fit into a single block of memory, the faster your queries will be.

Period.

Journey to the center of the database



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

Smaller, smaller, smaller

- Use the smallest data type possible
  - Do you really need that BIGINT?
- The smaller your data types, the more index (and data) records can fit into a single block of memory
  - Especially important for indexed fields

Store IP addresses as INT, not CHAR

- An IP address always reduces down to an INT UNSIGNED
- Each subnet part corresponds to one 8-byte division of the underlying INT UNSIGNED
- Use INET_ATON() to convert from a string to an integer
- Use INET_NTOA() to convert from integer to string

IP address example

```
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
```

MySQL

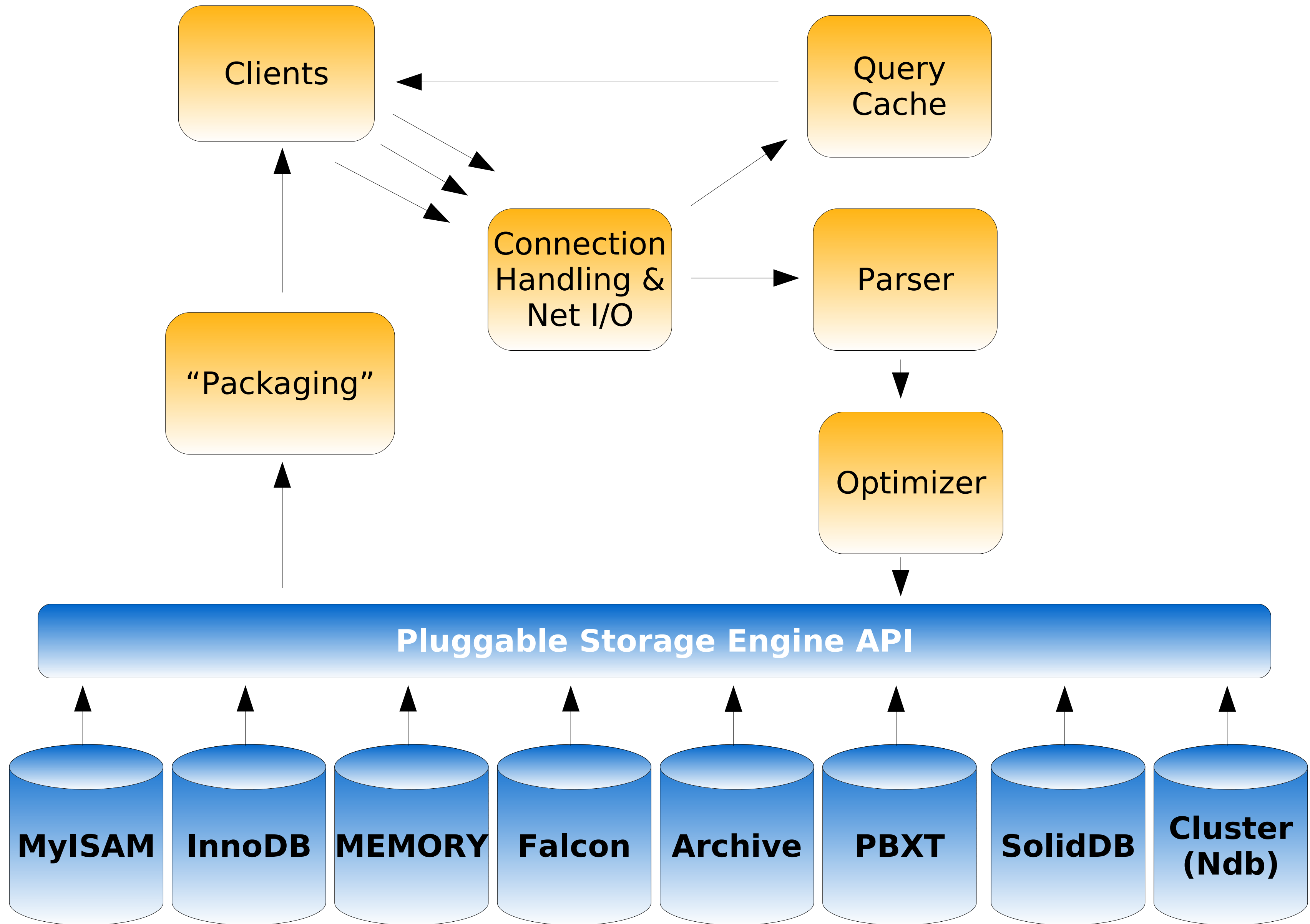
#4: Using persistent connections in PHP

- Persistent connections don't jive with a shared nothing architecture
- If you zombie a process in Apache that has a persistent connection attached, you just lost that resource
- Connections to MySQL are 10 to 100 times faster than Oracle or PostgreSQL
  - Specifically designed to be lightweight and short-lived

#5: Using a heavy DB abstraction layer

- If you don't need to worry about portability, do not use a heavy abstraction layer
  - e.g. ADODB, MDB2, PearDB, etc)
- Use a lightweight layer
  - e.g. PDO (recommended) or a homegrown wrapper if desired
  - Wrapper for scale-out support within your library

#6: Not understanding storage engines



Storage engines

- ✓ Single most mis-understood part of MySQL
- ✓ Learn both the benefits and drawbacks of each engine
- ✓ Single-engine architectures are typically not optimal
- ✓ Index → Data layout is most overlooked difference between engines

Often over-looked engines - ARCHIVE

- Incredible insert speeds
- Great compression rates (zlib)
  - Typically 6-8x smaller than MyISAM
- No UPDATEs
- Ideal for auditing and, duh, archiving
  - Web traffic records
  - CDROM bulk tables (table scans only)
  - Data that can never be updated

Often over-looked engines - MEMORY

- Data lost on server restart
  - Use `init_file` to load up the table on restart
- Allows indexes to be specified as either HASH or BTREE
- Ideal for summary and transient data
  - “Weekly top X” tables
  - Table counts for InnoDB tables
  - Data you want to “pin” in memory

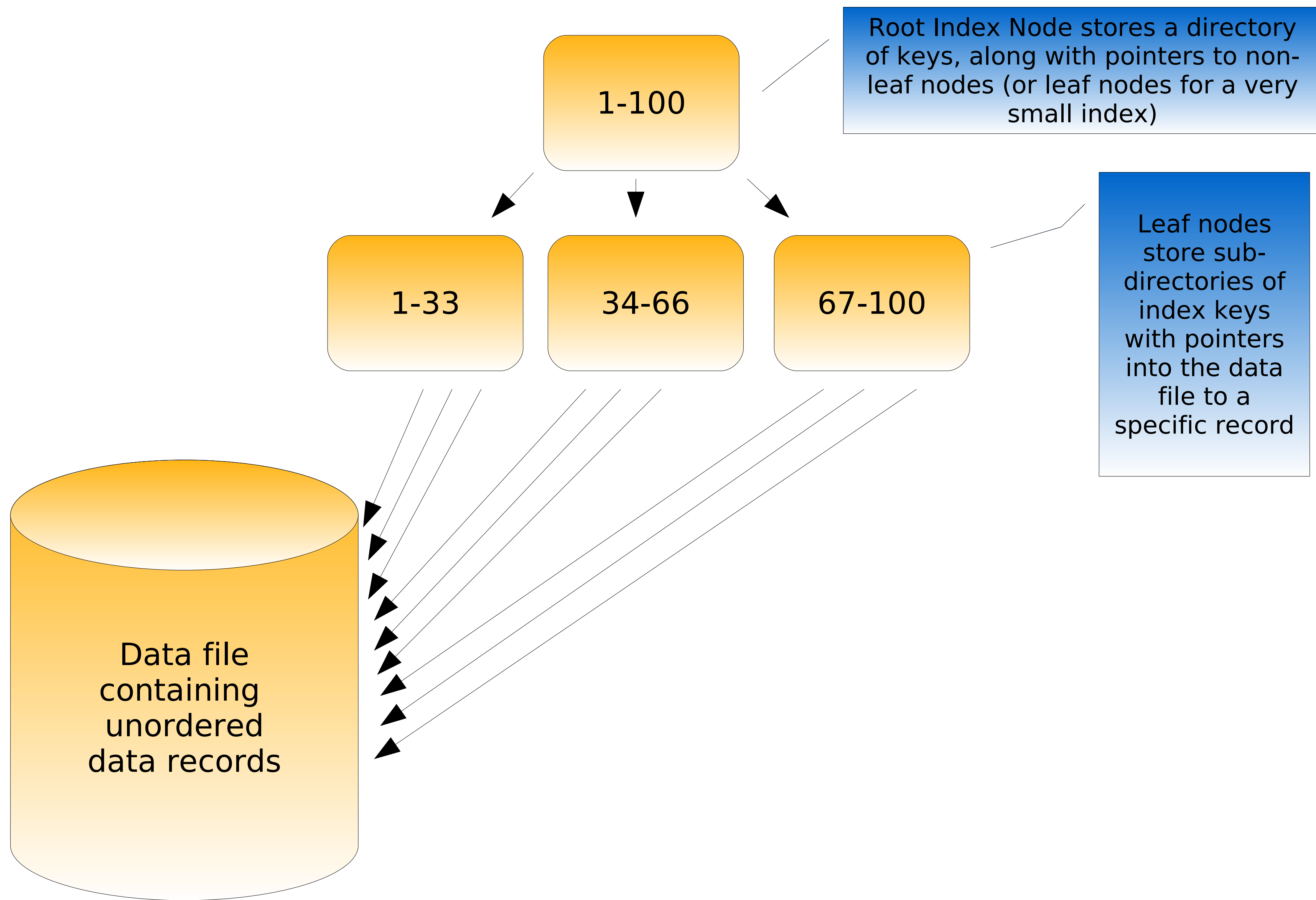
#7: Not understanding index layouts

Very important in order to make the right decisions on index and storage engine choices

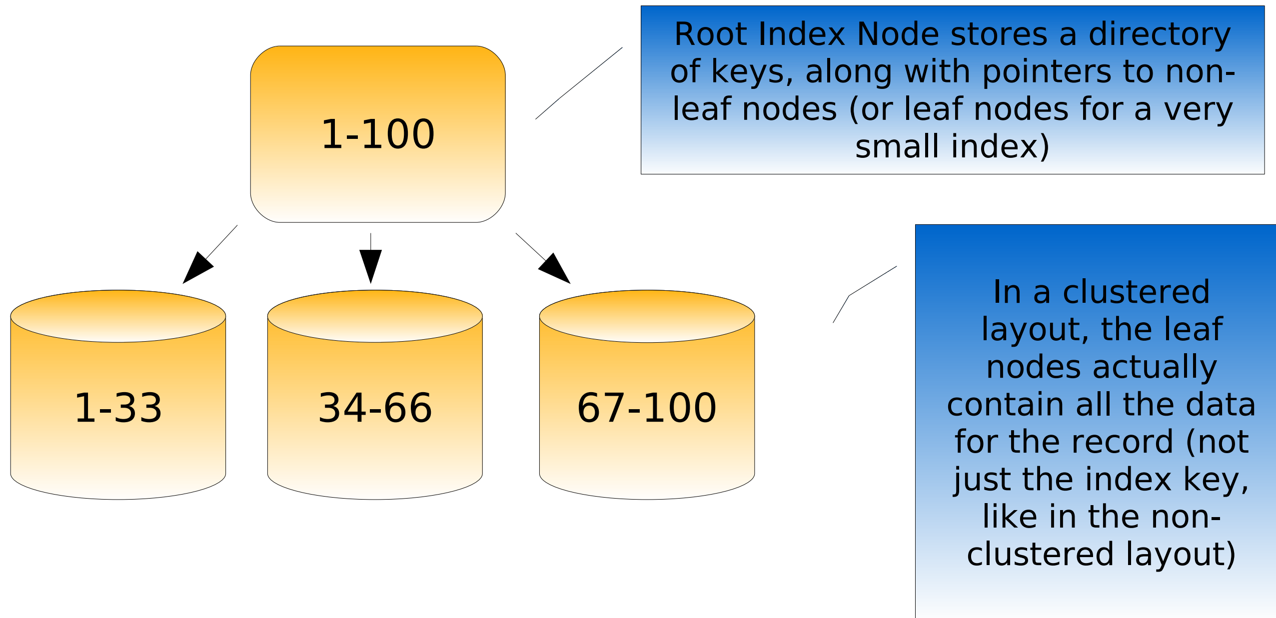
Clustered vs. Non-clustered layout

- Engines implement how they “lay out” both data and index records in memory and on disk
- A clustered organization stores it's data on disk in the order of the primary key (sort of.)
- A non-clustered organization has no implicit order to the data records, only the index records

Non-clustered layout



Clustered layout



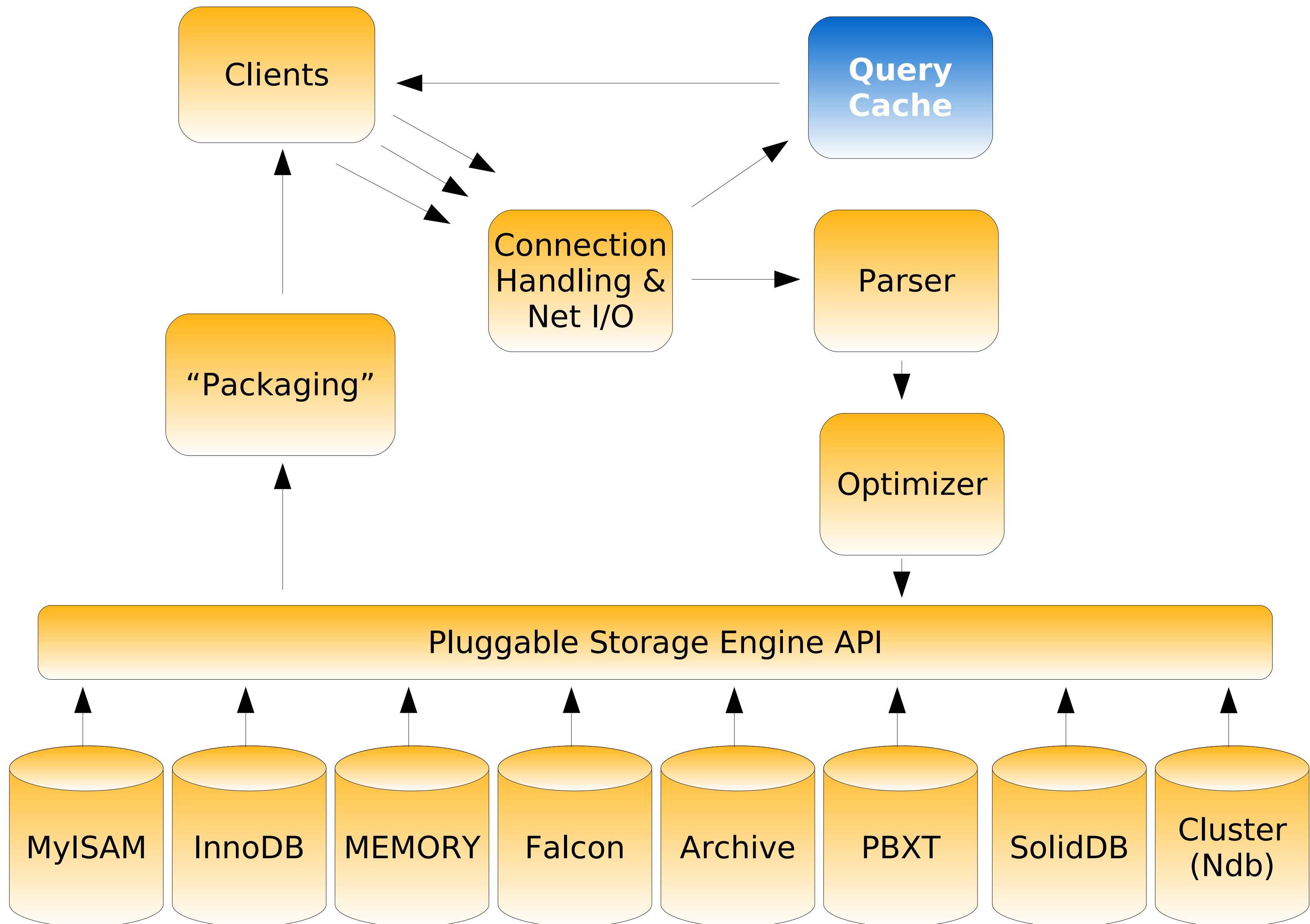
So, bottom line:

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.**

A word on clustered layouts

- 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, behind the scenes, and with you having no control over the key (this is a 6 byte number with InnoDB...)

#8: Not understanding the Query Cache



The query cache

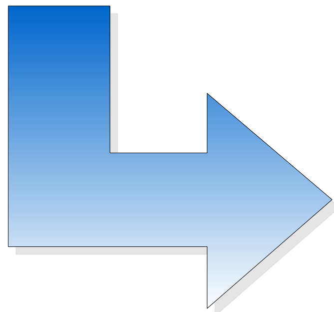
- Must understand application read/write ratio
- QC design is a compromise between CPU usage and read performance
- Bigger query cache != better performance, even for heavy read applications

Query cache invalidation

- Coarse invalidation designed to prevent CPU overuse during finding and storing cache entries
- This means ***any*** modification to ***any*** table referenced in the SELECT will invalidate ***any*** cache entry which uses that table
- Remedy with vertical table partitioning

Solving cache invalidation

```
CREATE TABLE Products (
  product_id INT UNSIGNED 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
```



```
CREATE TABLE Products (
  product_id INT UNSIGNED 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 UNSIGNED 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;
```

#9: Using stored procedures...

...without understanding what is going on behind the scenes with stored procedure compilation

The problem with stored procedures

- Unlike every other RDBMS, compiled stored procedure execution plans kept on the ***connection thread***
- This means that if you issue a stored procedure to just get data and only issue it once in a PHP page request, you're just wasting cycles (~7-8% regression)
- Solution: just use prepared statements and dynamic SQL for everything but:
  - ETL-type procedures
  - Stuff that's complex and not executed often
  - Stuff that's simple and executed multiple times ***per request***

#10: Operating on indexed column with a function

- Indexes speed up SELECTs on a column, but...
- If you operate upon that indexed column with a function (or bitwise operator, BTW), the index cannot be used
- Most of the time, there are ways to rewrite the query to isolate the indexed column on one side of the equation

Rewrite for indexed column isolation

```
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)

Rewrite for indexed column isolation #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, custo
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”;
```

#11: Having missing or useless indexes

- Indexes speed up SELECTs on a column, but only if there is a decent selectivity associated with the column
  - S = d/n
  - Number of distinct values in a column divided by the total records in the table
- But... each index will slow down INSERT, UPDATE, and DELETE operations

First, get rid of useless indexes

```
SELECT
  t.TABLE_SCHEMA
 , t.TABLE_NAME
 , s.INDEX_NAME
 , s.COLUMN_NAME
 , s.SEQ_IN_INDEX
 , (
   SELECT MAX(SEQ_IN_INDEX)
   FROM INFORMATION_SCHEMA.STATISTICS s2
   WHERE s.TABLE_SCHEMA = s2.TABLE_SCHEMA
   AND s.TABLE_NAME = s2.TABLE_NAME
   AND s.INDEX_NAME = s2.INDEX_NAME
  ) AS `COLS_IN_INDEX`
 , s.CARDINALITY AS "CARD"
 , t.TABLE_ROWS AS "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
WHERE t.TABLE_SCHEMA != 'mysql'
AND t.TABLE_ROWS > 10
AND s.CARDINALITY IS NOT NULL
AND (s.CARDINALITY / IFNULL(t.TABLE_ROWS, 0.01)) < 1.00
ORDER BY `SEL %`, TABLE_SCHEMA, TABLE_NAME
LIMIT 10;
```

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

The missing indexes

- Always have an index on join conditions
  - Nicely, if you add a foreign key constraint, you'll have one automatically
- Look to add indexes on columnd used in WHERE and GROUP BY expressions
- Look for opportunities for covering indexes
  - e.g. If you do a bunch of reads of `product_id` and `inventory_count`, consider putting an index on *both* columns (in that order)

Be aware 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**;

#12: Not being a join-fu master

Knowledge of black-belt SQL coding, including the rewriting of subqueries to standard joins and eliminating cursors through joins, is the foundation for good MySQL performance

The small things... SQL Coding

- Keep things simple
- Break complex SQL into its corresponding sets of information
- Think in terms of sets, not for-each loops!
  - For-each thinking leads to correlated subqueries (bad!)
  - Set-based thinking leads to joins (good!)

Set-based SQL thinking

"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 (bad!)
- Think instead:
  - OK, I have 2 sets of data here. One set of product names and another set of maximum sold prices

Sometimes, things look tricky...

```
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)
```

...but perform *much* better!

```
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)
```

#13: Not accounting for deep scans

Web applications with search functionality can be crippled by search engine spider deep scans

The deep scan problem

```
SELECT
  p.product_id
, p.name as product_name
, p.description as product_description
, v.name as vendor_name
FROM products p
INNER JOIN vendors v
ON p.vendor_id = v.vendor_id
ORDER BY modified_on DESC
LIMIT $offset, $count;
```

- The deep scan will put offsets in the hundreds or thousands...
- This means that the full (or close to full) data set must be returned as an ordered set, and then skipped through to the offset
  - Can get very slow, as loads of temporary tables could be created to deal with the large set sorting

Solving deep scan slowdowns

```
/*
 * Along with the offset, pass in the last key value
 * of the ordered by column in the current page of results
 * Here, we assume a “next page” link...
 */
$last_key_where= (empty($_GET['last_key'])
  ? “WHERE p.name >= '{$_GET['last_key']}' “
  : '');

$sql= “SELECT
  p.product_id
, p.name as product_name
, p.description as product_description
, v.name as vendor_name
FROM products p
INNER JOIN vendors v
ON p.vendor_id = v.vendor_id
$last_key_where
ORDER BY p.name
LIMIT $offset, $count”;

/*
 * Now you will only be retrieving a fraction of the
 * needs-to-be-sorted result set for those larger
 * offsets
 */
```

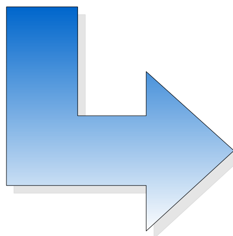
#14: SELECT COUNT(*) with no WHERE on an InnoDB table

- There is a bad performance problem when issuing a SELECT COUNT(*) on an InnoDB table when you don't specify a WHERE on an indexed column
  - i.e. Getting a count of the total number of records in the table
- The cause has to do with the complexity of the MVCC implementation which keeps a version of each record for transaction isolation

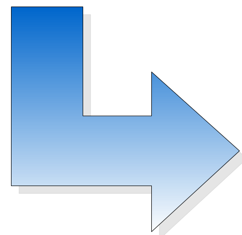
Solving InnoDB SELECT COUNT(*)

```
// Got 1M products in an InnoDB table?
// Don't do this!

SELECT COUNT(*) AS num_products
FROM products;
```



```
CREATE TABLE TableCounts (
  num_products INT UNSIGNED NOT NULL
, num_customers INT UNSIGNED NOT NULL
, num_users INT UNSIGNED NOT NULL
...
) ENGINE=MEMORY;
```

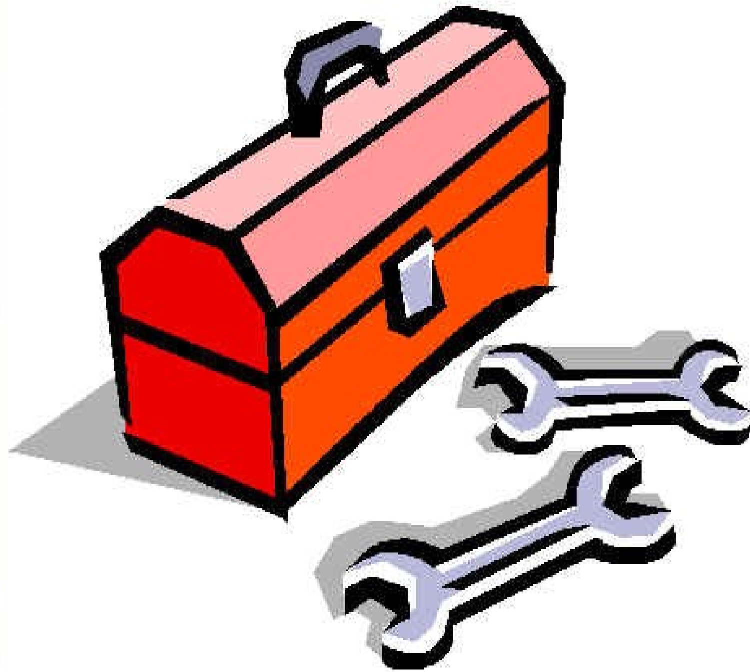


```
SELECT num_products FROM TableCounts;

// And, when modifying Products...
DELIMITER ;;
CREATE TRIGGER trg_ai_products
AFTER INSERT ON Products
UPDATE TableCounts
SET num_products = num_products +1;
END;;

CREATE TRIGGER trg_ad_products
AFTER DELETE ON Products
UPDATE TableCounts
SET num_products = num_products -1;
END;;
```

#15: Not profiling or benchmarking



Profiling is the concept of diagnosing a system for bottlenecks

Benchmarking is the process of evaluating application performance change over time and testing the load an application can withstand

Profiling concepts

- Try to profile on a testing or stage environment
  - If on a staging environment, make sure your data set is realistic!
- You are looking for bottlenecks in
  - Memory
  - Disk I/O
  - CPU
  - Network I/O and OS
- Slow query logging
  - `log_slow_queries=/path/to/log`
  - `log_queries_not_using_indexes`

Benchmarking concepts

- Track changes in application performance over time
  - Comparing the deltas after making a change
- Isolate to a single changed variable
- Record everything
  - Configuration files (my.cnf/ini)
  - SQL changes
  - Schema and indexing changes
- Shut off unnecessary programs
- Disable query cache

Your toolbox

SHOW PROFILE

MyTop/innotop

MyBench

ApacheBench (ab)

mysqlslap

super-smack

SysBench

EXPLAIN

JMeter/Ant

Slow Query Log

#16: Not using AUTO_INCREMENT

But wait, there's more!

- …SQL is highly optimized for primary …ys created as AUTO_INCREMENTing …egers
- Enables high-performance concurrent inserts
  - ✓ Lockless reading and appending
- Establishes a “hot spot” in memory and on disk which reduces swapping
- Reduces disk and page fragmentation by keeping new records together

#17: Not using ON DUPLICATE KEY UPDATE

But wait, there's even more!

- Cleans up your code
  - ✓ Prevents all that if (record_exists()) ... do_update() ... else ... do_insert()
- Avoids a round trip from connection to server
- ~5-6% faster than issuing two statements (SELECT and then INSERT or UPDATE)
- Can be even greater with large incoming data sets

Recap

1. Thinking too small
2. Not using EXPLAIN
3. Choosing the wrong data types
4. Using persistent connections in PHP
5. Using a heavy DB abstraction layer
6. Not understanding storage engines
7. Not understanding index layouts
8. Not understanding how the query cache works

Recap

9. Using stored procedures improperly
10. Operating on an indexed column with a function
11. Having missing or useless indexes
12. Not being a join-fu master
13. Not accounting for deep scans
14. Doing SELECT COUNT(*) without WHERE on an InnoDB table
15. Not profiling or benchmarking
16. Not using AUTO_INCREMENT
17. Not using ON DUPLICATE KEY UPDATE

Final thoughts

- Get involved!
  - http://forge.mysql.com
  - http://forge.mysql.com/worklog/
- MySQL Camp II
  - August 23-24
  - Brooklyn, NYC – Polytechnic University
- Grab MySQL 6.0 now and hammer it
- Email me questions and feedback please! <jay@mysql.com>