PERCONA

TRAINING

Indexes - What You Need to Know

http://www.percona.com/training/

Indexes - Need to Know

QUERY PLANNING

About This Chapter

- The number one goal is to have faster queries.
- The process is:
  - We first ask MySQL what its intended execution plan is.
  - If we don't like it, we make a change, and try again...

It All Starts with EXPLAIN

- Bookmark this manual page:
  - http://dev.mysql.com/doc/refman/5.7/en/explain-output.html
- It is the best source for anyone getting started.

Example Data

- IMDB database loaded into InnoDB tables (~5GB)
- Download it and import it for yourself using imdbpy2sql.py:
  - http://imdbpy.sourceforge.net

Table of Interest

```
CREATE TABLE title (
  id              int         NOT NULL AUTO_INCREMENT,
  title           text        NOT NULL,
  imdb_index      varchar(12) DEFAULT NULL,
  kind_id         int         NOT NULL,
  production_year int         DEFAULT NULL,
  imdb_id         int         DEFAULT NULL,
  phonetic_code   varchar(5)  DEFAULT NULL,
  episode_of_id   int         DEFAULT NULL,
  season_nr       int         DEFAULT NULL,
  episode_nr      int         DEFAULT NULL,
  series_years    varchar(49) DEFAULT NULL,
  md5sum          varchar(32) DEFAULT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Find the Title Bambi

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
    -> WHERE title = 'Bambi' ORDER BY production_year\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3331824
     filtered: 10.00
        Extra: Using where; Using filesort
1 row in set, 1 warning (0.00 sec)
```

Warning on EXPLAIN?

```
mysql> show warnings\G
 ************* 1. row *************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `imdb`.`title`.`id` AS `id`,
        `imdb`.`title`.`title` AS `title`,
        `imdb`.`title`.`production_year` AS `production_year`
        from `imdb`.`title`
        where (`imdb`.`title`.`title` = 'Bambi')
        order by `imdb`.`title`.`production_year`
 1 row in set (0.00 sec)
```

- Displays how the optimizer qualifies table and column names in the SELECT statement
- What the query looks like after rewriting and optimization rules are applied

Aha! Now Add an Index

```
mysql> ALTER TABLE title ADD INDEX (title);
ERROR 1170 (42000): BLOB/TEXT column 'title' used in key
specification without a key length
```

Aha! Now Add an Index

```
mysql> ALTER TABLE title ADD INDEX (title);
ERROR 1170 (42000): BLOB/TEXT column 'title' used in key
specification without a key length
```

```
mysql> ALTER TABLE title ADD INDEX (title(50));
Query OK, 0 rows affected (8.09 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Let's Revisit

```
mysql> EXPLAIN SELECT id, title, production_year FROM title
    -> WHERE title = 'Bambi' ORDER by production_year\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: title
          key: title
      key_len: 152
          ref: const
         rows: 4
     filtered: 100.00
        Extra: Using where; Using filesort
1 row in set, 1 warning (0.00 sec)
```

- ref is equality for comparison, but not PK lookup.
- Identified 'title' as a candidate index and chose it.
- Size of the index used.
- Anticipated number of rows.

Other Ways of Accessing

```
mysql> EXPLAIN SELECT id, title, production_year FROM title
    -> WHERE id = 55327\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

- const: at most, one matching row.
- Primary Key in InnoDB is always faster than secondary keys.

LIKE

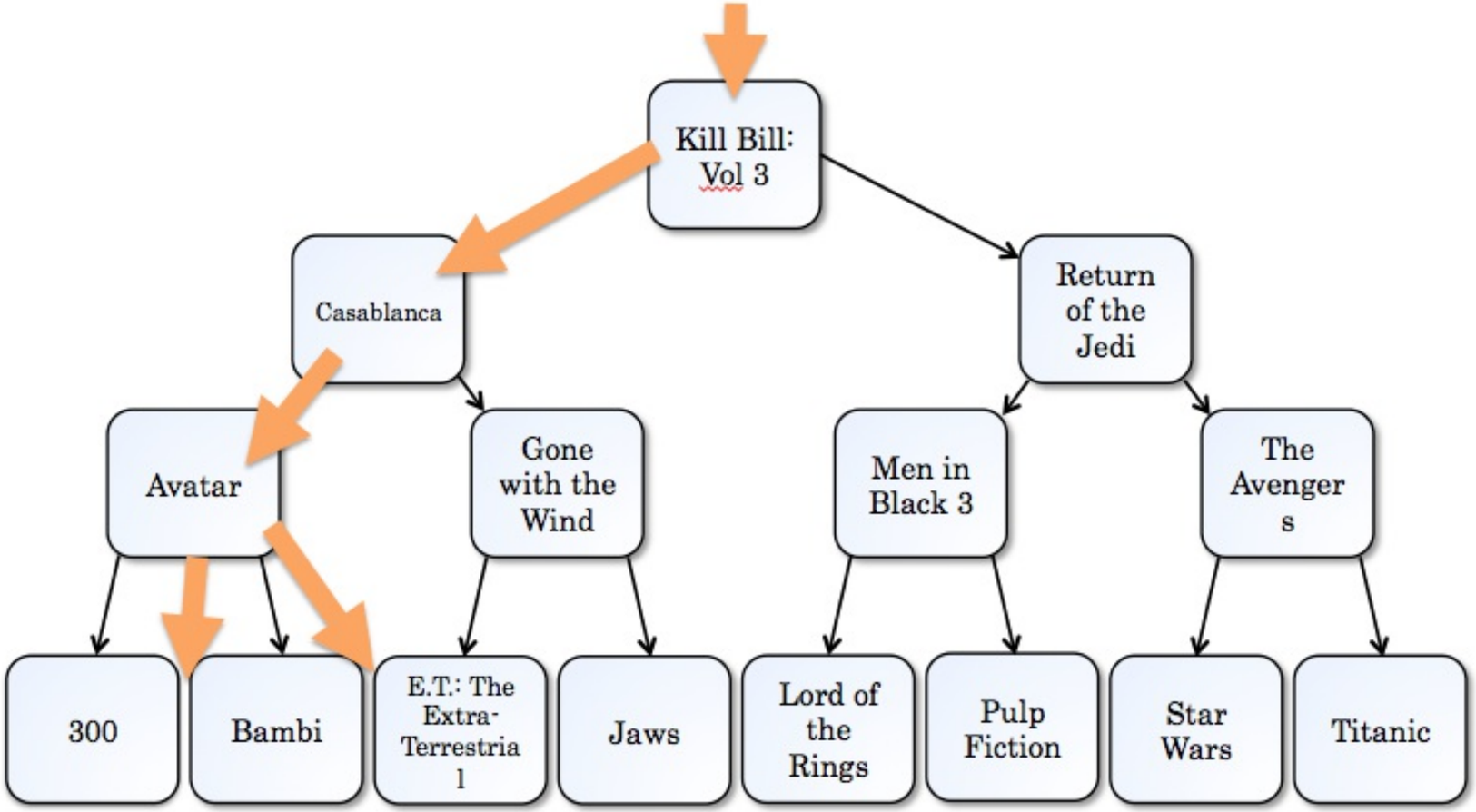
```
mysql> EXPLAIN SELECT id, title, production_year FROM title
    -> WHERE title LIKE 'Bamb%'\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: title
          key: title
      key_len: 152
          ref: NULL
         rows: 176
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

- Type is Range. BETWEEN, IN() and < > are also ranges.
- Number of rows to examine has increased; we are not specific enough.

Why is That a Range?

- We're looking for titles between BambA and BambZ*
- When we say index in MySQL, we mean trees.
  - That is, B-Tree/B+Tree/T-Tree.
  - Pretend they're all the same (for simplification).
  - There is only radically different indexing methods for specialized uses: MEMORY Hash, FULLTEXT, spatial or 3rd party engines.

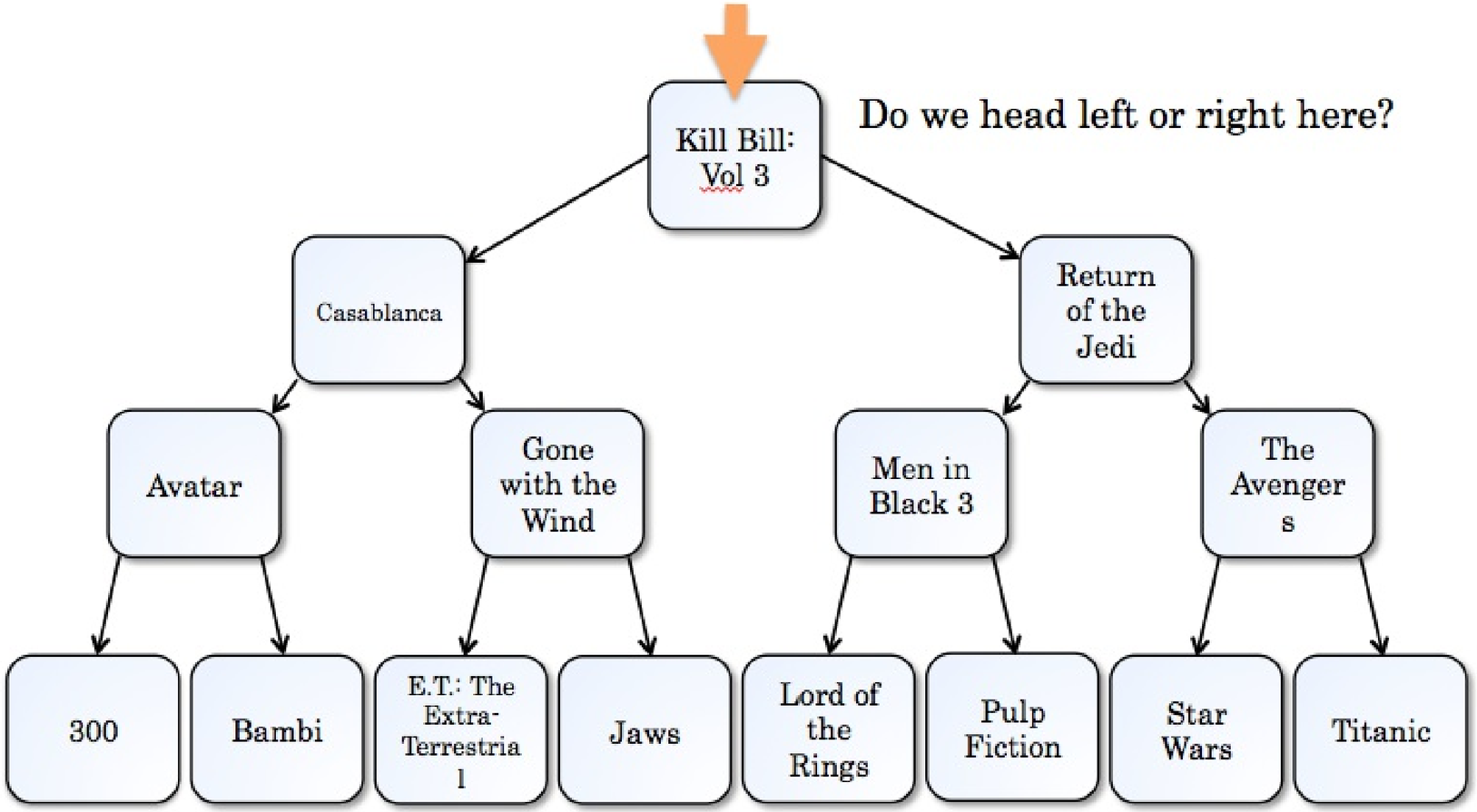
What's That?



Could This Be a Range?

```
mysql> EXPLAIN SELECT id, title, production_year FROM title
    -> WHERE title LIKE '%ulp Fiction'\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3331824
     filtered: 11.11
        Extra: Using where
1 row in set (0.00 sec)
```

No, We Can't Traverse



LIKE 'Z%'

```
mysql> EXPLAIN SELECT id, title, production_year FROM title
    -> WHERE title LIKE 'Z%'\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: title
          key: title
      key_len: 152
          ref: NULL
         rows: 24934
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

LIKE 'T%'

```
mysql> EXPLAIN SELECT id, title, production_year FROM title
    -> WHERE title LIKE 'T%'\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: title
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3331824
     filtered: 21.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

MySQL is Reasonably Smart

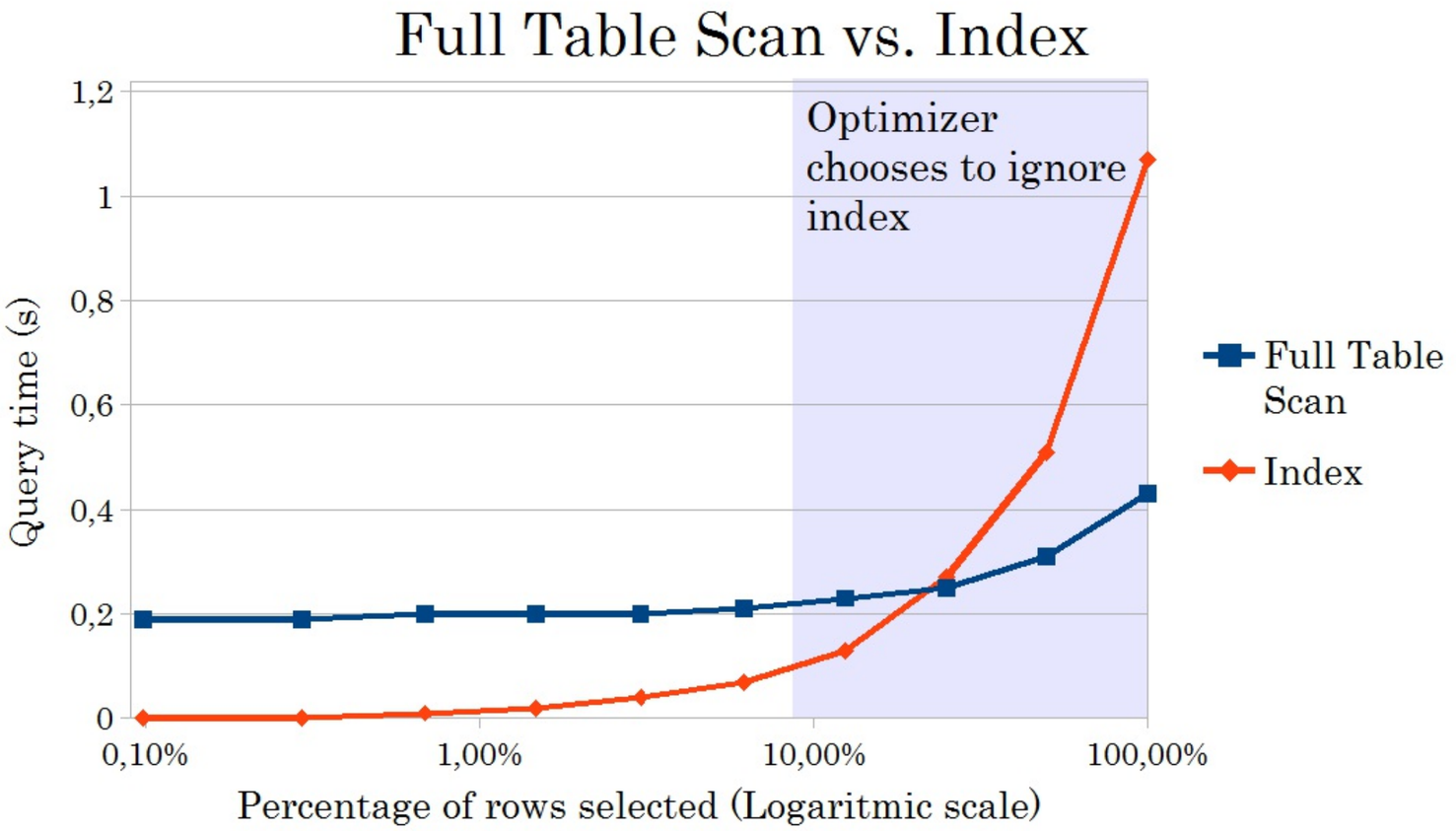
- It dynamically samples the data to choose which is the better choice—or in some cases uses static statistics.
- This helps the optimizer choose:
  - Which indexes will be useful.
  - Which indexes should be avoided.
  - Which is the better index when there is more than one.

Why Avoid Indexes?

- B-Trees work like humans search a phone book;
  - Use an index if you want just a few rows.
  - Scan cover-to-cover if you want a large percentage.

Why Avoid Indexes (cont.)

- Benchmark on a different schema (lower is better):



What You Should Take Away

- Data is absolutely critical.
  - Development environments should contain sample data exported from production systems.
  - A few thousands of rows is usually enough for the optimizer to behave like it does in production.

What You Should Take Away (cont.)

- Input values are absolutely critical.
  - Between two seemingly identical queries, execution plans may be very different.
  - Just like you test application code functions with several values for input arguments.

Indexes - Need to Know

EXPLAINING THE EXPLAIN

How to Explain the EXPLAIN

- In queries with regular joins, tables are read in the order displayed by EXPLAIN.
- *id* is a sequential identifier of SELECT statements in the query.
- *select_type* indicates type of SELECT (simple, primary, subquery, union, derived, ...).
- *type* says which join type will be used.
- *possible_keys* indicates which indexes MySQL can choose from to find the rows in this table.
- *key* indicates which index is used.
- *partitions* shows which partitions are being accessed.

How to Explain the EXPLAIN (cont.)

- *key_len* longest length of the key that was used (which parts of a composite index are being used). (http://bugs.mysql.com/bug.php?id=83062)
- *ref* which columns or constants are compared to the index to select rows from the table.
- *filtered* shows the estimated percentage of table rows that will be filtered by the table condition.
- *rows* says how many rows have to be examined in order to execute each step of the query.
- *Extra* contains additional information about how MySQL resolves the query

http://dev.mysql.com/doc/refman/5.7/en/explain-output.html#explain-extra-information

Types in EXPLAIN

- The following slides show possible values for EXPLAIN type, ordered (approximately) from the fastest to the slowest.
  - FULLTEXT access type (and its special indexes) are not covered on this section.

NULL

- Not really a plan: no data is returned
- See 'Extra' for a reason

```
mysql> EXPLAIN SELECT * FROM title WHERE 1 = 2\G

************* 1. row *************
   select_type: SIMPLE
         table: NULL
          type: NULL
 possible_keys: NULL
           key: NULL      -- Internally equivalent to
       key_len: NULL      -- SELECT NULL WHERE 0;
           ref: NULL
          rows: NULL
      filtered: NULL
         Extra: Impossible WHERE
1 row in set, 1 warning (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM title WHERE id = -1\G
       ...
       type: NULL
       Extra: no matching row in const table
```

const

- Used when comparing a literal with a non-prefix PRIMARY/UNIQUE index.
- The table has at the most one matching row, which will be read at the start of the query.
- Because there is only one row, the values can be regarded as constants by the optimizer. *This is very fast since table is read only once.

const (cont.)

```
mysql> EXPLAIN SELECT * FROM title WHERE id = 55327\G

*************** 1. row ***************
           id: 1
  select_type: SIMPLE
        table: title
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

eq_ref

- One row will be read from this table for each combination of rows from the previous tables.
- The best possible join type (after const).
- Used when the whole index is used for the = operator with a UNIQUE or PRIMARY KEY.

eq_ref (cont.)

```
mysql> EXPLAIN SELECT title.title, kind_type.kind
    -> FROM kind_type JOIN title ON kind_type.id = title.kind_id
    -> WHERE title.title = 'Bambi'\G
```

```
*************** 1. row ***************
           id: 1
  select_type: SIMPLE
        table: title
   partitions: NULL
         type: ref
possible_keys: title
          key: title
      key_len: 152
          ref: const
         rows: 11
     filtered: 100.00
        Extra: Using where
```

```
*************** 2. row ***************
           id: 1
  select_type: SIMPLE
        table: kind_type
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.title.kind_id
         rows: 1
     filtered: 100.00
        Extra: NULL
```

ref

- Several rows will be read from this table for each combination of rows from the previous tables.
- Used if the join uses only a left-most prefix of the index, or if the index is not UNIQUE or PRIMARY KEY.
- Still not bad, if the index matches only few rows.

ref (cont.)

```
mysql> ALTER TABLE users ADD INDEX (first_name);
mysql> EXPLAIN SELECT distinct u1.first_name FROM users u1 JOIN users u2
    -> WHERE u1.first_name = u2.first_name and u1.id <> u2.id\G
```

```
************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: u1
         type: index
possible_keys: first_name
          key: first_name
      key_len: 102
          ref: NULL
         rows: 49838
     filtered: 100.00
        Extra: Using index;
               Using temporary
```

```
************* 2. row *************
           id: 1
  select_type: SIMPLE
        table: u2
         type: ref
possible_keys: first_name
          key: first_name
      key_len: 102
          ref: imdb.u1.first_name
         rows: 14
     filtered: 90.00
        Extra: Using where;
               Using index;
               Distinct
```

- Can you think of a more efficient way of writing this query?

A More Efficient Query

```
mysql> EXPLAIN SELECT first_name FROM users GROUP BY first_name
    -> HAVING count(first_name) > 1;

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: users
         type: index
possible_keys: first_name
          key: first_name
      key_len: 102
          ref: NULL
         rows: 49873
     filtered: 100.00
        Extra: Using index
1 row in set, 1 warning (0.00 sec)
```

ref_or_null

- This is a join type, like *ref*, but with the addition that MySQL does an extra search for rows that contain NULL values.
- This join type optimization is used most often in resolving subqueries.

ref_or_null (cont.)

```
mysql> ALTER TABLE cast_info ADD INDEX (nr_order);
mysql> EXPLAIN SELECT * FROM cast_info
    -> WHERE nr_order = 1 or nr_order IS NULL\G

*************** 1. row ***************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref_or_null
possible_keys: nr_order
          key: nr_order
      key_len: 5
          ref: const
         rows: 26707053
     filtered: 100.00
        Extra: Using index condition
1 row in set, 1 warning (0.00 sec)
```

index_merge

- Results from more than one index are combined either by intersection or union.
- In this case, the key column contains a list of indexes.

```
mysql> ALTER TABLE title ADD INDEX (production_year);
mysql> EXPLAIN SELECT * FROM title
    -> WHERE title = 'Dracula' OR production_year = 1922\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: index_merge
possible_keys: production_year,title
          key: title,production_year
      key_len: 152,5
          ref: NULL
         rows: 3503
     filtered: 100.00
        Extra: Using sort_union(title,production_year); Using where
1 row in set, 1 warning (0.00 sec)
```

unique_subquery/index_subquery

- unique_subquery
  - The result of a subquery is covered by a unique index.
  - The subquery is used within an IN(...) predicate.
- index_subquery
  - Similar to unique_subquery, only allowing for non-unique indexes.

[indexlunique]_subquery (cont.)

```
mysql> SET optimizer_switch='materialization=off';
mysql> EXPLAIN SELECT * FROM title WHERE title = 'Bambi'
    -> AND kind_id NOT IN
    ->  (SELECT id FROM kind_type WHERE kind like 'tv%')\G
```

```
************ 1. row ************
           id: 1
  select_type: PRIMARY
        table: title
         type: ref
possible_keys: title
          key: title
      key_len: 152
          ref: const
         rows: 11
     filtered: 100.00
        Extra: Using where
************ 2. row ************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: kind_type
         type: unique_subquery
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: func
         rows: 1
     filtered: 42.86
        Extra: Using where
2 rows in set, 1 warning (0.00 sec)
```

For *index_subquery*, use a non-PRIMARY, non-UNIQUE key

range

- Only rows that are in a given range will be retrieved.
- An index will still be used to select the rows
- The *key_len* contains the longest key part that is used. (http://bugs.mysql.com/bug.php?id=83062)
- The *ref* column will be NULL for this type.

range (cont.)

```
mysql> EXPLAIN SELECT * FROM title
    -> WHERE title = 'Bambi' OR title = 'Dumbo'
    -> OR title = 'Cinderella'\G

*************** 1. row ***************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: title
          key: title
      key_len: 152
          ref: NULL
         rows: 90
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

range (cont.)

```
mysql> EXPLAIN SELECT * FROM title
    -> WHERE title like 'Bamb%';
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: title
          key: title
      key_len: 152
          ref: NULL
         rows: 176
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

range (cont.)

```
mysql> EXPLAIN SELECT * FROM title
    -> WHERE production_year BETWEEN 1998 AND 1999\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: production_year
          key: production_year
      key_len: 5
          ref: NULL
         rows: 217402
     filtered: 100.00
        Extra: Using index condition
1 row in set, 1 warning (0.00 sec)
```

index

- The whole index tree is scanned.
- Otherwise same as *ALL*.
- Faster than ALL since the index file is (should be) smaller than the data file.
- MySQL can use this join type when the query uses only columns that are part of a single index.

index (cont.)

```
mysql> EXPLAIN SELECT count(*), production_year,
    -> GROUP_CONCAT(DISTINCT kind_id ORDER BY kind_id) as kind_id
    -> FROM title
    -> GROUP BY production_year ORDER BY production_year\G

*************** 1. row ***************
           id: 1
  select_type: SIMPLE
        table: title
         type: index
possible_keys: production_year
          key: production_year
      key_len: 5
          ref: NULL
         rows: 3244766
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

- "How many releases per year, and what are their types"

ALL

- A full table scan; the entire table is scanned.
- Not good even for the first (non-const) table.
- Very bad for subsequent tables, since it means a full table scan for each combination of rows from the previous tables is performed.
- Solutions: rephrase query, add more indexes.

ALL (cont.)

```
mysql> ALTER TABLE title ADD INDEX (production_year);
mysql> EXPLAIN SELECT * from title
    -> WHERE MAKEDATE(production_year, 1) >= now() - INTERVAL 1 YEAR\G

************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3244766
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

- An index exists on *production_year*. Whats going on?
- What is a better solution?

Much Better

```
mysql> EXPLAIN SELECT * from title
    -> WHERE production_year >= YEAR(NOW() - INTERVAL 1 YEAR)\G
************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: title
   partitions: NULL
         type: range
possible_keys: production_year
          key: production_year
      key_len: 5
          ref: NULL
         rows: 205352
     filtered: 100.00
        Extra: Using index condition
1 row in set, 1 warning (0.00 sec)
```

- *Rule of Thumb*: Don't manipulate data already stored

Extra: What You Would Like to See

- Using index
  - Excellent! MySQL can search for the rows directly from the index tree, without reading the actual table (covering index).
- Distinct
  - Good! MySQL stops searching for more rows for the combination after it has found the first matching row.
- Not exists
  - Good! MySQL is able to do a LEFT JOIN optimization, and some rows can be left out.
- Using index condition
  - Tables are read by accessing the index and testing to determine whether to read the full rows. Index information is used to defer ("push down") reading full table rows unless it is necessary.

Extra: What You Don't Like to See

- Using filesort
  - Extra sorting pass needed! (Does not imply a file created on disk!)
- Using temporary
  - Temporary table needed! (Does not imply temp table on disk.)
  - Typically happens with different ORDER BY and GROUP BY
- Using join buffer
  - Tables are processed in large batches of rows, instead of by indexed lookups.
- Range checked for each record (index map: N)
  - No good index found for direct comparisons.
  - Individual records are separately optimized for index retrieval.
  - This is not fast, but faster than a join with no index at all.

PERCONA
TRAINING

Any Questions?