MySQL Query Optimization

Kenny Gryp <kenny.gryp@percona.com>
Percona Live Washington DC / 2012-01-11

MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries

MySQL Query Optimization

- 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...
- More Information:
  - High Performance MySQL, 2nd Edition: http://shop.oreilly.com/product/9780596101718.do
  - `EXPLAIN`! http://dev.mysql.com/doc/refman/5.5/en/using-explain.html

MySQL Query Optimization

- **Basic Query Tuning**
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries

First Example

```
1.  CREATE TABLE `title` (
2.    `id` int(11) NOT NULL AUTO_INCREMENT,
3.    `title` text NOT NULL,
4.    `imdb_index` varchar(12) DEFAULT NULL,
5.    `kind_id` int(11) NOT NULL,
6.    `production_year` int(11) DEFAULT NULL,
7.    `imdb_id` int(11) DEFAULT NULL,
8.    `phonetic_code` varchar(5) DEFAULT NULL,
9.    `episode_of_id` int(11) DEFAULT NULL,
10.   `season_nr` int(11) DEFAULT NULL,
11.   `episode_nr` int(11) DEFAULT NULL,
12.   `series_years` varchar(49) DEFAULT NULL,
13.   PRIMARY KEY (`id`)
14. ) ENGINE=InnoDB AUTO_INCREMENT=1543721;
```

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: 1900087
        Extra: Using where; Using filesort;
1 row in set (0.00 sec)
```

3.09s

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: 1900087
        Extra: Using where; Using filesort;
1 row in set (0.00 sec)
```

ALL means tablescan

In this case a sort is required because of the order by, but not for all rows, only matching rows

Additional filtering may be possible before passing to sort.

Specify Index Length

```
1.  mysql> ALTER TABLE title ADD INDEX (title);
2.  ERROR 1170 (42000): BLOB/TEXT column 'title' used in key
3.  specification without a key length
4.
5.
6.  mysql> ALTER TABLE title ADD INDEX (title(50));
7.  Query OK, 1543719 rows affected (1 min 19.14 sec)
8.  Records: 1543719  Duplicates: 0  Warnings: 0
```

Size of Key?

- Length of index is limited:
  - 1000 bytes MyISAM
  - 767 bytes InnoDB
- UTF-8 uses up to 3 bytes (3 bytes are used in determining)
- Variable length strings (VARCHAR, TEXT...): add 1 or 2 bytes for length

We must revisit...

```
1.  mysql> EXPLAIN SELECT id,title,production_year FROM title
2.  WHERE title  = 'Bambi' ORDER by production_year\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: ref
8.  possible_keys: title
9.            key: title
10.        key_len: 152
11.            ref: const
12.           rows: 4
13.          Extra: Using where; Using filesort;
14.  1 row in set (0.14 sec)
```

Much Faster!

We must revisit...

```
1.  mysql> EXPLAIN SELECT id,title,production_year FROM title
2.  WHERE title  = 'Bambi' ORDER by production_year\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: ref
8.  possible_keys: title
9.            key: title
10.       key_len: 152
11.           ref: const
12.          rows: 4
13.         Extra: Using where; Using filesort;
14.  1 row in set (0.14 sec)
```

Using = for comparison, but not primary key lookup.

Identified title as a candidate index, chose to use it.

Size of the index used (in bytes)

Anticipated number of rows to be examined dropped considerably.

Other ways of accessing

```
1.  mysql> EXPLAIN SELECT id,title,production_year FROM title
2.  WHERE id = 55327\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: const
8.  possible_keys: PRIMARY
9.            key: PRIMARY
10.       key_len: 4
11.           ref: const
12.          rows: 1
13.         Extra:
14.  1 row in set (0.06 sec)
```

At most one matching row.

In InnoDB the primary key is often much faster than all other keys.

Range Scan

```
1.  mysql> EXPLAIN SELECT id,title,production_year FROM title
2.  WHERE title LIKE 'Bamb%'\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: range
8.  possible_keys: title
9.            key: title
10.       key_len: 152
11.           ref: NULL
12.          rows: 98
13.         Extra: Using where
14.  1 row in set (0.00 sec)
```

Type is range. BETWEEN, IN() and < > are also ranges.

Anticipated number of rows to be examined has increased - we are not specific enough.

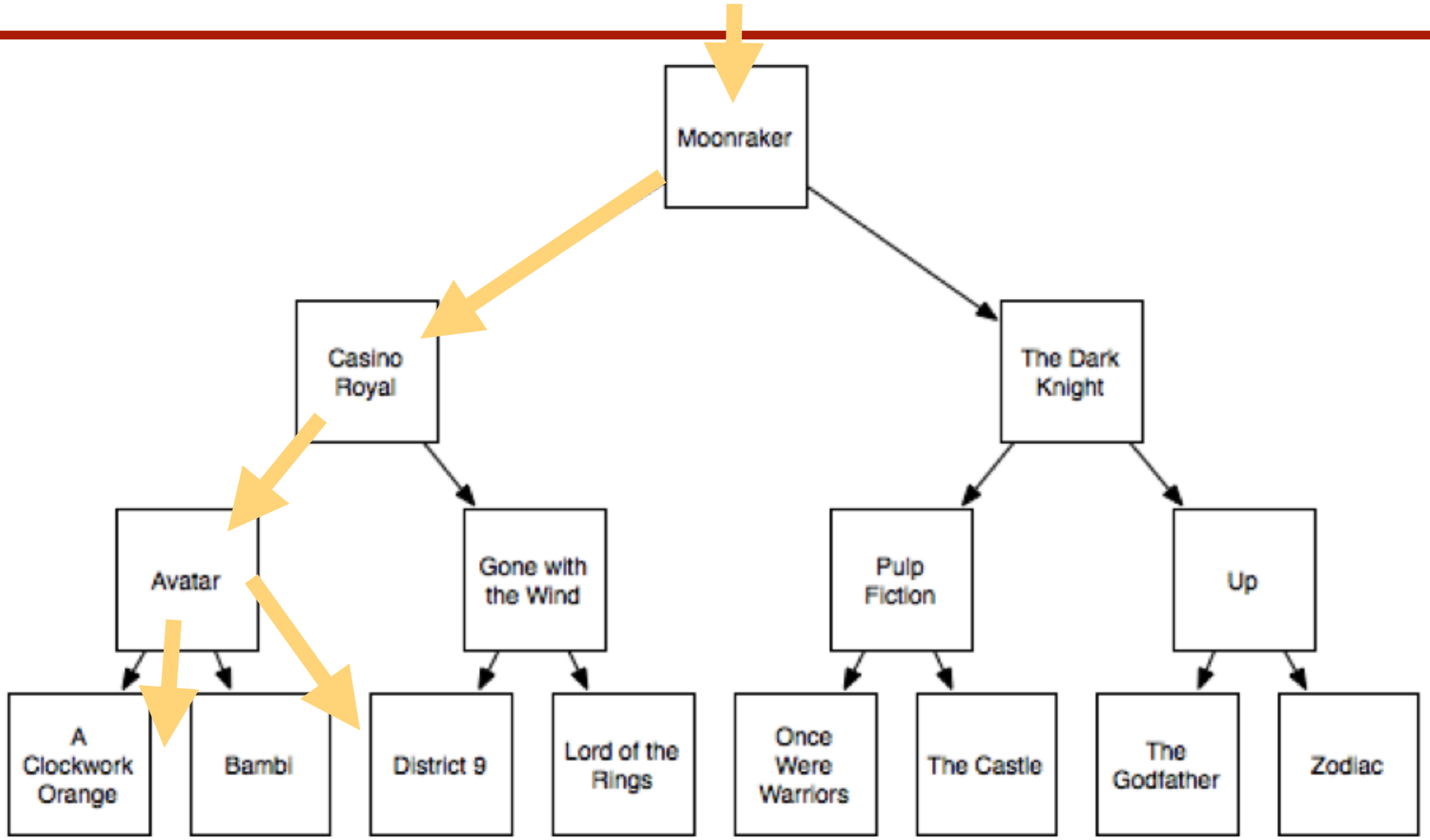
Ignore the time with EXPLAIN. Only look at the time for a query.

Why's that a range?

- We're looking for titles between Bamb***A*** and Bamb***Z****
- 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's no radically different indexing methods in MySQL unless you play storage engine Bingo**.

* In reality the range is a little wider
** The memory & ndb storage engine supports hash indexes

What's that?

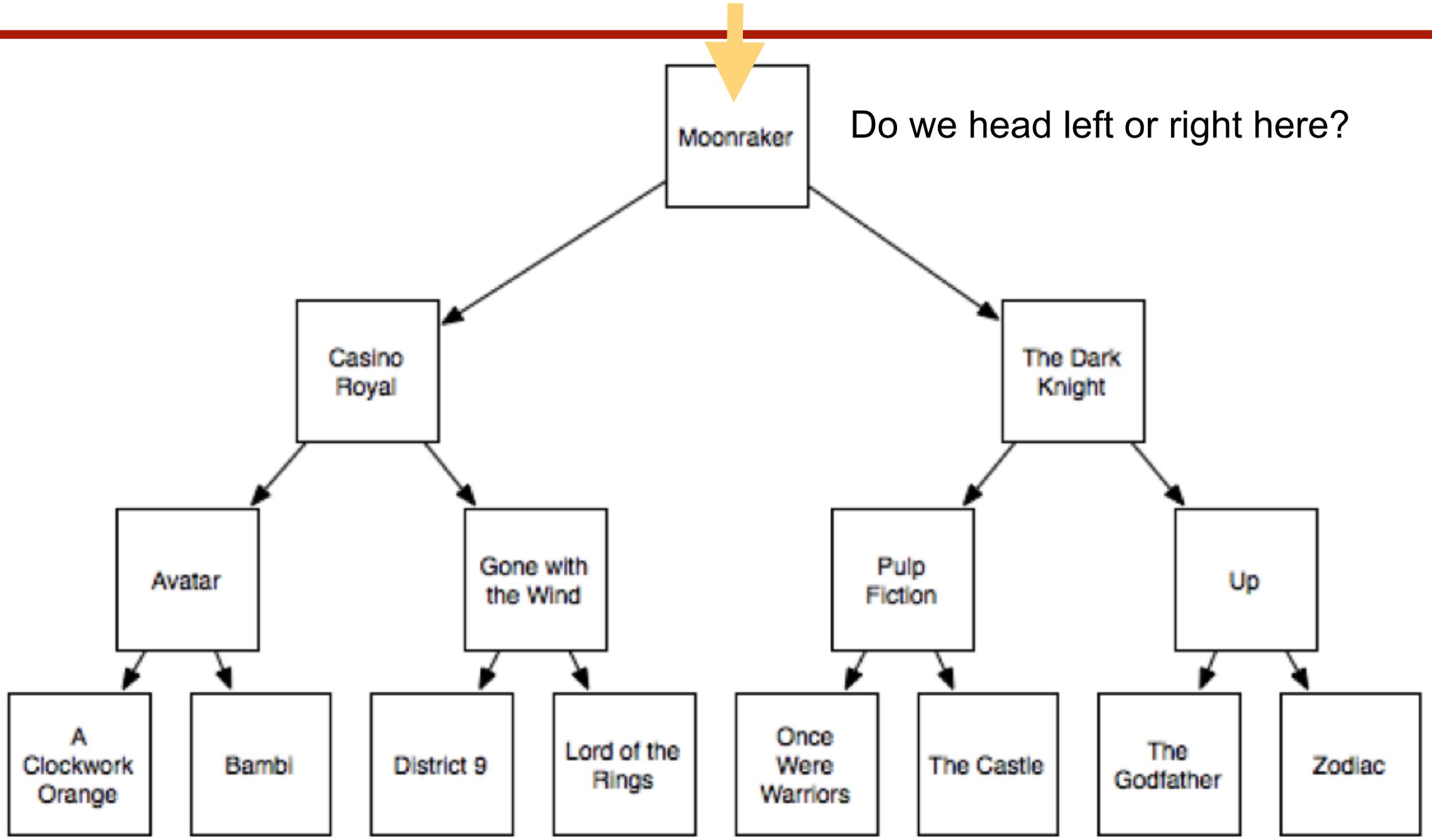

Moonraker
Casino Royal
The Dark Knight
Avatar
Gone with the Wind
Pulp Fiction
Up
A Clockwork Orange
Bambi
District 9
Lord of the Rings
Once Were Warriors
The Castle
The Godfather
Zodiac

Could this be a range?

```
1.  mysql> EXPLAIN SELECT id,title,production_year FROM title
2.  WHERE title LIKE '%ulp Fiction'\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: ALL
8.  possible_keys: NULL
9.            key: NULL
10.       key_len: NULL
11.           ref: NULL
12.          rows: 1900087
13.         Extra: Using where;
14.  1 row in set (0.00 sec)
```

3.2s

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: 9420
        Extra: Using where
1 row in set (0.00 sec)
```

LIKE ‘T%’

```
1.  mysql> EXPLAIN SELECT id,title,production_year FROM title
2.  WHERE title LIKE 'T%'\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: ALL
8.  possible_keys: title
9.            key: NULL
10.       key_len: NULL
11.           ref: NULL
12.          rows: 1492490
13.         Extra: Using where
14.  1 row in set (0.00 sec)
```

3.13s

LIKE ‘The %’

```
1.  mysql> EXPLAIN SELECT id,title,production_year FROM title
2.  WHERE title LIKE 'The %'\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: ALL
8.  possible_keys: title
9.            key: NULL
10.       key_len: NULL
11.           ref: NULL
12.          rows: 1492490
13.         Extra: Using where
14.  1 row in set (0.00 sec)
```

3.07s

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.

* To refresh statistics run ANALYZE TABLE table_name;

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.)

- We benchmarked this on a different schema:

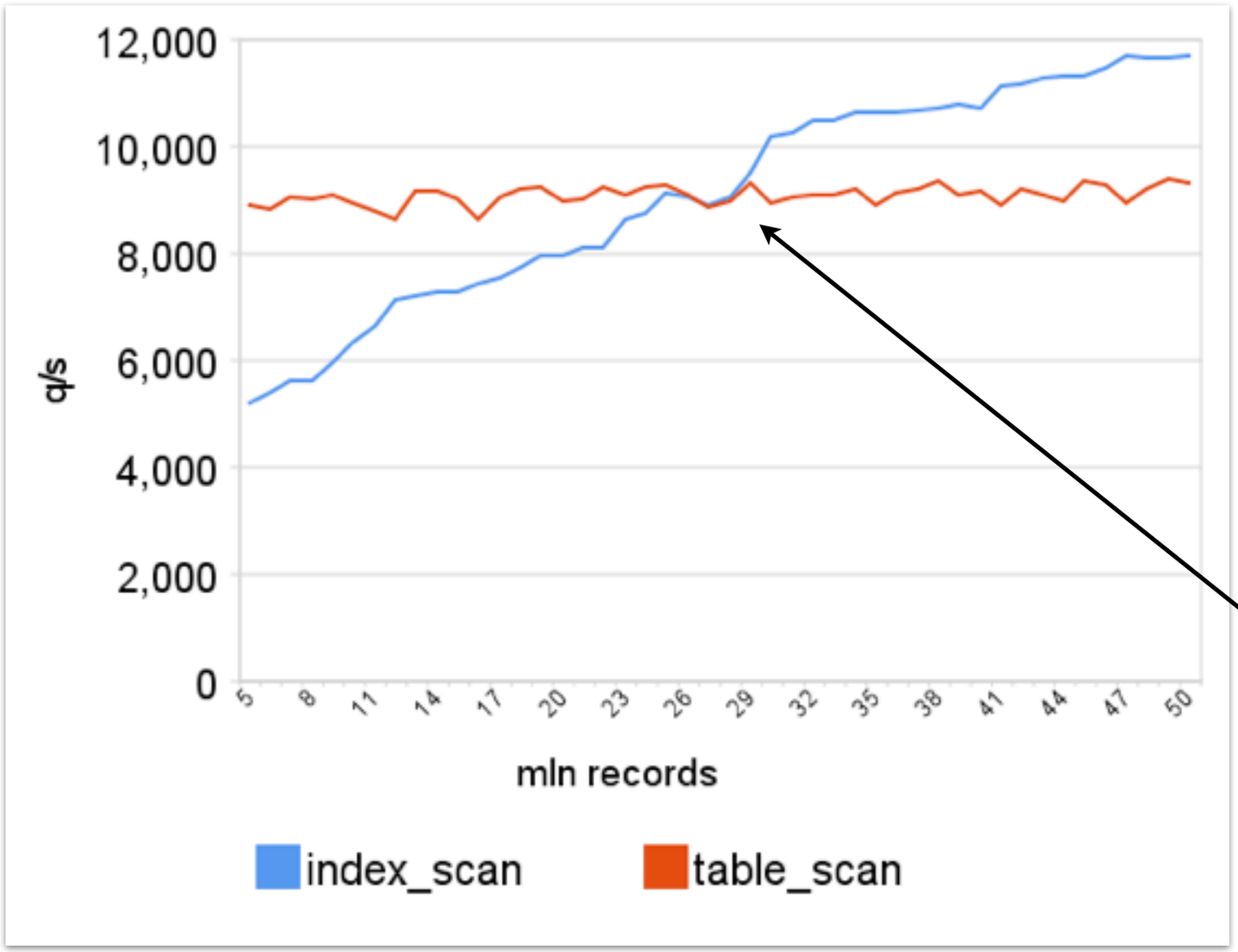


Table scan has a relatively fixed cost (red line).

The index has completely different effectiveness depending on how much it can filter.

Hopefully MySQL switches at the right point.

What you should take away:

- **Data is absolutely critical.**
  - Development environments should contain sample data exported from production systems.
- **Input values are absolutely critical.**
  - Between two seemingly identical queries, execution plans may be *very* different.

See also: http://www.mysqlperformanceblog.com/2009/10/16/how-not-to-find-unused-indexes/

Anticipated Number Of Rows

```
1.  mysql> ALTER TABLE title DROP INDEX title;
2.  mysql> EXPLAIN SELECT * FROM title WHERE title = 'Pilot' AND
3.  production_year BETWEEN 1997 and 2009\G
4.  *************************** 1. row ***************************
5.             id: 1
6.    select_type: SIMPLE
7.          table: title
8.           type: ALL
9.  possible_keys: NULL
10.           key: NULL
11.       key_len: NULL
12.           ref: NULL
13.          rows: 1569823
14.         Extra: Using where
15.  1 row in set (0.00 sec)
```

This number of rows is a guess. It keeps changing between examples.

Statistics Sampling

- InnoDB only keeps statistics samples in memory - and not on disk*.
- Sampling is performed when a table is first opened, and estimates are based on an estimate from sampling 8 random pages.
  - This number is used whether the table have 10 rows or 10 million rows.

In InnoDB plugin this is now configurable with `innodb_stats_sample_pages`. The setting is global, and will apply to all tables.

* In XtraDB 12 statistics can be retained with innodb_use_sys_stats_table=1.
* MySQL 5.6: innodb_stats_persistent_sample_pages

Statistics (cont.)

- Statistics are automatically regenerated on most meta-data commands:
  - SHOW TABLE STATUS
  - SHOW INDEX
  - Information Schema commands.

Disable with `innodb_stats_on_metadata=0` (5.1 and above).

- As well as:
  - When the table size changes by more than 1/16th.
  - If more than 2,000,000,000 rows have been inserted.

Disable with `innodb_stats_auto_update=0` (XtraDB only).

Improve this Query

```
mysql> ALTER TABLE title DROP INDEX title;
mysql> EXPLAIN SELECT * FROM title WHERE title = 'Pilot' AND
production_year BETWEEN 1997 and 2009\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1569823
        Extra: Using where
1 row in set (0.00 sec)
```

We're Spoiled for Choice.

```
1.  # Which one is best?
2.  # ALTER TABLE title ADD INDEX py (production_year);
3.  # ALTER TABLE title ADD INDEX t (title);
4.  # ALTER TABLE title ADD INDEX py_t (production_year, title);
5.  # ALTER TABLE title ADD INDEX t_py (title, production_year);
6.
7.  # Start by trying the production_year example:
8.  mysql> ALTER TABLE title ADD INDEX py (production_year);
9.  Query OK, 1543719 rows affected (38.07 sec)
10. Records: 1543719  Duplicates: 0  Warnings: 0
```

Index on production_year

```
1.  mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
2.  AND production_year BETWEEN 1997 and 2009\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: ALL
8.  possible_keys: py
9.            key: NULL
10.       key_len: NULL
11.           ref: NULL
12.          rows: 1592559
13.         Extra: Using where
14.  1 row in set (0.02 sec)
```

3.53s

Might work if...

```
1.  mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
2.  AND production_year BETWEEN 2006 and 2009\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: range
8.  possible_keys: py
9.            key: py
10.        key_len: 5
11.            ref: NULL
12.           rows: 148320
13.          Extra: Using where
14.  1 row in set (0.00 sec)
```

0.92s

Index on title(50)

```
mysql> ALTER TABLE title ADD INDEX t (title(50));
Query OK, 1543719 rows affected (38.07 sec)
Records: 1543719  Duplicates: 0  Warnings: 0
mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
AND production_year BETWEEN 2006 and 2009\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: py,t
          key: t
      key_len: 152
          ref: const
         rows: 926
        Extra: Using where
1 row in set (0.00 sec)
```

0.02s

Comparing the two:

* `mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot' AND production_year BETWEEN 2006 and 2009\G`

```
1.           id: 1
2.  select_type: SIMPLE
3.        table: title
4.         type: range
5. possible_keys: py
6.          key: py
7.      key_len: 5
8.          ref: NULL
9.         rows: 148320
10.       Extra: Using where
11. 1 row in set (0.00 sec)
```

```
1.           id: 1
2.  select_type: SIMPLE
3.        table: title
4.         type: ref
5. possible_keys: py,t
6.          key: t
7.      key_len: 152
8.          ref: const
9.         rows: 926
10.       Extra: Using where
11. 1 row in set (0.00 sec)
```

Composite Indexes.

- What is better?
  - `INDEX py_t (production_year, title)`
  - `INDEX t_py (title, production_year)`

Index on py_t

```
mysql> ALTER TABLE title ADD INDEX py_t
(production_year, title(50));
Query OK, 1543719 rows affected (2 min 15.64 sec)
Records: 1543719  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
AND production_year BETWEEN 2006 and 2009\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: py,t,py_t
          key: t
      key_len: 152
          ref: const
         rows: 926
        Extra: Using where
```

http://www.mysqlperformanceblog.com/2010/01/09/getting-around-optimizer-limitations-with-an-in-list/

Index on t_py

```
mysql> ALTER TABLE title ADD INDEX t_py
(title(50), production_year);
Query OK, 1543719 rows affected (1 min 52.63 sec)
Records: 1543719  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
AND production_year BETWEEN 2006 and 2009\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: py,t,py_t,t_py
          key: t_py
      key_len: 157
          ref: NULL
         rows: 82
        Extra: Using where
```

0.00s

Recommendations

- Index over multiple columns if it can improve filtering. i.e.
  - GOOD: Only some pilots made between 2006-2009.
  - BAD: All pilots made between 2006-2009.

Recommendations (cont.)

- Don't know what order to specify the columns?
  - RULE: Think how to filter the fastest. Use that order left to right.
  - EXCEPTION: If there's a range (><, `BETWEEN`, `%`). Those always go to the RIGHT.
    - After a a column is used for rangescan, groupby, you cannot use the next column in the composite index

http://www.mysqlperformanceblog.com/2010/01/09/getting-around-optimizer-limitations-with-an-in-list/

A quick sidetrack...

- So far indexes have only been used for filtering.
  - This is the most typical case - **don't forget it**.
- There are also some other ways that MySQL can use an index
  - To avoid having to sort.
  - To prevent temporary tables.
  - To avoid reading rows.
  - ..

The first example again

```
1.  mysql> EXPLAIN SELECT id,title,production_year FROM title
2.  WHERE title  = 'Bambi' ORDER BY production_year\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: ALL
8.  possible_keys: NULL
9.            key: NULL
10.       key_len: NULL
11.           ref: NULL
12.          rows: 1900087
13.         Extra: Using where; Using filesort;
14.  1 row in set (0.00 sec)
```

3.13s

Index prevents sort

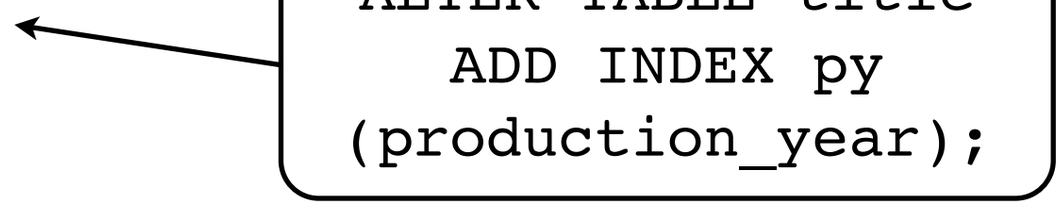
```
mysql> ALTER TABLE title ADD INDEX t_py
  (title(50), production_year);

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: t_py
          key: t_py
      key_len: 152
          ref: const
         rows: 4
        Extra: Using where
1 row in set (0.00 sec)
```

Temporary Table

```
1.  mysql> EXPLAIN select count(*) as c, production_year FROM title
2.   GROUP BY production_year\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: title
7.           type: ALL
8.  possible_keys: NULL
9.            key: NULL
10.        key_len: NULL
11.            ref: NULL
12.           rows: 1524577
13.          Extra: Using temporary; Using filesort
14.  1 row in set (0.00 sec)
```

“Loose index scan”

```
mysql> EXPLAIN select count(*) as c, production_year FROM title
 GROUP BY production_year\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: index
possible_keys: NULL
          key: py
      key_len: 5
          ref: NULL
         rows: 1524577
        Extra: Using index
1 row in set (0.00 sec)
```



Retrieving only limited columns

- Query:

```
SELECT person_id FROM cast_info WHERE
person_role_id = 35721;
```

Retrieving only limited columns:

```
1.  mysql> EXPLAIN SELECT person_id FROM cast_info
2.     WHERE person_role_id = 35721\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: cast_info
7.           type: ref
8.  possible_keys: person_role_id
9.            key: person_role_id
10.       key_len: 5
11.           ref: const
12.          rows: 146
13.         Extra: Using where
14.  1 row in set (0.01 sec)
```

```
ALTER TABLE
cast_info ADD INDEX
(person_role_id);
```

Covering Index Optimization:

```
1.  mysql> EXPLAIN SELECT person_id FROM cast_info
2.  WHERE person_role_id = 35721\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: cast_info
7.           type: ref
8.  possible_keys: person_role_id,person_role_id_person_id
9.            key: person_role_id_person_id
10.        key_len: 5
11.            ref: const
12.           rows: 146
13.          Extra: Using where; Using index
14. 1 row in set (0.00 sec)
```

```
ALTER TABLE
cast_info ADD INDEX
person_role_id_person_
id(person_role_id,
```

Key_len in EXPLAIN

```
key_len: 152
```

- How much bytes of the index is being used for the query?
- In composite indexes, can be used to determine how many columns will be used to _filter_ on.

Key Length Example

```
1.  mysql> EXPLAIN SELECT person_id FROM cast_info
2.  WHERE person_role_id = 35721\G
3.  *************************** 1. row ***************************
4.             id: 1
5.    select_type: SIMPLE
6.          table: cast_info
7.           type: ref
8.  possible_keys: person_role_id,person_role_id_person_id
9.            key: person_role_id_person_id
10.       key_len: 5
11.           ref: const
12.          rows: 146
13.         Extra: Using where; Using index
14. 1 row in set (0.00 sec)
```

`key_len` is only for `WHERE` clause, not the covering part

MySQL Query Optimization

- Basic Query Tuning
- **Beyond EXPLAIN**
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries

The limitations of EXPLAIN

- `EXPLAIN` shows MySQL's intentions - there is no post execution analysis.
  - How many rows actually had to be sorted?
  - Was that temporary table created on disk?
  - Did the `LIMIT 10` result in a quick match, resulting in fewer rows scanned?
  - .. we don't know.

More Advanced

- Combine EXPLAIN with other MySQL diagnostics:
  - `SHOW SESSION STATUS`
    - Recommended to run before and after the query.
    - Available in MySQL 5.0+
  - `SHOW PROFILES`
    - Available in 5.0 (limited), 5.1.
    - Breaks down the time taken on various steps of query execution.
    - Huge amount of skew in any numbers it reports under Linux.
- Slow Query Log Extended Statistics (Percona Server)
  - Will let you know examined rows, temp table on disk, sort on disk, how many IOPS in InnoDB etc.

```
mysql-5141> EXPLAIN select STRAIGHT_JOIN count(*) as c, person_id
FROM cast_info FORCE INDEX(person_id) INNER JOIN title ON
(cast_info.movie_id=title.id) WHERE title.kind_id = 1
GROUP BY cast_info.person_id ORDER by c DESC LIMIT 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: index
possible_keys: NULL
          key: person_id
      key_len: 8
          ref: NULL
         rows: 8
        Extra: Using index; Using temporary; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: eq_ref
possible_keys: PRIMARY,title_kind_id_exists
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.movie_id
         rows: 1
        Extra: Using where
2 rows in set (0.00 sec)
```

Find the actor that stared in the most movies.

MySQL says that only 8 rows were examined in 5.1.41

```
mysql-5089> EXPLAIN select STRAIGHT_JOIN count(*) as c, person_id
FROM cast_info FORCE INDEX(person_id) INNER JOIN title ON
(cast_info.movie_id=title.id) WHERE title.kind_id = 1
GROUP BY cast_info.person_id ORDER by c DESC LIMIT 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: index
possible_keys: NULL
          key: person_id
      key_len: 8
          ref: NULL
         rows: 22187768
        Extra: Using index; Using temporary; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: eq_ref
possible_keys: PRIMARY,title_kind_id_exists
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.movie_id
         rows: 1
        Extra: Using where
2 rows in set (0.00 sec)
```

This is the output from 5.0.89

Double Checking

```
mysql> show status like 'ha%';
+----------------------------+----------+
| Variable_name              | Value    |
+----------------------------+----------+
| Handler_commit             | 0        |
| Handler_delete             | 0        |
| Handler_discover           | 0        |
| Handler_prepare            | 0        |
| Handler_read_first         | 1        |
| Handler_read_key           | 13890229 |
| Handler_read_next          | 14286456 |
| Handler_read_prev          | 0        |
| Handler_read_rnd           | 0        |
| Handler_read_rnd_next      | 2407004  |
| Handler_rollback           | 0        |
| Handler_savepoint          | 0        |
| Handler_savepoint_rollback | 0        |
| Handler_update             | 0        |
| Handler_write              | 2407001  |
+----------------------------+----------+
15 rows in set (0.00 sec)
```

"The number of times the first entry in an index was read"

"The number of requests to read a row based on a key."

"The number of requests to read the next row in key order."

"The number of requests to read the next row in the data file."

"The number of requests to insert a row in a table."

http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html

SHOW PROFILES

- SET profiling = 1;
- *.. run query ..*
- SHOW PROFILES;

```
| Query_ID | Duration     | Query
|        1 | 211.21064300 | select STRAIGHT_JOIN count(*)
as c, person_id  FROM cast_info FORCE INDEX(person_id)
INNER JOIN title ON  (cast_info.movie_id=title.id) WHERE
title.kind_id = 1  GROUP BY cast_info.person_id ORDER by
c DESC LIMIT 1 |
```

- show profile for query 1;

SHOW PROFILES (cont.)

```
mysql> show profile for query 1;
+------------------------------+------------+
| Status                       | Duration   |
+------------------------------+------------+
| starting                     |   0.002133 |
| checking permissions         |   0.000009 |
| checking permissions         |   0.000009 |
| Opening tables               |   0.000035 |
| System lock                  |   0.000022 |
| init                         |   0.000033 |
| optimizing                   |   0.000020 |
| statistics                   |   0.000032 |
| preparing                    |   0.000031 |
| Creating tmp table           |   0.000032 |
| Sorting for group            |   0.000021 |
| executing                    |   0.000005 |
                    ..
                                      ..
| Copying to tmp table         | 113.862209 |
| converting HEAP to MyISAM    |   0.200272 |
| Copying to tmp table on disk |  96.506704 |
| Sorting result               |   0.634087 |
| Sending data                 |   0.000047 |
| end                          |   0.000006 |
| removing tmp table           |   0.004839 |
| end                          |   0.000016 |
| query end                    |   0.000004 |
| freeing items                |   0.000064 |
| logging slow query           |   0.000004 |
| logging slow query           |   0.000003 |
| cleaning up                  |   0.000006 |
+------------------------------+------------+
25 rows in set (0.00 sec)
```

Slow Log Statistics

- SET GLOBAL long_query_time = 0;
- SET GLOBAL log_slow_verbosity = 'full';

This was executed on a machine with entirely cold caches.

```
# Time: 100924 13:58:47
# User@Host: root[root] @ localhost []
# Thread_id: 10  Schema: imdb  Last_errno: 0  Killed: 0
# Query_time: 399.563977  Lock_time: 0.000110  Rows_sent: 1  Rows_examined: 46313608
Rows_affected: 0  Rows_read: 1
# Bytes_sent: 131  Tmp_tables: 1  Tmp_disk_tables: 1  Tmp_table_sizes: 25194923
# InnoDB_trx_id: 1403
# QC_Hit: No  Full_scan: Yes  Full_join: No  Tmp_table: Yes  Tmp_table_on_disk: Yes
# Filesort: Yes  Filesort_on_disk: Yes  Merge_passes: 5
#   InnoDB_IO_r_ops: 1064749  InnoDB_IO_r_bytes: 17444847616  InnoDB_IO_r_wait: 26.935662
#   InnoDB_rec_lock_wait: 0.000000  InnoDB_queue_wait: 0.000000
#   InnoDB_pages_distinct: 65329
SET timestamp=1285336727;
select STRAIGHT_JOIN count(*) as c, person_id  FROM cast_info FORCE INDEX(person_id)
INNER JOIN title ON  (cast_info.movie_id=title.id) WHERE title.kind_id = 1  GROUP BY
cast_info.person_id ORDER by c DESC LIMIT 1;
```

MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- **Optimizing JOINs**
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries

Join Analysis

```
mysql> EXPLAIN SELECT person_info.* FROM name INNER JOIN person_info
ON (name.id=person_info.person_id) AND name.name='Bana, Eric'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: person_info
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1885354
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.person_info.person_id
         rows: 1
        Extra: Using where
```

Filter out as much as possible first, you can only do this by looking at WHERE clause

Join Analysis

```
mysql> EXPLAIN SELECT person_info.* FROM name INNER JOIN person_info
ON (name.id=person_info.person_id) AND name.name='Bana, Eric'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: ref
possible_keys: PRIMARY,name
          key: name
      key_len: 152
          ref: const
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: person_info
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1885354
        Extra: Using where; Using join buffer
2 rows in set (0.00 sec)
```

```
ALTER TABLE name ADD
  INDEX (name(50));
```

Join Analysis

```
mysql> EXPLAIN SELECT person_info.* FROM name INNER JOIN person_info
ON (name.id=person_info.person_id) AND name.name='Bana, Eric'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: ref
possible_keys: PRIMARY,name
          key: name
      key_len: 152
          ref: const
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: person_info
         type: ref
possible_keys: person_id
          key: person_id
      key_len: 4
          ref: imdb.name.id
         rows: 2
        Extra:
2 rows in set (0.01 sec)
```

```
ALTER TABLE
person_info ADD INDEX
(person_id);
```

```
mysql> EXPLAIN SELECT name.* FROM name INNER JOIN cast_info
ON name.id=cast_info.person_id INNER JOIN char_name
ON cast_info.person_role_id=char_name.id WHERE char_name.name = 'James Bond'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 22743540
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_id
         rows: 1
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_role_id
         rows: 1
        Extra: Using where
3 rows in set (0.07 sec)
```

The order you see these tables mentioned is the order MySQL has decided to join on.

```
mysql> EXPLAIN SELECT name.* FROM name INNER JOIN cast_info
ON name.id=cast_info.person_id INNER JOIN char_name
ON cast_info.person_role_id=char_name.id WHERE char_name.name = 'James Bond'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 22743540
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_id
         rows: 1
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_role_id
         rows: 1
        Extra: Using where
3 rows in set (0.07 sec)
```

Filter out as much as possible first, you can only do this by looking at WHERE clause

First Index:

```
mysql> ALTER TABLE char_name ADD index name_idx (name(50));
```

```
mysql> EXPLAIN SELECT name.* FROM name INNER JOIN cast_info
ON name.id=cast_info.person_id INNER JOIN char_name
ON cast_info.person_role_id=char_name.id WHERE char_name.name = 'James Bond'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: ref
possible_keys: PRIMARY,name_idx
          key: name_idx
      key_len: 152
          ref: const
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 22743540
        Extra: Using where; Using join buffer
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_id
         rows: 1
        Extra:
3 rows in set (0.24 sec)
```

Filter out as much as possible first, you can only do this by looking at WHERE clause

The order changed. cast_info was previously first!

Second Index:

```
mysql> ALTER TABLE cast_info ADD INDEX
person_role_id_person_id(person_role_id, person_id);
```

```
mysql> EXPLAIN SELECT name.* FROM name INNER JOIN cast_info
ON name.id=cast_info.person_id INNER JOIN char_name
ON cast_info.person_role_id=char_name.id WHERE char_name.name = 'James Bond'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: ref
possible_keys: PRIMARY,name_idx
          key: name_idx
      key_len: 152
          ref: const
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref
possible_keys: person_role_id_person_id
          key: person_role_id_person_id
      key_len: 5
          ref: imdb.char_name.id
         rows: 4
        Extra: Using where; Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_id
         rows: 1
        Extra:
3 rows in set (0.17 sec)
```

TIP: Using a *covering index* means that we retrieve all data directly from the index.

0.00s

Join Methods

- You need to filter as fast as possible. Here's why:
  - MySQL only uses one join method - a nested loop join.

Sample Query

- Find all actors that were active between 1960 and 1970:

Actors:

id	first_name	last_name
1	Sean	Connery
2	George	Lazenby
3	Roger	Moore
4	Timothy	Dalton
5	Pierce	Brosnan
6	Daniel	Craig

Movies:

id	name	year
1	Dr. No	1962
2	From Russia with Love	1963
3	Goldfinger	1964
3	You only live twice	1967
5	On Her Majesty's Secret Service	1969
..	..	..

Sample Query

- Find all actors that were active between 1960 and 1970:

Actors:

id	first_name	last_name
1	Sean	Connery
2	George	Lazenby
3	Roger	Moore
4	Timothy	Dalton
5	Pierce	Brosnan
6	Daniel	Craig

Movies:

id	name	year
1	Dr. No	1962
2	From Russia with Love	1963
3	Goldfinger	1964
3	You only live twice	1967
5	On Her Majesty's Secret Service	1969
..	..	..

Sample Query

- Find all actors that were active between 1960 and 1970:

Actors:

id	first_name	last_name
1	Sean	Connery
2	George	Lazenby
3	Roger	Moore
4	Timothy	Dalton
5	Pierce	Brosnan
6	Daniel	Craig

Movies:

id	name	year
1	Dr. No	1962
2	From Russia with Love	1963
3	Goldfinger	1964
3	You only live twice	1967
5	On Her Majesty's Secret Service	1969
..	..	..

Sample Query

- Find all actors that were active between 1960 and 1970:

Actors:

id	first_name	last_name
1	Sean	Connery
2	George	Lazenby
3	Roger	Moore
4	Timothy	Dalton
5	Pierce	Brosnan
6	Daniel	Craig

Movies:

id	name	year
1	Dr. No	1962
2	From Russia with Love	1963
3	Goldfinger	1964
3	You only live twice	1967
5	On Her Majesty's Secret Service	1969
..	..	..

Sample Query

- Find all actors that were active between 1960 and 1970:

Actors:

id	first_name	last_name
1	Sean	Connery
2	George	Lazenby
3	Roger	Moore
4	Timothy	Dalton
5	Pierce	Brosnan
6	Daniel	Craig

Movies:

id	name	year
1	Dr. No	1962
2	From Russia with Love	1963
3	Goldfinger	1964
3	You only live twice	1967
5	On Her Majesty's Secret Service	1969
..	..	..

If that query is common

- When you can't filter enough on one table, bring some of the other filters from the other tables to the first one:

Actors:

id	first_name	last_name	start_date	finish_date
1	Sean	Connery	1962	1971
2	George	Lazenby	1969	1969
3	Roger	Moore	1973	1985
4	Timothy	Dalton	1987	1989
5	Pierce	Brosnan	1995	2002
6	Daniel	Craig	2006	2011

MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- **Subquery**
- Other Optimizations
- Table Schema
- Identifying Bad Queries

Subquery Analysis

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id IN
(SELECT id FROM kind_type WHERE kind='video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1567676
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: kind_type
         type: const
possible_keys: PRIMARY,kind_id
          key: kind_id
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
2 rows in set (0.00 sec)
```

Will it fix it if we add an index on `title.kind_id`?

4.84s

With index on title.kind_id

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id IN
(SELECT id FROM kind_type WHERE kind='video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1574389
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: kind_type
         type: const
possible_keys: PRIMARY,kind_id
          key: kind_id
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
2 rows in set (0.11 sec)
```

No! It doesn't. Why is this?

4.9s

Scalar Subquery

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id =
(SELECT id FROM kind_type WHERE kind='video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: title
         type: ref
possible_keys: k
          key: k
      key_len: 4
          ref: const
         rows: 8502
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: SUBQUERY
        table: kind_type
         type: const
possible_keys: kind_id
          key: kind_id
      key_len: 47
          ref:
         rows: 1
        Extra: Using index
2 rows in set (0.10 sec)
```

Change to using equality, it works! but only when kind is unique!

0.07s

Solving via Join

```
mysql> EXPLAIN SELECT title.* FROM title INNER JOIN kind_type ON
(title.kind_id = kind_type.id) WHERE kind_type.kind IN ('video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: kind_type
         type: const
possible_keys: PRIMARY,kind_id
          key: kind_id
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: kind_id
          key: kind_id
      key_len: 4
          ref: const
         rows: 8502
        Extra:
2 rows in set (0.00 sec)
```

It's okay to have multiple kind's specified using this syntax.

```
ALTER TABLE
title ADD KEY
(kind_id);
```

Should We Completely Avoid Them?

- No, Benchmark!
- "Delayed Join"
  http://www.mysqlperformanceblog.com/2007/04/06/using-delayed-join-to-optimize-count-and-limit-queries/

MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- **Other Optimizations**
- Table Schema
- Identifying Bad Queries

Next Problem (cont.)

- The problem with this schema, is there's just a couple of outliers with really long names:

```
mysql> SELECT max(length(title)) from title;
+--------------------+
| max(length(title)) |
+--------------------+
|                334 |
+--------------------+
1 row in set (6.64 sec)
mysql> SELECT max(length(name)) from char_name;
+--------------------+
| max(length(name)) |
+--------------------+
|                478 |
+--------------------+
1 row in set (7.80 sec)
```

Two ways to solve this:

- 1. Pick a good length to get a lot of uniqueness:

```
1.  mysql> SELECT count(distinct(title)) as n_unique,
2.  count(distinct(LEFT(title, 100))) as n100, count(distinct(LEFT(title, 75))) as n75,
3.  count(distinct(LEFT(title, 50))) as n50, count(distinct(LEFT(title, 40))) as n40,
4.  count(distinct(LEFT(title, 30))) as n30, count(distinct(LEFT(title, 20))) as n20,
5.  count(distinct(LEFT(title, 10))) as n10  FROM title;
6.
```

Two ways to solve this:

- 1. Pick a good length to get a lot of uniqueness:

```
mysql> SELECT count(distinct(title)) as n_unique,
count(distinct(LEFT(title, 100))) as n100, count(distinct(LEFT(title, 75))) as n75,
count(distinct(LEFT(title, 50))) as n50, count(distinct(LEFT(title, 40))) as n40,
count(distinct(LEFT(title, 30))) as n30, count(distinct(LEFT(title, 20))) as n20,
count(distinct(LEFT(title, 10))) as n10  FROM title;
+----------+--------+--------+--------+--------+--------+--------+--------+
| n_unique | n100   | n75    | n50    | n40    | n30    | n20    | n10    |
+----------+--------+--------+--------+--------+--------+--------+--------+
|   998335 | 998320 | 998291 | 997887 | 996727 | 991532 | 960894 | 624949 |
+----------+--------+--------+--------+--------+--------+--------+--------+
```

Two ways to solve this:

- 1. Pick a good length to get a lot of uniqueness:

```
mysql> SELECT count(distinct(title)) as n_unique,
count(distinct(LEFT(title, 100))) as n100, count(distinct(LEFT(title, 75))) as n75,
count(distinct(LEFT(title, 50))) as n50, count(distinct(LEFT(title, 40))) as n40,
count(distinct(LEFT(title, 30))) as n30, count(distinct(LEFT(title, 20))) as n20,
count(distinct(LEFT(title, 10))) as n10  FROM title;
+----------+--------+--------+--------+--------+--------+--------+--------+
| n_unique | n100   | n75    | n50    | n40    | n30    | n20    | n10    |
+----------+--------+--------+--------+--------+--------+--------+--------+
|   998335 | 998320 | 998291 | 997887 | 996727 | 991532 | 960894 | 624949 |
+----------+--------+--------+--------+--------+--------+--------+--------+
```

96% uniqueness, but only 20 chars instead of 300+
Looks pretty good to me:

```
ALTER TABLE title ADD index (name(20))
```

Option 2: Emulate a Hash Index

```
1.  mysql> ALTER TABLE title ADD title_crc32 INT UNSIGNED, ADD INDEX (title_crc32);
2.  mysql> UPDATE title SET title_crc32 = crc32(title);
3.  mysql> SELECT count(distinct(BINARY title)),
4.  count(distinct(title_crc32)) from title;
5.
```

- Is possible only with MEMORY engine:

```
ALTER TABLE table ADD INDEX USING HASH (title);
```

Option 2: Emulate a Hash Index

```
mysql> ALTER TABLE title ADD title_crc32 INT UNSIGNED, ADD INDEX (title_crc32);
mysql> UPDATE title SET title_crc32 = crc32(title);
mysql> SELECT count(distinct(BINARY title)),
count(distinct(title_crc32)) from title;
+----------------------------------+----------------------------------+
| count(distinct(BINARY title)) | count(distinct(title_crc32)) |
+----------------------------------+----------------------------------+
|                          1001509 |                          1001393 |
+----------------------------------+----------------------------------+
1 row in set (44.81 sec)
```

A good hashing algorithm has good distribution. How good is this?

Option 2: Hash Index (cont.)

- Query needs to be transformed slightly to:
  - `SELECT * FROM title WHERE title_crc32=crc32('my_title') AND title='my_title';`
- All updates/inserts need to also update the value of title_crc32:
  - Can be easily done via the application, or a trigger if write load is very low.

Pros/Cons

Prefix Index:

- **Pro:**
  Built in to MySQL/no magic required.

- **Cons:**
  Not very effective when the start of the string is not very unique.

Hash Index:

- **Pro:**
  Very Good when there is not much uniqueness until very far into the string.

- **Cons:**
  Equality searches only.
  Requires ugly magic to work with collations/ case sensitivity.

Things are looking good!?

- Please ***don't*** take away that adding indexes == only secret to performance.
  - The story is a lot more complicated.
  - We have to get around optimizer limitations, and a lack of index/join options.

Optimizer Hints

- Optimizer decision making is all about tradeoffs.
  - MySQL wants to pick the best plan, but it can't be exhaustive in deciding if it takes too long.
- If MySQL is off by a lot, you may want to provide a hint:
  - `USE INDEX`
  - `FORCE INDEX`
  - `IGNORE INDEX`
  - `STRAIGHT_JOIN`
- See: http://dev.mysql.com/doc/refman/5.5/en/index-hints.html

Optimizer Hints, Should We?

```
mysql> explain SELECT title.* FROM title force key (kind_id_title)
INNER JOIN kind_type ON  (title.kind_id = kind_type.id)
WHERE kind_type.kind IN ('video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: kind_type
         type: const
possible_keys: PRIMARY,kind
          key: kind
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: kind_id_title
          key: kind_id_title
      key_len: 4
          ref: const
         rows: 14240
        Extra:
```

More features & workarounds

- `EXPLAIN` only works for `SELECT`: convert `UPDATE`/`DELETE` to `SELECT` (feature added in 5.6)
- The IN() list workaround
  - http://www.mysqlperformanceblog.com/2010/01/09/getting-around-optimizer-limitations-with-an-in-list/
- Index Merge
  - http://dev.mysql.com/doc/refman/5.1/en/index-merge-optimization.html

MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- **Table Schema**
- Identifying Bad Queries

CREATE TABLE

```
1.  mysql> SHOW CREATE TABLE title\G
2.  *************************** 1. row ***************************
3.         Table: title
4.  Create Table: CREATE TABLE `title` (
5.    `id` int(11) NOT NULL AUTO_INCREMENT,
6.    `title` text NOT NULL,
7.    `imdb_index` varchar(12) DEFAULT NULL,
8.    `kind_id` int(11) NOT NULL,
9.    `production_year` int(11) DEFAULT NULL,
10.   `imdb_id` int(11) DEFAULT NULL,
11.   `phonetic_code` varchar(5) DEFAULT NULL,
12.   `episode_of_id` int(11) DEFAULT NULL,
13.   `season_nr` int(11) DEFAULT NULL,
14.   `episode_nr` int(11) DEFAULT NULL,
15.   `series_years` varchar(49) DEFAULT NULL,
16.   `title_crc32` int(10) unsigned DEFAULT NULL,
17.   PRIMARY KEY (`id`),
18.   KEY `kind_id` (`kind_id`),
19.   KEY `title_3` (`title`(255)),
20.   KEY `kind_id_title` (`kind_id`,`title`(50))
21. ) ENGINE=InnoDB AUTO_INCREMENT=1543721 DEFAULT CHARSET=utf8
```

SHOW FIELDS

```
mysql> SHOW COLUMNS FROM title;
+-----------------+------------------+------+-----+---------+----------------+
| Field           | Type             | Null | Key | Default | Extra          |
+-----------------+------------------+------+-----+---------+----------------+
| id              | int(11)          | NO   | PRI | NULL    | auto_increment |
| title           | text             | NO   | MUL | NULL    |                |
| imdb_index      | varchar(12)      | YES  |     | NULL    |                |
| kind_id         | int(11)          | NO   | MUL | NULL    |                |
| production_year | int(11)          | YES  |     | NULL    |                |
| imdb_id         | int(11)          | YES  |     | NULL    |                |
| phonetic_code   | varchar(5)       | YES  |     | NULL    |                |
| episode_of_id   | int(11)          | YES  |     | NULL    |                |
| season_nr       | int(11)          | YES  |     | NULL    |                |
| episode_nr      | int(11)          | YES  |     | NULL    |                |
| series_years    | varchar(49)      | YES  |     | NULL    |                |
| title_crc32     | int(10) unsigned | YES  |     | NULL    |                |
+-----------------+------------------+------+-----+---------+----------------+
12 rows in set (0.00 sec)
```

SHOW INDEXES

```
mysql> SHOW INDEXES FROM title;
+-------+------------+----------------+-----+-------------+------+---------+----------+--------+------+------------+
| Table | Non_unique | Key_name       | Seq | Column_name | Coll | Card    | Sub_part | Packed | Null | Index_type |
+-------+------------+----------------+-----+-------------+------+---------+----------+--------+------+------------+
| title |          0 | PRIMARY        |   1 | id          | A    | 1542249 |     NULL | NULL   |      | BTREE      |
| title |          1 | kind_id        |   1 | kind_id     | A    |     204 |     NULL | NULL   |      | BTREE      |
| title |          1 | title_3        |   1 | title       | A    |     204 |      255 | NULL   |      | BTREE      |
| title |          1 | kind_id_title  |   1 | kind_id     | A    |     204 |     NULL | NULL   |      | BTREE      |
| title |          1 | kind_id_title  |   2 | title       | A    |     204 |       50 | NULL   |      | BTREE      |
+-------+------------+----------------+-----+-------------+------+---------+----------+--------+------+------------+
5 rows in set (0.00 sec)

```

Day One Advice (1)

- Keep it simple - store atomic types in each field.
  - This means storing first name and last name as two separate fields.
- Don’t try and get tricky with how you store the data.
  - i.e. this field means it’s a phone number unless this other field is set to something.

Day One Advice (2)

- Use appropriate data types -
  - If you're not sure about the length, varchar(100) is still much better than varchar(255).
  - Use an Integer for a number. Decimal for precision numbers, float for non-precision numbers, etc.
  - If integers don't have to be negative, use unsigned.

Day One Advice (3)

- Plan how you will be accessing the data.
  - If you know that you have to do 4 expensive joins to execute a common query - it might not be the best solution.
  - It's okay to have redundancy in the database from very early on. One example is pre-generating the 'average score' on IMDB titles.

Clustered Index

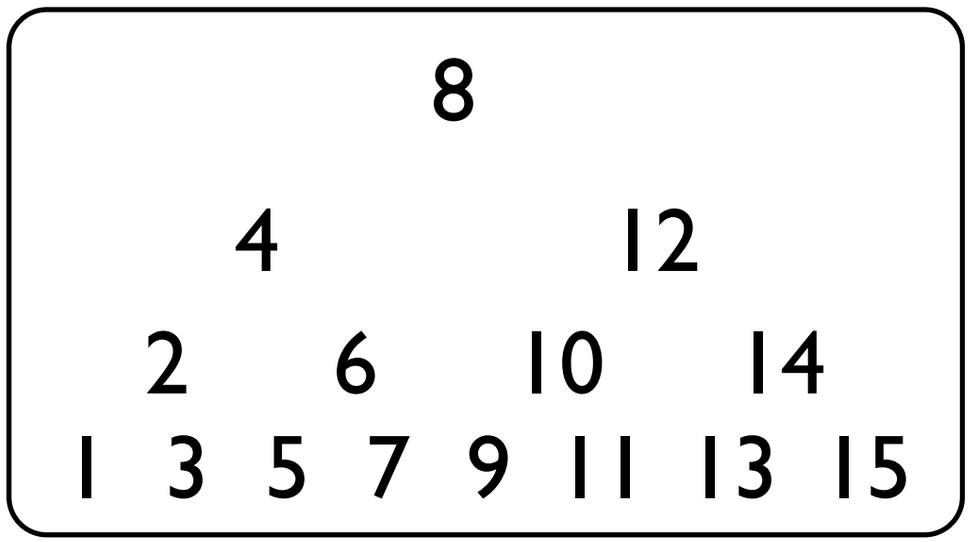
- Everything in InnoDB is an index:
  - Data is stored in a clustered index organized by the primary key. In the absence of a primary key, the first unique not null key is selected*.
  - Other indexes are stored in secondary indexes.

* In the absence of a unique key, a hidden 6 byte key is created.

What is a clustered index?

- First lets look at how MyISAM stores data*:

Staff.MYI



Data is stored “roughly” in insertion order, with no guarantees, i.e.

Deleted rows may be filled with newer records.

Staff.MYD

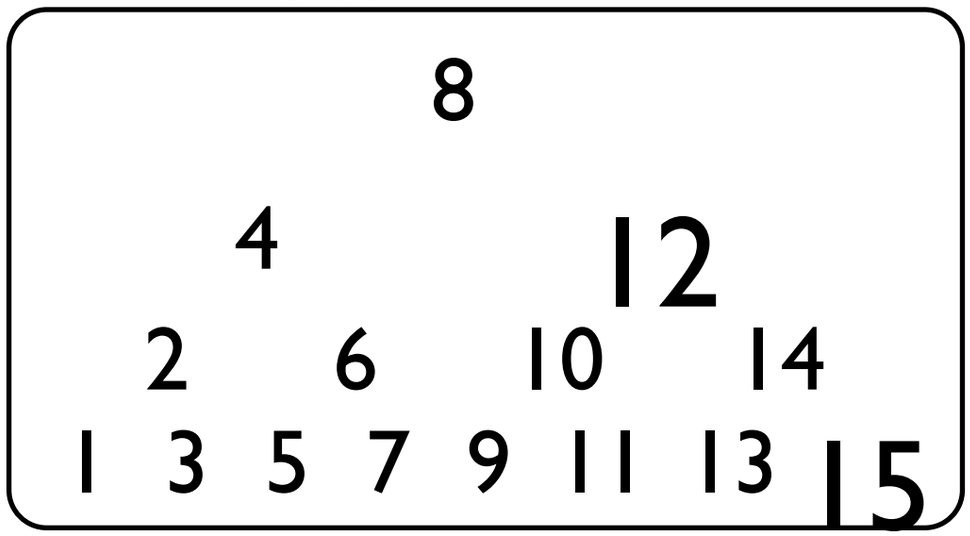
ID	First Name
1	lePeter
2	leVadim
7	leFred
4	leEwen
5	leBaron
..	..

* Illustrating B-Tree as Binary Tree for simplicity

What is a clustered index (cont.)

- A MyISAM primary key lookup looks something like this:

Staff.MYI

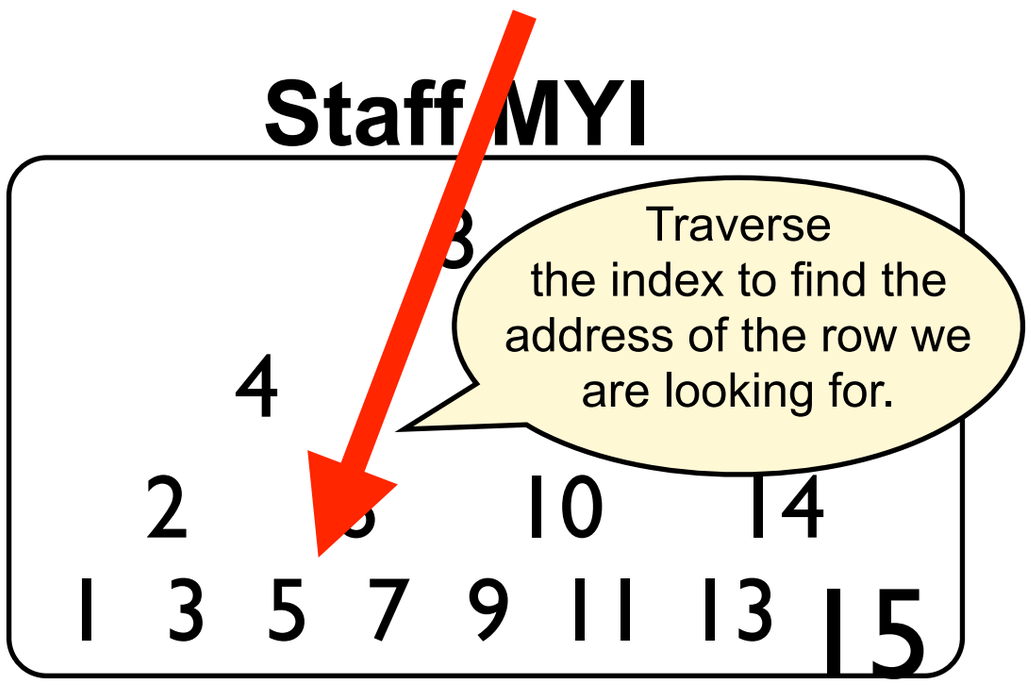


Staff.MYD

ID	First Name
1	lePeter
2	leVadim
7	leFred
4	leEwen
5	leBaron
..	..

What is a clustered index (cont.)

- A MyISAM primary key lookup looks something like this:

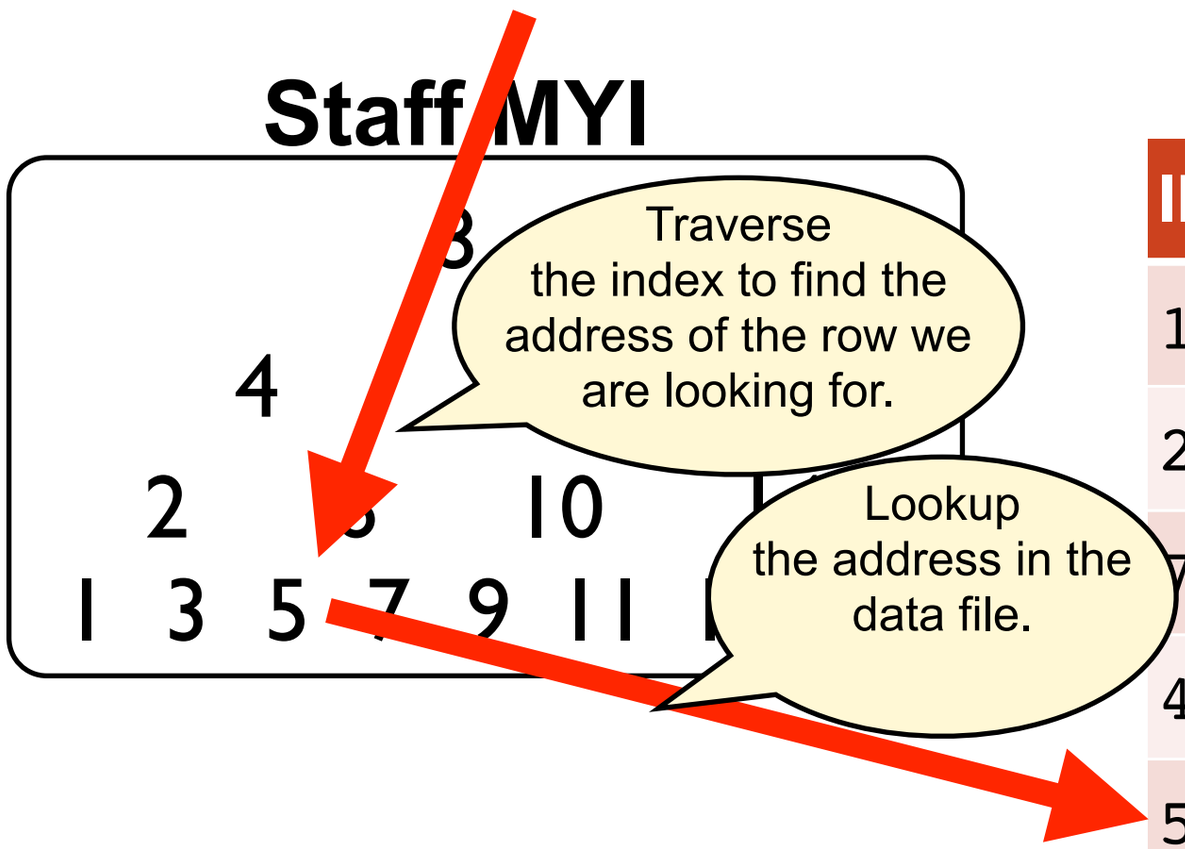


Staff.MYD

ID	First Name
1	lePeter
2	leVadim
7	leFred
4	leEwen
5	leBaron
..	..

What is a clustered index (cont.)

- A MyISAM primary key lookup looks something like this:

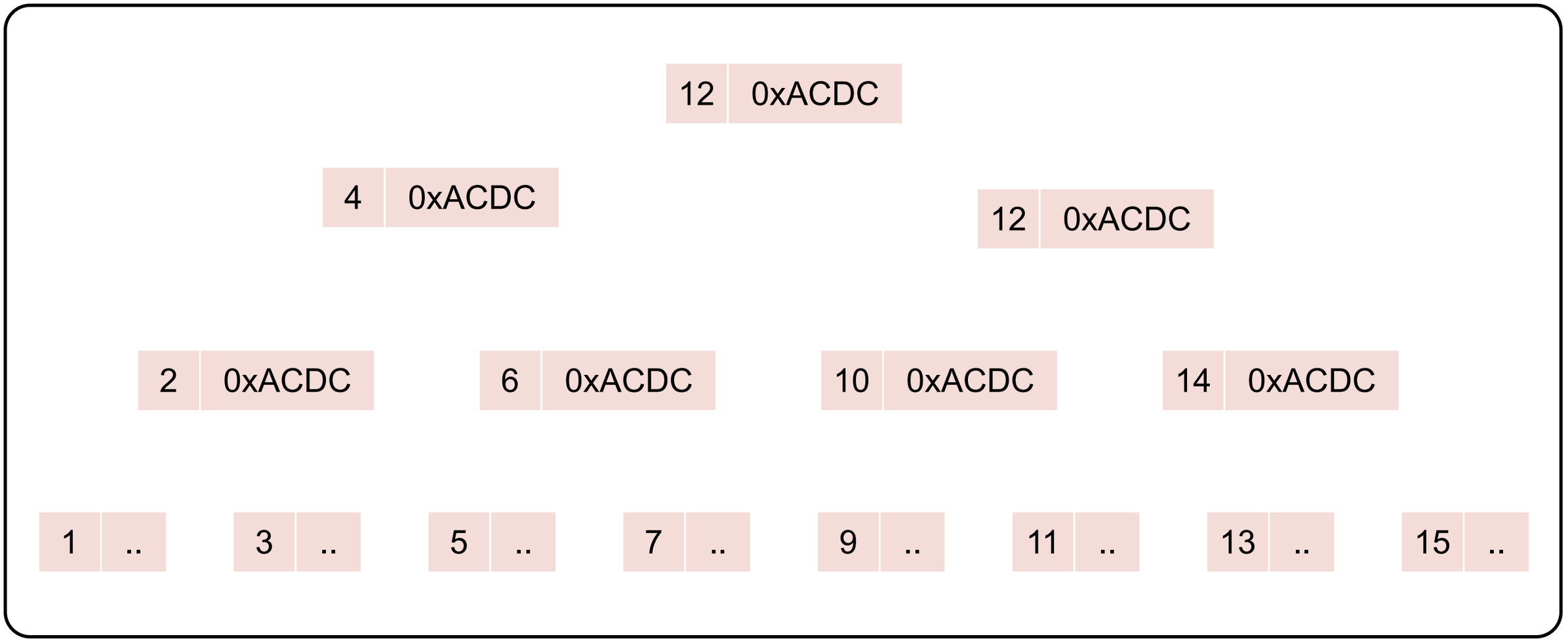


Staff.MYD

ID	First Name
1	lePeter
2	leVadim
7	leFred
4	leEwen
5	leBaron
..	..

What is a clustered index (cont.)

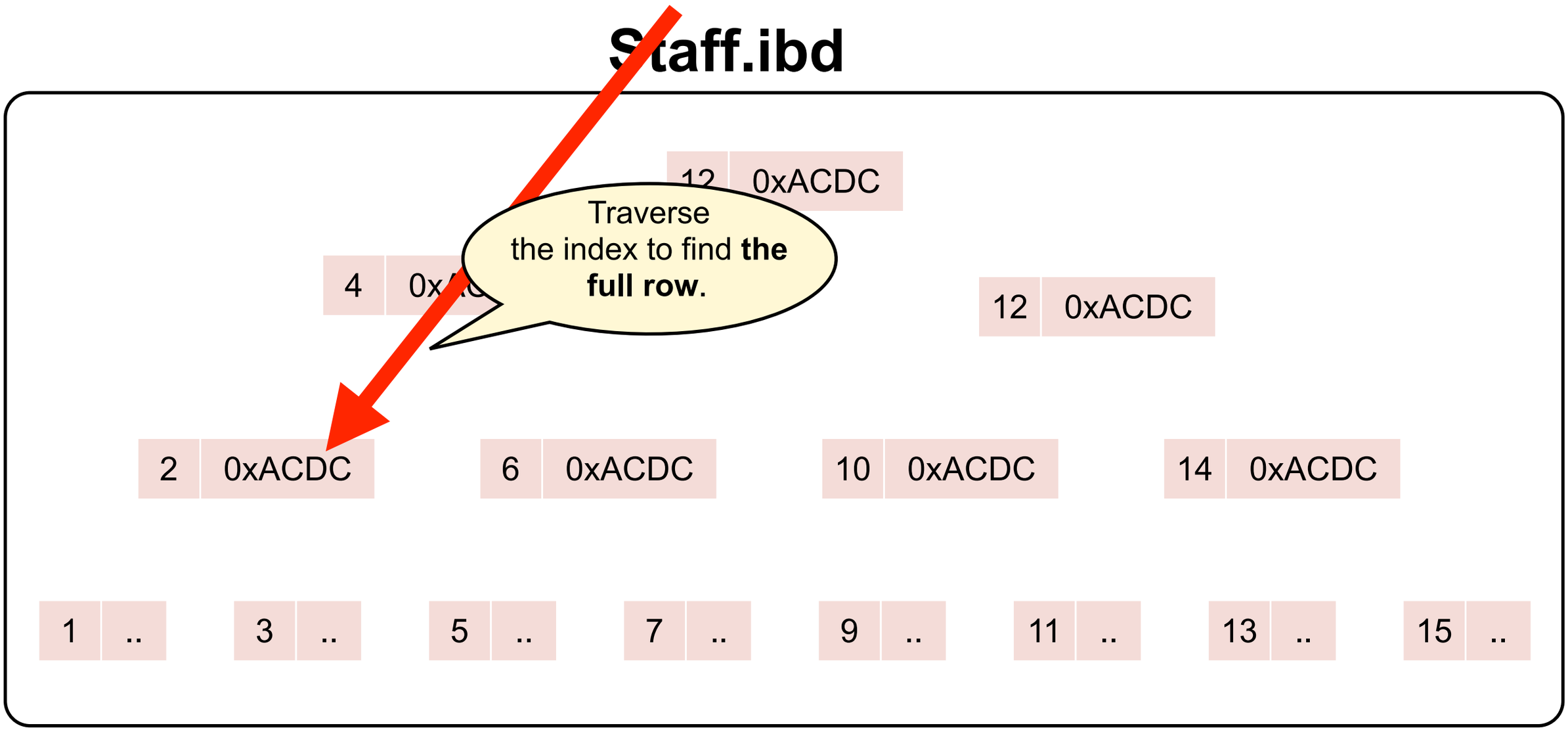
- An InnoDB Primary Key lookup looks like this:



* Illustrating B+Tree as Binary Tree for simplicity.

What is a clustered index (cont.)

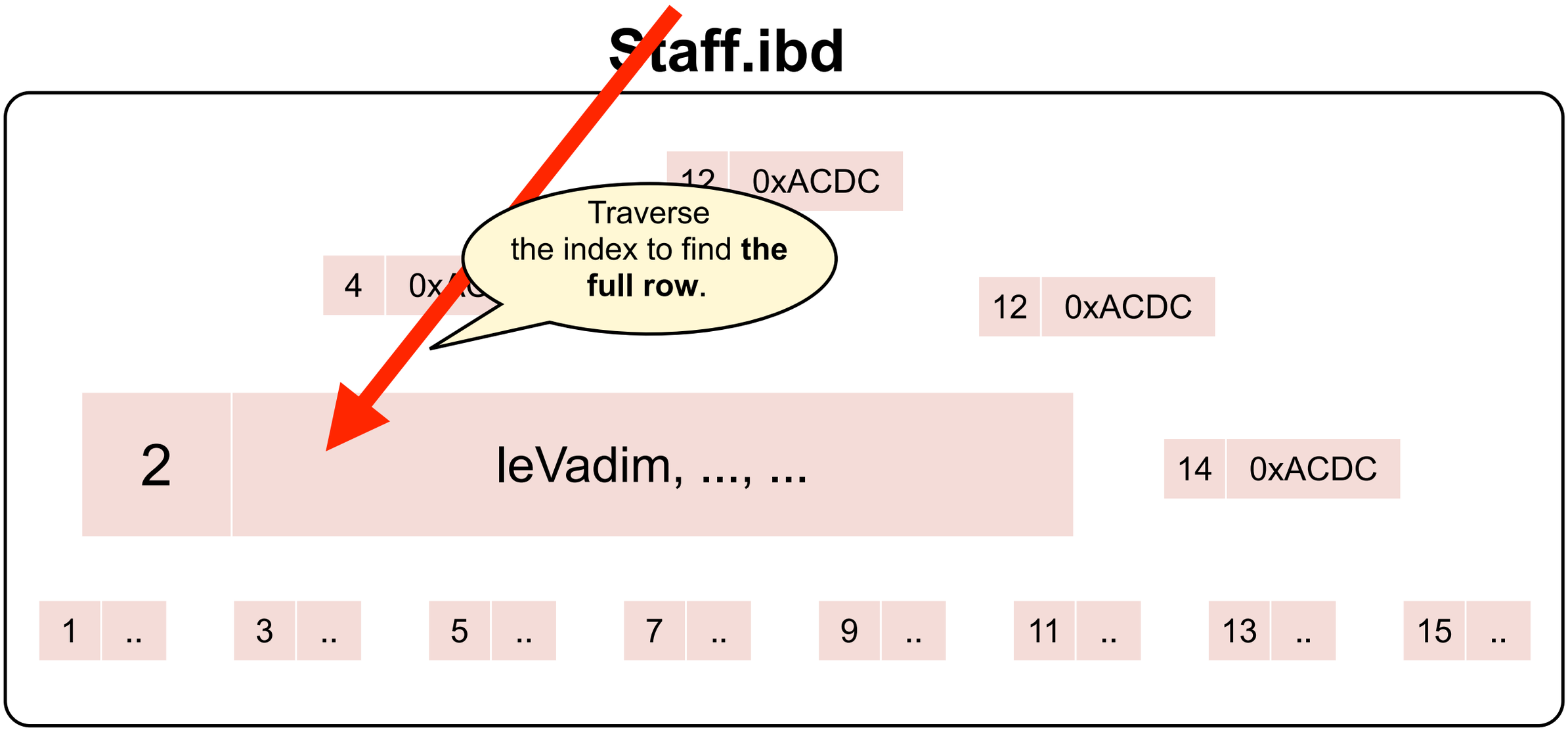
- An InnoDB Primary Key lookup looks like this:



* Illustrating B+Tree as Binary Tree for simplicity.

What is a clustered index (cont.)

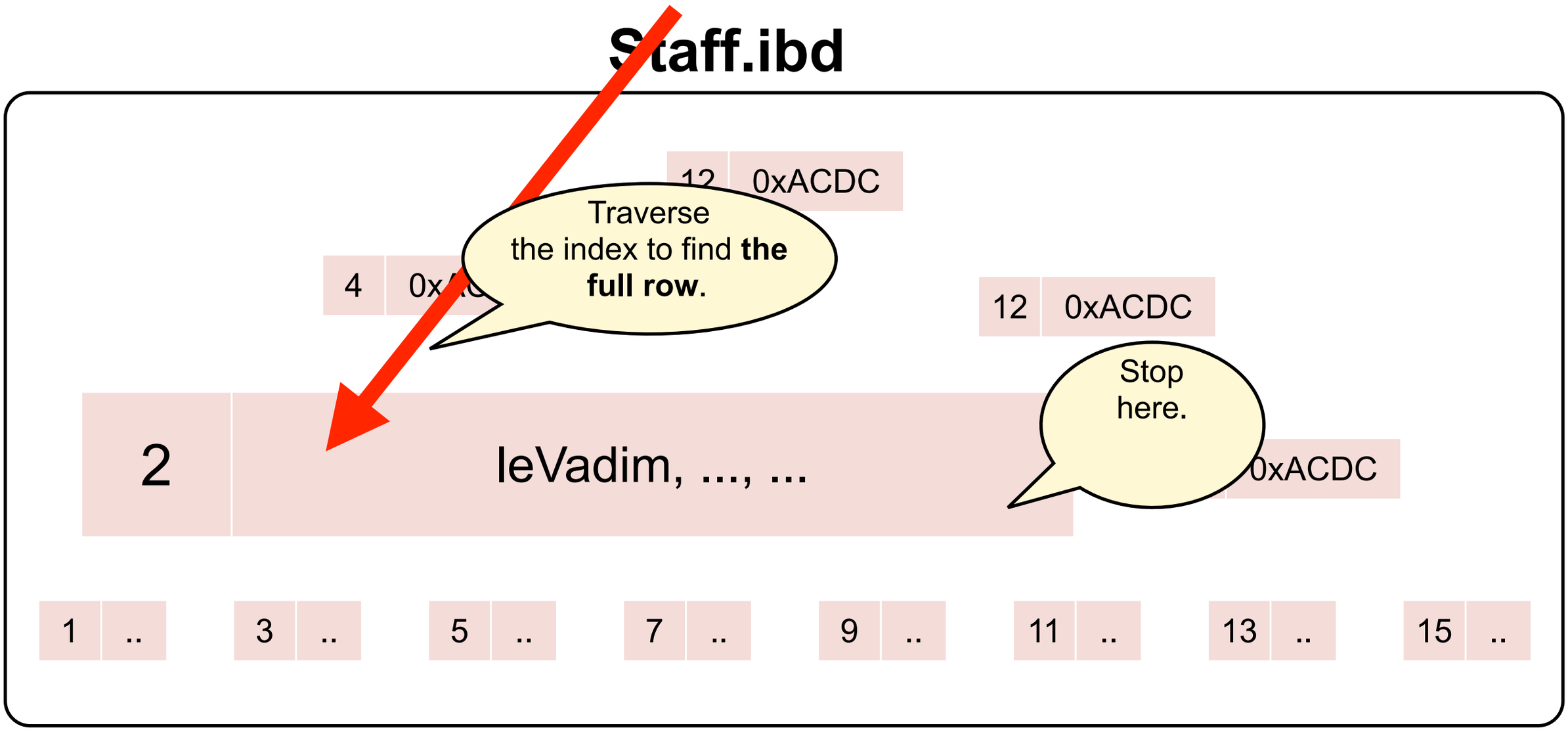
- An InnoDB Primary Key lookup looks like this:



* Illustrating B+Tree as Binary Tree for simplicity.

What is a clustered index (cont.)

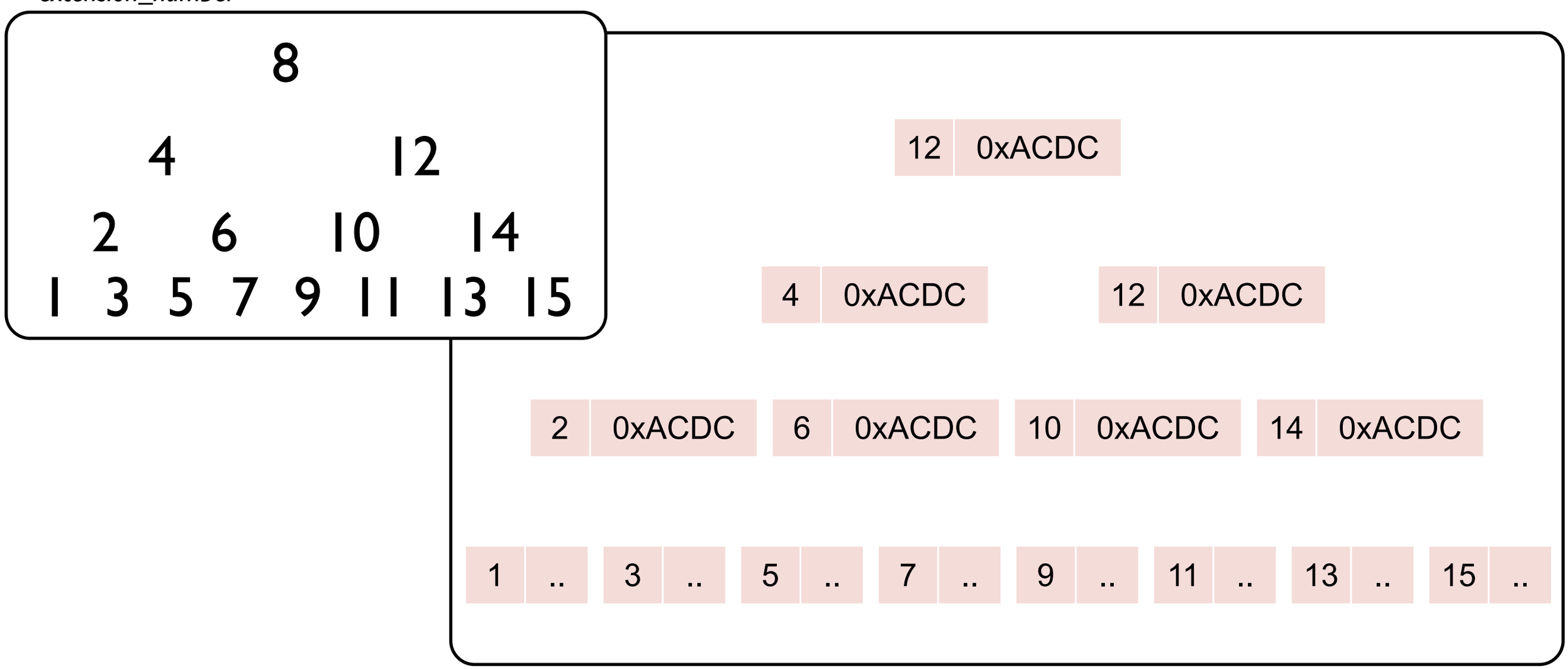
- An InnoDB Primary Key lookup looks like this:



* Illustrating B+Tree as Binary Tree for simplicity.

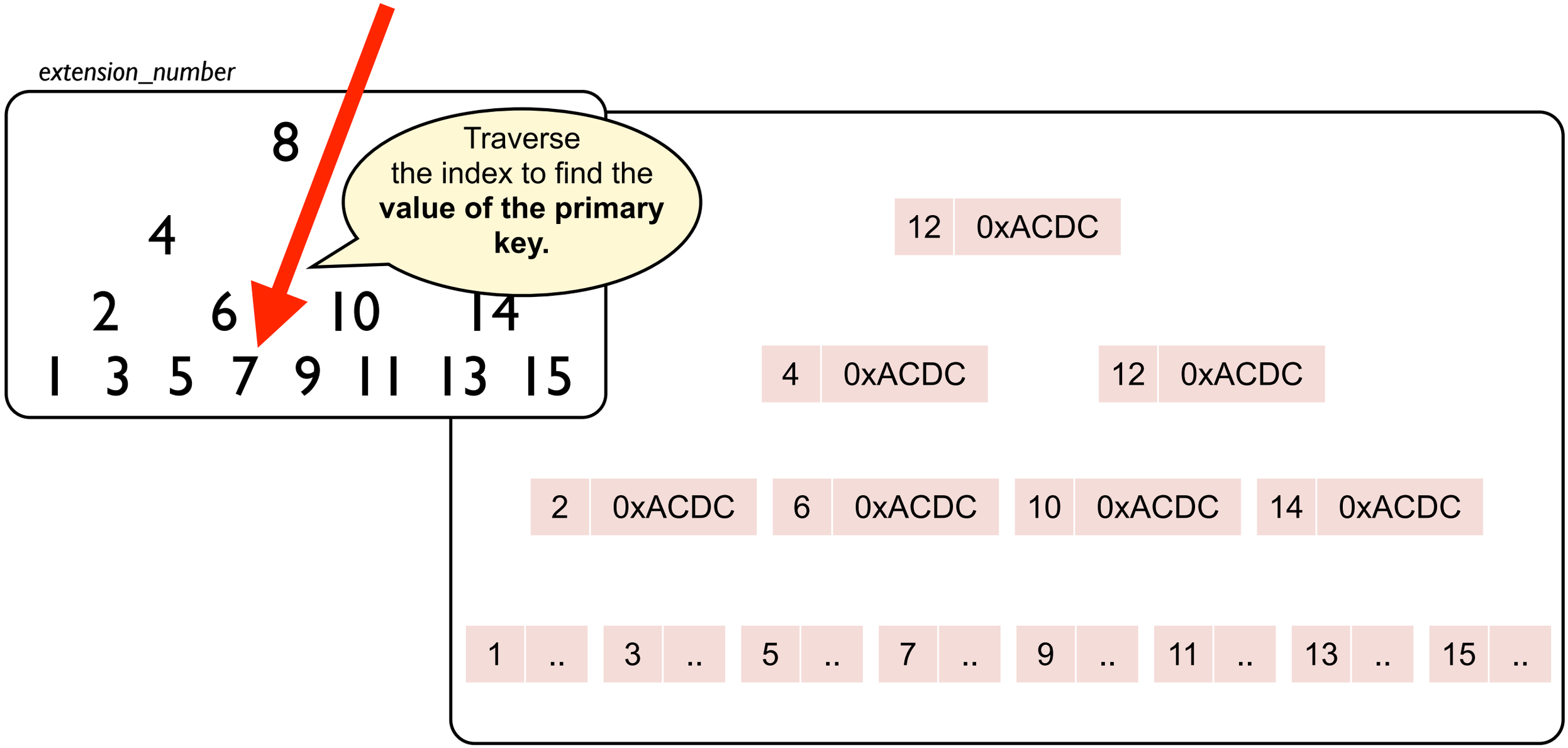
What is a clustered index (cont.)

- A secondary key lookup looks like this:



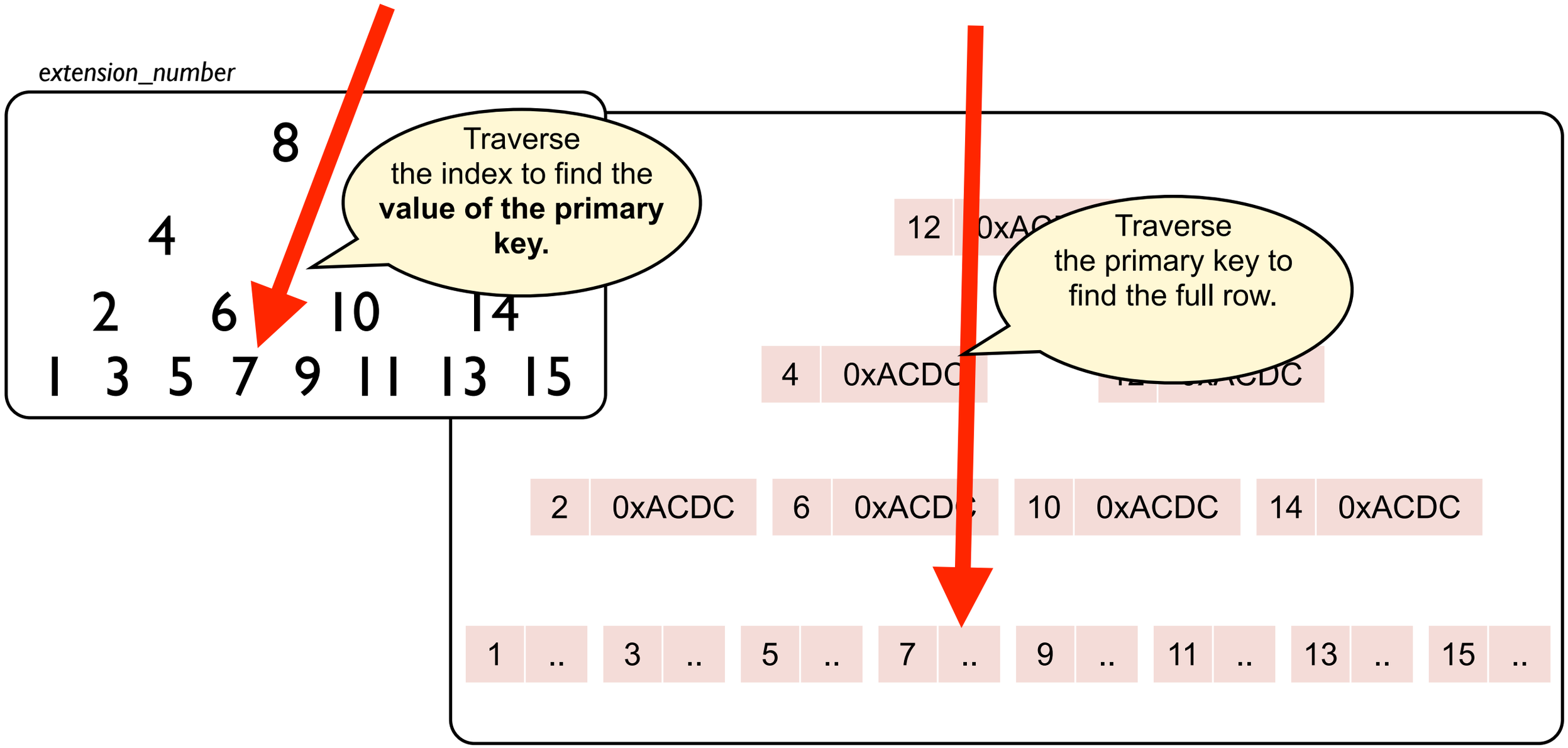
What is a clustered index (cont.)

- A secondary key lookup looks like this:



What is a clustered index (cont.)

- A secondary key lookup looks like this:



Clustered Index (cont.)

- This design has some interesting consequences:
  - Primary key lookups are very fast.
  - Inserting data in order is fast - out of order can be very slow, and cause fragmentation.
  - Secondary indexes can become very large if you have a large primary key.

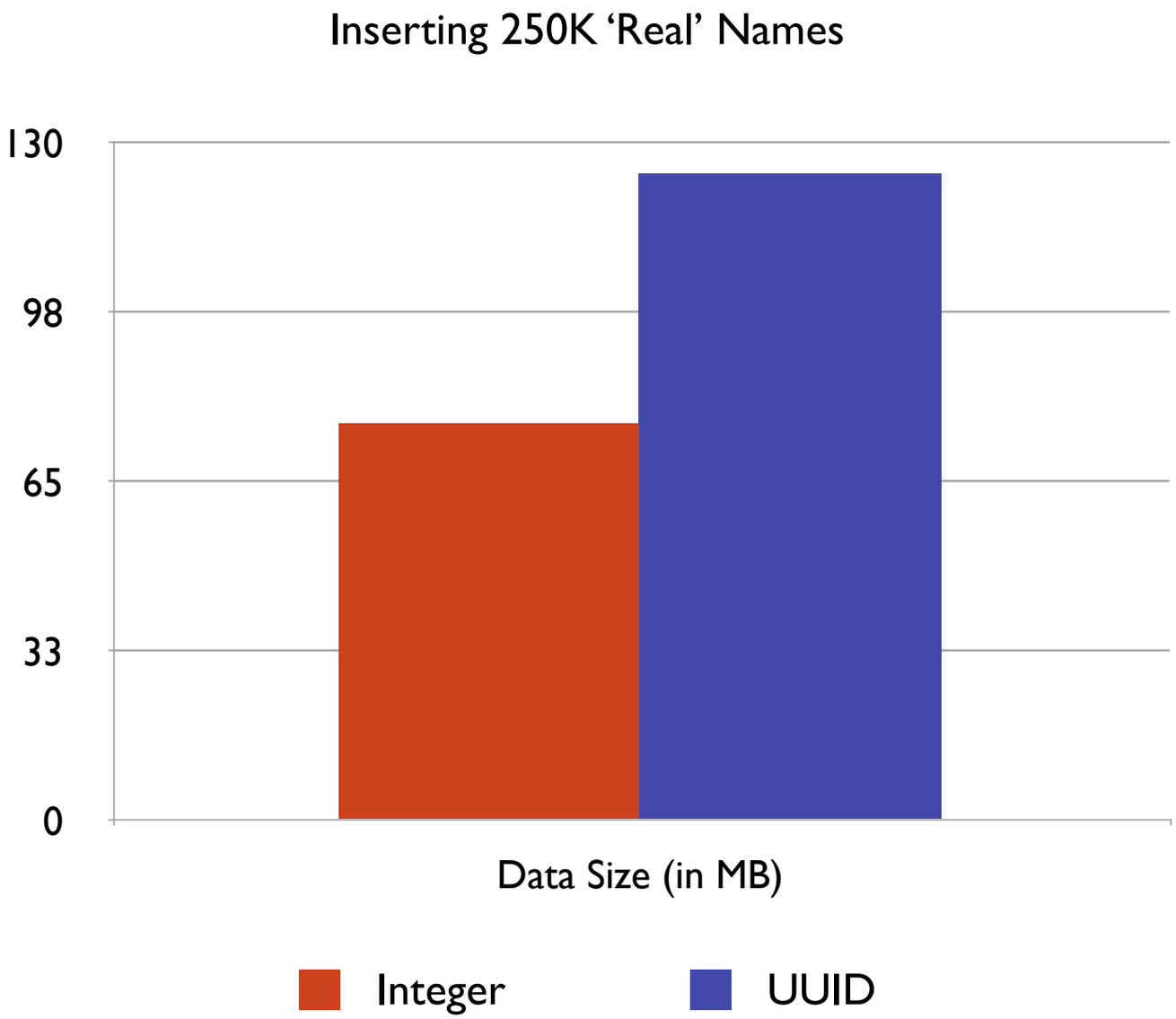
Clustered Index (cont.)

- In practical terms this means:
  - Don’t use GUIDs for InnoDB tables!
  - Never piggy-back the primary key index into the end of a composite index or covering index - it is already included for free.

Some MySQL(InnoDB) Specifics

- **Primary Keys:**
  - Always specify one.
  - Keep it short.
  - Try and make it your primary access method.
  - Keep insertion incremental.
- **Composite Keys:**
  - Don’t ever include the primary key index as part of a *covering* index.

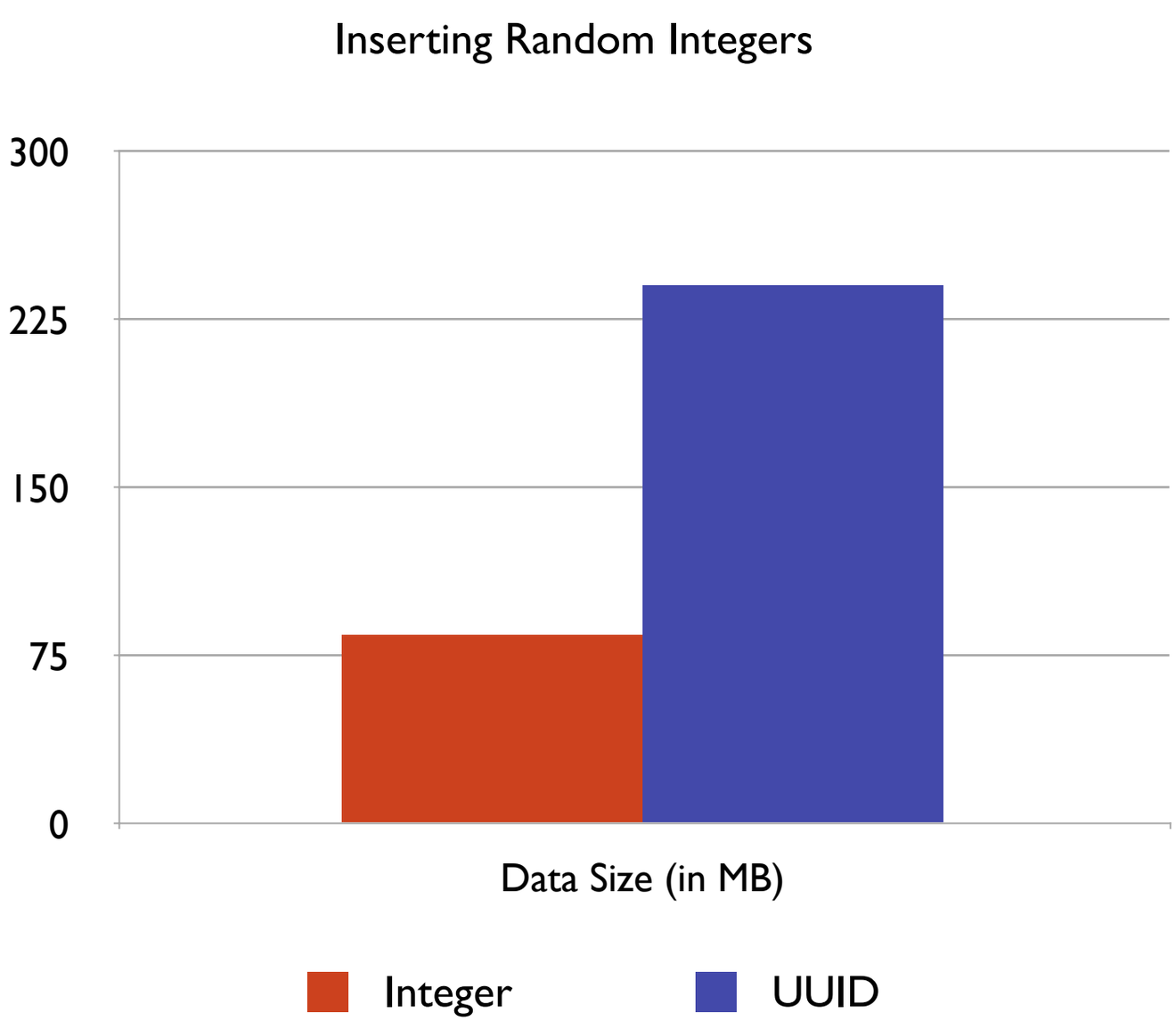
Our results (typical case)




```
CREATE TABLE uuid_users (
 PRIMARY KEY,
 emailaddress varchar(100),
 firstname varchar(20),
 lastname varchar(20),
 birthday varchar(10),
 occupation varchar(70),
 INDEX(emailaddress),
 INDEX(lastname, firstname),
 INDEX(occupation)
) ENGINE=InnoDB;
```

The UUID primary key makes the table about 65% larger.

Our results (worst case)




```
CREATE TABLE mydata (
 PRIMARY KEY,
 col1 INT NOT NULL,
 col2 INT NOT NULL,
 col3 INT NOT NULL,
 col4 INT NOT NULL,
 col5 INT NOT NULL,
 INDEX (col1),
 INDEX (col2),
 INDEX (col3),
 INDEX (col4),
 INDEX (col5)
) ENGINE=InnoDB;
```

The UUID primary key makes the table almost x3!

Hot column on a wide table

```
•CREATE TABLE users (
   ID INTEGER,
   first_name VARCHAR(60),
   last_name VARCHAR(60),
   email VARCHAR(100),
   ..
   phone_number varchar(20),
   last_login_date DATE
);
```

Hot column on a wide table

- Solutions & Workarounds:
  - Move `user_id` and `last_login_date` to another table (good for reads and writes).
  - Use a covering index (better for situations where read heavy).

Hot column on a wide table

- Another example of this problem is with the ‘view count’ on an item.
- For this, writing to memcached and only pushing down to MySQL on every *nth* write may be required.
- Denormalization might not buy you enough time.

Over-indexed tables

- Infrequently used indexes can be responsible for decreasing write capacity.
  - more data in buffer pool
  - more disk IO
  - more time to update
- For reads, the optimizer has more choices to make and a more difficult decision process.

Under-indexed Tables

- Under-indexed tables can result in too many rows needing to be examined after an index has been used - or in the worst case, no index used.
  - This can cause contention on what contents you are able to keep in memory - and it will likely increase the size of your working set.

What makes a good schema?

What is good?

- It all depends on the queries you send to it.
  - i.e. if you can't add a very effective index, you need to make changes.

Best way to Design Schema

- Use a program where you can map out each of the objects on an ER diagram.
  - i.e. MySQL Workbench.
- Think ahead and see if there are any common access patterns which do not fit well.
  - i.e. I always want to know the total amount of the invoice without having to sum up all the invoice items.
- Export the ER diagram to SQL.

Can you make schema better?

- It is very hard to retrofit into an Application.
- For some obvious bad-choices, the ‘band aid’ approach may work.
  - This command shows the most optimal data type:

```
SELECT * FROM title PROCEDURE ANALYSE(1,1)\G
```

PROCEDURE ANALYZE

```
*************************** 1. row ***************************
             Field_name: imdb.title.id
              Min_value: 1
              Max_value: 1543720
             Min_length: 1
             Max_length: 7
       Empties_or_zeros: 0
                  Nulls: 0
Avg_value_or_avg_length: 771860.5411
                    Std: 891266.7873
      Optimal_fieldtype: MEDIUMINT(7) UNSIGNED NOT NULL
*************************** 2. row ***************************
             Field_name: imdb.title.title
              Min_value: # 1997 Honda Accord: Gauges Upgrade -
              Max_value: Þröng sýn
             Min_length: 1
             Max_length: 334
       Empties_or_zeros: 0
                  Nulls: 0
Avg_value_or_avg_length: 16.4844
                    Std: NULL
      Optimal_fieldtype: TEXT NOT NULL
```

MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- **Identifying Bad Queries**

Identifying Bad Queries

- Slowlog
- Logging in the Application
- *-Proxy
- MySQL Query Analyzer

http://www.mysql.com/products/enterprise/query.html

- `pt-query-digest`

http://www.percona.com/doc/percona-toolkit/2.0/pt-query-digest.html

pt-query-digest

- generate reports from
  - slow query log
    `pt-query-digest /path/to/slow.log`
  - binlog files
  - processlist
  - postgresql log files
  - general log (not so useful)
  - tcpdump files that captured traffic from: mysql, memcached, http
- store reports in db: `--review,--review-history`
- enhanced filtering capabilities
  `'$event->{fingerprint} =~ m/^select/'`

pt-query-digest

```
# 834.7s user time, 9.1s system time, 302.78M rss, 392.96M vsz
# Current date: Mon Nov 29 09:47:43 2010
# Hostname: servername
# Files: STDIN
# Overall: 670.66k total, 1.73k unique, 955.33 QPS, 3.08x concurrency _____
# Time range: 2010-11-29 09:14:29.955239 to 09:26:11.979320
# Attribute          total     min     max     avg     95%  stddev  median
# ============     ======= ======= ======= ======= ======= ======= =======
# Exec time          2163s       0       3s      3ms      2ms     29ms     89us
# Rows affecte       18.58k       0      146    0.03        0    0.49        0
# Query size        121.29M       6   21.55k  189.64   363.48   328.74    97.36
# Warning coun       438.18k       0   25.60k    0.67        0   122.19        0
# Boolean:
# No good inde   0% yes,  99% no
# No index use  10% yes,  89% no
```

```
# Profile
# Rank Query ID           Response time    Calls  R/Call   Apdx V/M   Item
# ==== ================== ================ ====== ======== ==== ===== ====
#    1 0x3928FBFF36663F33  1349.6240 62.4%  11976   0.1127 1.00  0.03 SELECT
loan_officer_states
#    2 0x8A539A15CDC891EB   114.9014  5.3%    437   0.2629 1.00  0.50 SELECT
processing_assigned
#    3 0xFA5D75AB1925777C    92.9441  4.3%    791   0.1175 1.00  0.06 SELECT security_dashboard
#    4 0x6F1DB5CAB019DB16    77.5712  3.6%     43   1.8040 0.65  0.73 SELECT
#    5 0xDFEC78D47187A0CD    67.1673  3.1%    296   0.2269 1.00  0.17 SELECT history assigned
#    6 0x5D51E5F01B88B79E    49.0330  2.3%  15630   0.0031 1.00  0.00 ADMIN CONNECT
#    7 0xD704F6F4D36804AB    43.4990  2.0%    274   0.1588 1.00  0.12 SELECT user_agents
#    8 0x7EC8CF8EAFC26907    30.0898  1.4%    416   0.0723 1.00  0.07 SELECT security_dashboard
#    9 0x599BEF84DBA12853    19.6506  0.9%  13424   0.0015 1.00  0.01 UPDATE user_sessions
#   10 0x19EE1A1A48A2B249    18.8828  0.9%  54835   0.0003 1.00  0.00 SELECT leads contact_info
#   11 0xDD930BC5FC65A135    18.6386  0.9%  54975   0.0003 1.00  0.00 SELECT history
#   12 0x277A0E5B9646746B    16.2016  0.7%  55280   0.0003 1.00  0.00 SELECT history
#   13 0x522C69BD415338C6    13.5388  0.6%    300   0.0451 1.00  0.02 SELECT history assigned
#   14 0xA018F3BA9E66B42B    13.5138  0.6%     41   0.3296 1.00  0.00 SELECT new_rate_locks
#   15 0x59F9E8645FFF4A16    12.7311  0.6%  55331   0.0002 1.00  0.00 SELECT realtor_leads
#   16 0xEE18B363E8DB0222    10.6596  0.5%    161   0.0662 1.00  0.11 SELECT
#   17 0xDF78E27C3290E5F2    10.2883  0.5%    345   0.0298 1.00  0.01 SELECT history lo_history
#   18 0x0C82802FC73439D3    10.0459  0.5%      9   1.1162 0.67  0.20 SELECT users help_history
#   19 0x5462226BD2AF82D9     7.1391  0.3%     75   0.0952 1.00  0.16 SELECT tasks task_note
#   20 0x177159F6BEA4126A     6.7048  0.3%  55342   0.0001 1.00  0.01 SELECT nb_alert_notes
# MISC 0xMISC               179.8054  8.3% 350684   0.0005   NS   0.0 <1713 ITEMS>
```

pt-query-digest

```
# Query 1: 17.06 QPS, 1.92x concurrency, ID 0x3928FBFF36663F33 at byte 14174
# This item is included in the report because it matches --limit.
# Scores: Apdex = 1.00 [1.0], V/M = 0.03
# Time range: 2010-11-29 09:14:30.052415 to 09:26:11.914796
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count          1   11976
# Exec time     62    1350s     25ms    395ms    113ms    219ms     54ms     91ms
# Rows affecte   0       39       0       35     0.00        0     0.32        0
# Query size    23  28.75M   2.46k   2.46k   2.46k   2.38k        0   2.38k
# Warning coun  11  51.51k       0  12.80k     4.40        0  233.99        0
# Boolean:
# No index use  99% yes,   0% no
# String:
# Databases
# Errors        none (273/99%), #1064 (1/0%)
# Hosts         172.20.101.178
# Users         dbuser
```

pt-query-digest

```
# Query_time distribution
#   1us
#  10us  ##############################################
# 100us  ###########
#   1ms  ##
#  10ms  #
# 100ms  ################################################
#    1s
#  10s+
# Tables
#    SHOW TABLE STATUS LIKE 'user_agents'\G
#    SHOW CREATE TABLE `user_agents`\G
# EXPLAIN /*!50100 PARTITIONS*/
SELECT user_agent_id, search_engine
FROM user_agents
WHERE user_agent='Mozilla/4.0 (compatible; MSIE 7.0;
Windows NT 5.1; .NET CLR 1.0.3705)'\G
```

```
# Item 1: 3.41 QPS, 0.97x concurrency, ID 0xABCE5AD2A2DD1BA1 at byte 2881246
# This item is included in the report because it matches --limit.
# Scores: Apdex = 0.97 [1.0], V/M = 19.02
# Query_time sparkline: | ^_____|
# Time range: 2011-04-05 16:12:13 to 16:14:45
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count          0     519
# Exec time      2    148s     11us     33s   285ms    53ms       2s    26us
# Lock time      0     5ms       0   334us     9us    66us    32us       0
# Rows sent      0      41       0       1    0.08    0.99    0.27       0
# Rows examine   1   4.97M       0 445.49k   9.80k   5.73k  49.33k       0
# Rows affecte   0       2       0       1    0.00       0    0.06       0
# Rows read      1   2.01M       0 250.47k   3.96k    1.96  27.94k    0.99
# Bytes sent     0 241.20k      11   8.01k  475.89  918.49  689.98  258.32
# Merge passes   0       0       0       0       0       0       0       0
# Tmp tables     0      15       0       1    0.03       0    0.17       0
# Tmp disk tbl   0       3       0       1    0.01       0    0.08       0
# Tmp tbl size   0   4.78k       0   4.78k    9.43       0  211.60       0
# Query size     0 100.95k      19   2.71k  199.17  363.48  206.60  151.03
# InnoDB:
# IO r bytes     0       0       0       0       0       0       0       0
# IO r ops       0       0       0       0       0       0       0       0
# IO r wait      0       0       0       0       0       0       0       0
# pages distin   1  67.99k       0  10.64k   1.26k   3.88k   2.47k   31.70
# queue wait     0       0       0       0       0       0       0       0
# rec lock wai   0       0       0       0       0       0       0       0
# Boolean:
# Filesort       0% yes,  99% no
# Full scan      7% yes,  92% no
# QC Hit        78% yes,  21% no
# Tmp table      2% yes,  97% no
# Tmp table on   0% yes,  99% no
```

pt-query-digest

```
# Tmp tbl size    0    4.78k        0    4.78k     9.43        0  211.60        0
# Query size      0  100.95k       19    2.71k   199.17   363.48  206.60   151.03
# InnoDB:
# IO r bytes      0        0        0        0        0        0       0        0
# IO r ops        0        0        0        0        0        0       0        0
# IO r wait       0        0        0        0        0        0       0        0
# pages distin    1   67.99k        0   10.64k    1.26k    3.88k   2.47k    31.70
# queue wait      0        0        0        0        0        0       0        0
# rec lock wai    0        0        0        0        0        0       0        0
# Boolean:
# Filesort         0% yes,  99% no
# Full scan        7% yes,  92% no
# QC Hit          78% yes,  21% no
# Tmp table        2% yes,  97% no
# Tmp table on     0% yes,  99% no
```

Is It Worth it?

- There's always room for improvement
- Ask yourself: Is the change going to have a benefit?
- How much effort does it take to get how much gain?
- Benchmark!
- Instrument!

http://www.percona.com/redir/files/white-papers/goal-driven-performance-optimization.pdf

MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries

Kenny Gryp
<kenny.gryp@percona.com>
@gryp

We're Hiring!
www.percona.com/about-us/careers/