MySQL 8.0 Optimizer Guide

Morgan Tocker
MySQL Product Manager (Server)

ORACLE

MySQL


ORACLE®

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda

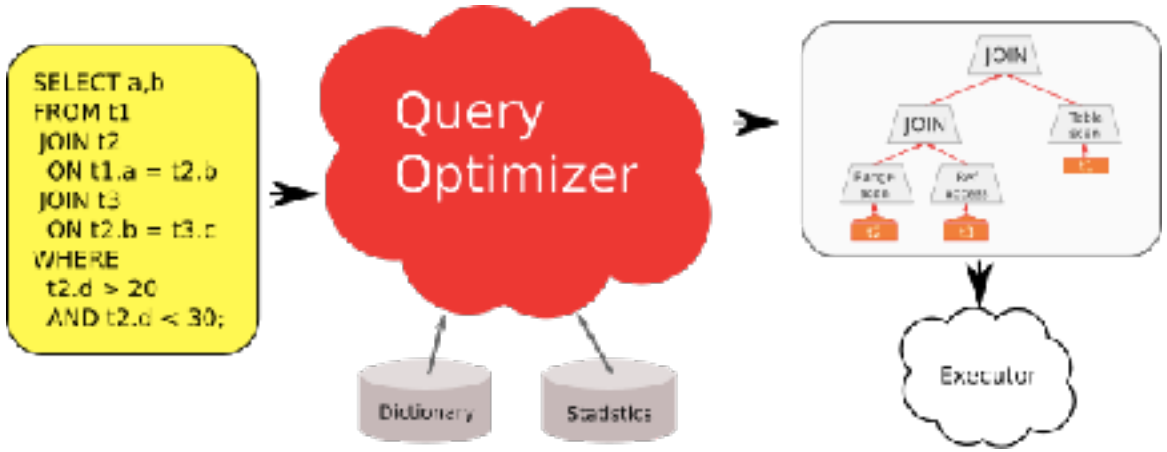
1. **Introduction**
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Introduction

- SQL is declarative
- You state *“what you want”* not *“how you want”*
- Can’t usually sight check queries to understand execution efficiency
- Database management system is like a GPS navigation system. It finds the “best”route.

GPS…

MySQL Optimizer



Diagnostic Commands

- EXPLAIN (all versions)
- EXPLAIN FORMAT=JSON (5.6+)
  - Supported by Workbench in Visual format
- Optimizer Trace (5.6+)

Examples from “The World Schema”

- Contains Cities, Countries, Language statistics
- Download from:
  - https://dev.mysql.com/doc/index-other.html
- Very small data set
  - Good for learning
  - Not good for explaining performance differences

Primary Table we are using

```
CREATE TABLE `Country` (
  `Code` char(3) NOT NULL DEFAULT '',
  `Name` char(52) NOT NULL DEFAULT '',
  `Continent` enum('Asia','Europe','North America','Africa','Oceania','Antarctica','South America') NOT NULL DEFAULT
'Asia',
  `Region` char(26) NOT NULL DEFAULT '',
  `SurfaceArea` float(10,2) NOT NULL DEFAULT '0.00',
  `IndepYear` smallint(6) DEFAULT NULL,
  `Population` int(11) NOT NULL DEFAULT '0',
  `LifeExpectancy` float(3,1) DEFAULT NULL,
  `GNP` float(10,2) DEFAULT NULL,
  `GNPOld` float(10,2) DEFAULT NULL,
  `LocalName` char(45) NOT NULL DEFAULT '',
  `GovernmentForm` char(45) NOT NULL DEFAULT '',
  `HeadOfState` char(60) DEFAULT NULL,
  `Capital` int(11) DEFAULT NULL,
  `Code2` char(2) NOT NULL DEFAULT '',
  PRIMARY KEY (`Code`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

Companion Website

- Content from “The Unofficial MySQL 8.0 Optimizer Guide”
  - http://www.unofficialmysqlguide.com/
- More detailed text for many of the examples here…
- Most still applies to 5.6+
  - EXPLAIN FORMAT=JSON in 5.6 does not show cost
  - Costs will be different
  - Output from Optimizer Trace may differ
  - Some features will be missing

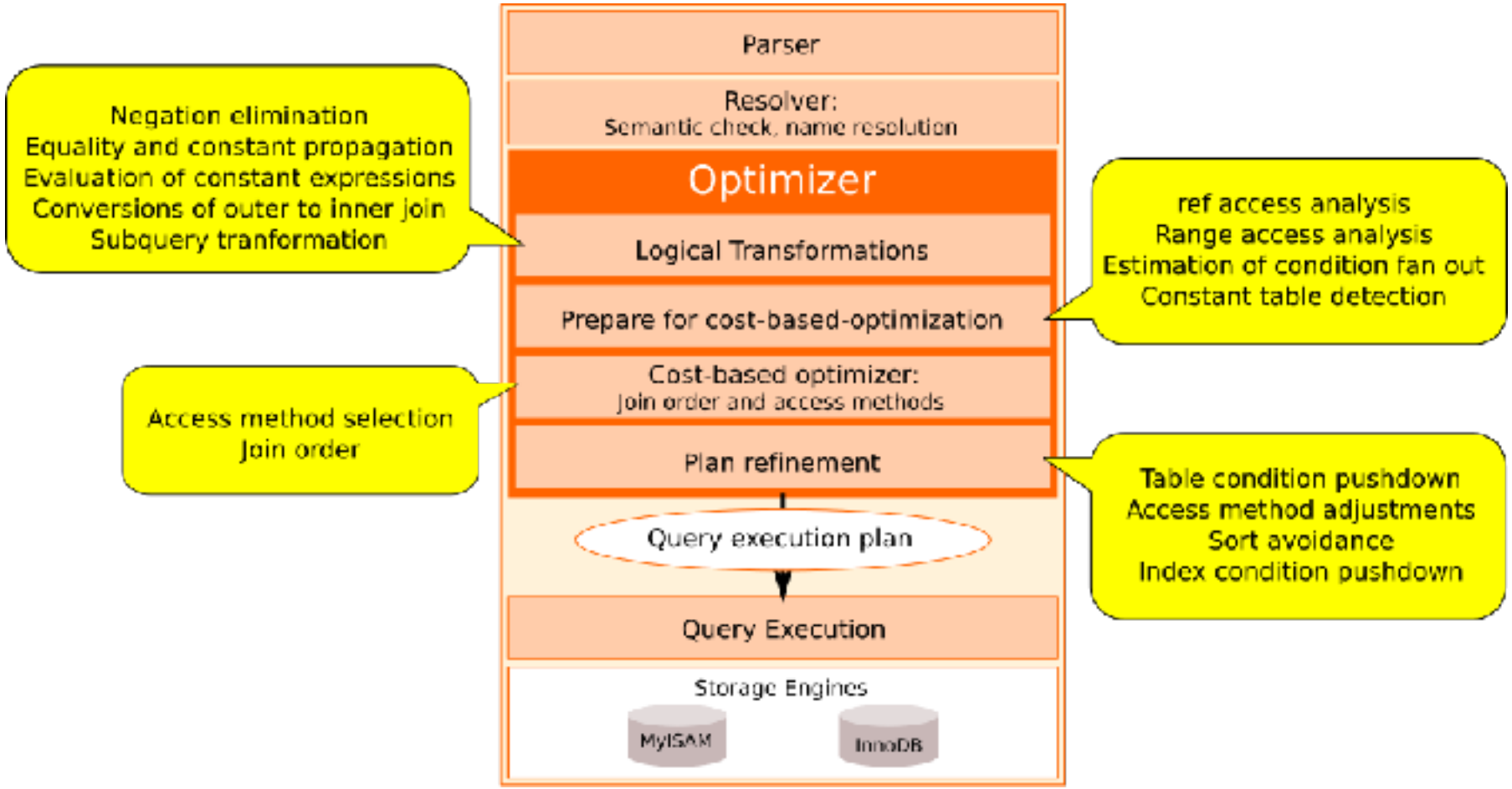
Danger: Code on slides!

- Some examples may appear small
- Please feel free to download this deck from:
  - https://www.slideshare.net/morgo/mysql-80-optimizer-guide
- Follow along on your laptop

Agenda

1. Introduction
2. **Server Architecture**
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Server Architecture



Just the Important Parts

- Comprised of the Server and Storage Engines
- Query Optimization happens at the Server Level
- Semantically there are four stages of Query Optimization
- Followed by Query Execution

Agenda

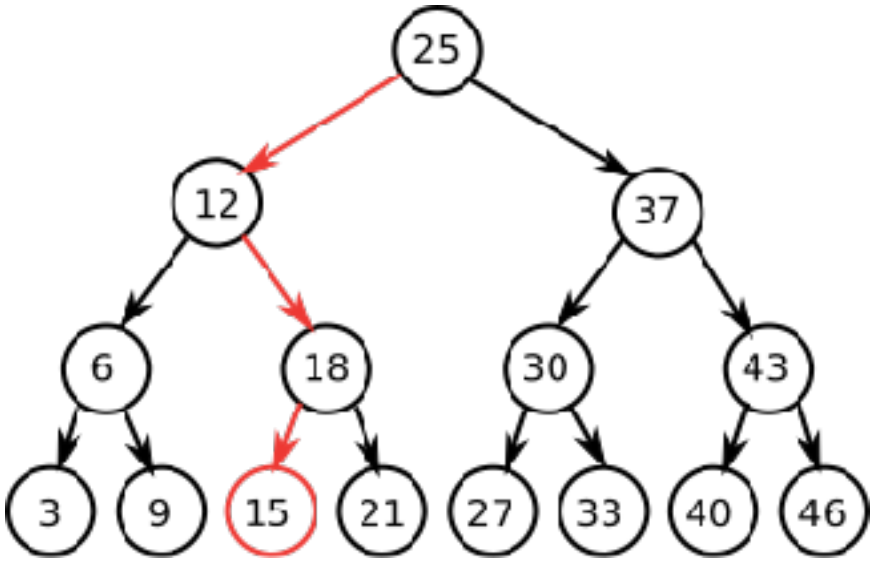
1. Introduction
2. Server Architecture
3. **B+trees**
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

B+trees

- When we mean "*add an index*" we usually mean "*add a B+tree index*":
  - Includes `PRIMARY, UNIQUE, INDEX` type indexes.
- Understanding the basic structure of B+trees helps with optimization

Binary Tree

- Not the same as a B+tree
- Understand Binary Tree first then compare and contrast



Locate `829813` in a (balanced) binary tree of 1MM ~= 20 hops.

is this good?

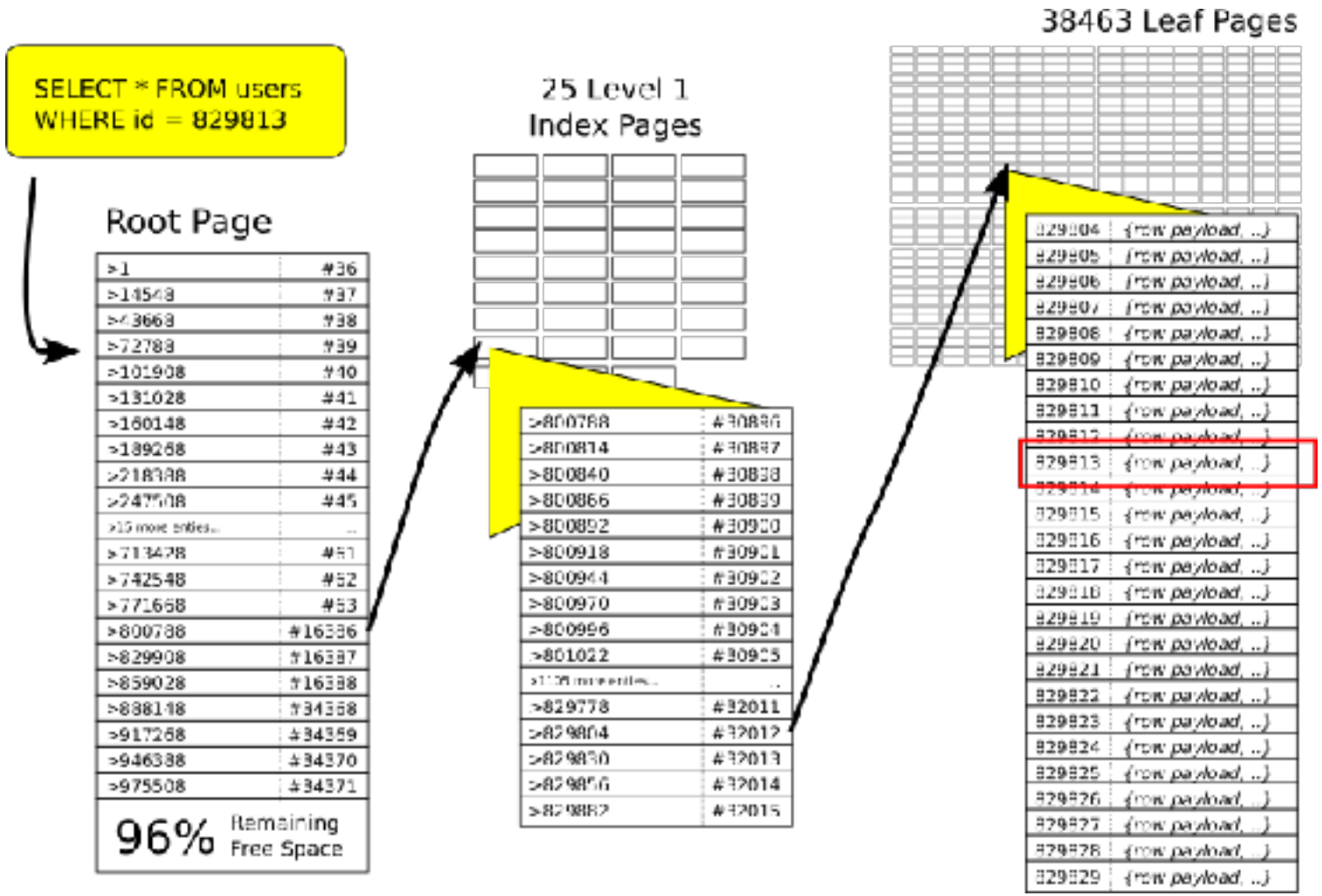
B+tree

- Amortizes disk accesses by clustering into pages:
- Can achieve same outcome in **two hops:**

```
CREATE TABLE users (
id INT NOT NULL auto_increment,
username VARCHAR(32) NOT NULL,
payload TEXT,
PRIMARY KEY (id)
);
```

B+tree

- Amortizes disk accesses by clustering into pages
- Can achieve same outcome in **two hops:**



B-trees are wide not deep

- From the root page: values >= 800788 but < 829908 are on page 16386.
- From page 16386: values >= 829804 but < 829830 are on **leaf page** 32012.
- Large fan out factor; 1000+ keys/page which point to another index page with 1000+ keys/page

InnoDB uses a Clustered Index

- In InnoDB the data rows are also stored in a B+tree, organized by the primary key
- Secondary key indexes always include the value of the primary key

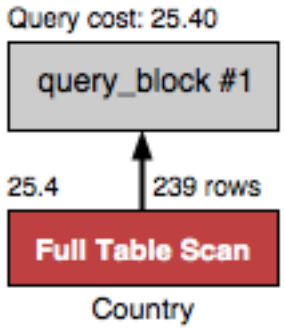
Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. **EXPLAIN**
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

EXPLAIN

- Pre-execution view of how MySQL *intends* to execute a query
- Prints what MySQL considers *the best plan* after a process of considering potentially thousands of choices

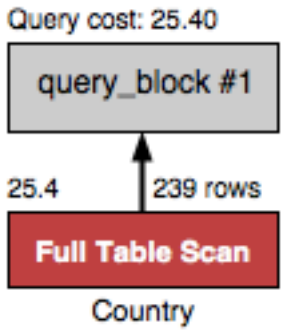
```
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE continent='Asia' and population > 5000000;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "25.40"
    },
    "table": {
      "table_name": "country",
      "access_type": "ALL",
      "rows_examined_per_scan": 239,
      "rows_produced_per_join": 11,
      "filtered": "6.46",
..
      "attached_condition": "((`world`.`country`.`Continent` = 'Asia')
and (`world`.`country`.`Population` > 5000000))"
    }
  }
}
```



What indexes will make this query faster?

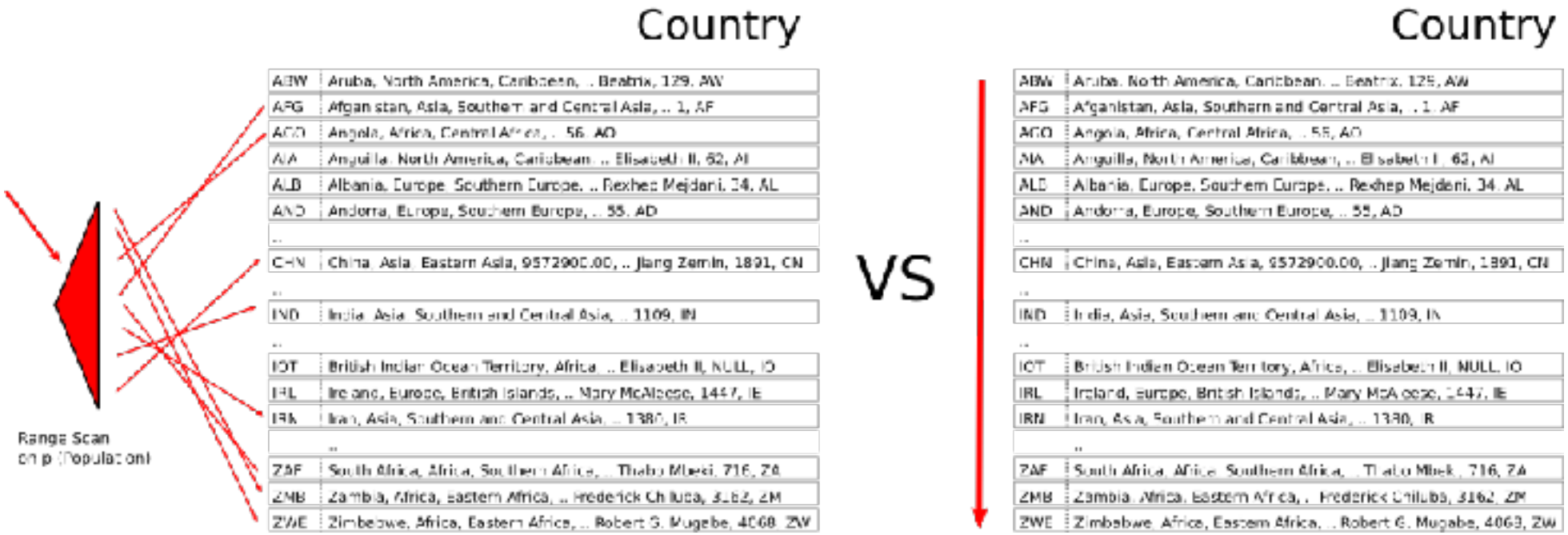
- Some Suggestions:
  - Index on p (population)
  - Index on c (continent)
  - Index on p_c (population, continent)
  - Index on c_p (continent, population)

```
ALTER TABLE Country ADD INDEX p (population);
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE continent='Asia' and population > 5000000;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "25.40"
    },
    "table": {
      "table_name": "Country",
      "access_type": "ALL",
      "possible_keys": [
        "p"
      ],
      "rows_examined_per_scan": 239,
      "rows_produced_per_join": 15,
      "filtered": "6.46",
..
      "attached_condition": "((`world`.`country`.`Continent` = 'Asia') and
(`world`.`country`.`Population` > 5000000))"
    ..
```



Why would an index not be used?

```
SELECT * FROM Country
WHERE continent='Asia'  AND population > 5000000;
```



Agenda

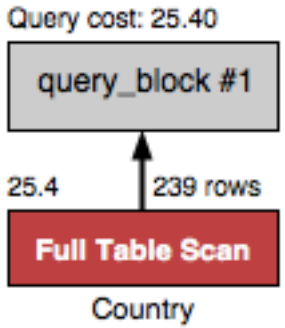
1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. **Optimizer Trace**
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Optimizer Trace

- What other choices did EXPLAIN not show?
- Why was that choice made?
- Output is quite verbose

```
ALTER TABLE Country ADD INDEX p (population);
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE continent='Asia' and population > 5000000;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "25.40"
    },
    "table": {
      "table_name": "Country",
      "access_type": "ALL",
      "possible_keys": [
        "p"
      ],
      "rows_examined_per_scan": 239,
      "rows_produced_per_join": 15,
      "filtered": "6.46",
      "cost_info": {
        "read_cost": "23.86",
        "eval_cost": "1.54",
        "prefix_cost": "25.40",
        "data_read_per_join": "3K"
      },
..
      "attached_condition": "((`world`.`country`.`Continent` = 'Asia') and
(`world`.`country`.`Population` > 5000000))"
..
```

It's available but not used. Why?



```
SET optimizer_trace="enabled=on";
SELECT * FROM Country WHERE continent='Asia' and population > 5000000;
SELECT * FROM information_schema.optimizer_trace;
{
  "steps": [
    {
      "join_preparation": {
        "select#": 1,
        "steps": [
          {
            "expanded_query": "/* select#1 */ select `country`.`Code` AS `Code`,`country`.`Name` AS
`Name`,`country`.`Continent` AS `Continent`,`country`.`Region` AS `Region`,`country`.`SurfaceArea` AS
`SurfaceArea`,`country`.`IndepYear` AS `IndepYear`,`country`.`Population` AS
`Population`,`country`.`LifeExpectancy` AS `LifeExpectancy`,`country`.`GNP` AS
`GNP`,`country`.`GNPOld` AS `GNPOld`,`country`.`LocalName` AS `LocalName`,`country`.`GovernmentForm`
AS `GovernmentForm`,`country`.`HeadOfState` AS `HeadOfState`,`country`.`Capital` AS
`Capital`,`country`.`Code2` AS `Code2` from `country` where ((`country`.`Continent` = 'Asia') and
(`country`.`Population` > 5000000))"
          }
        ]
      }
    },
    {
      "join_optimization": {
        "select#": 1,
        "steps": [
          {
            "condition_processing": {
              "condition": "WHERE",
```

```
            "original_condition": "((`country`.`Continent` = 'Asia') and (`country`.`Population` > 5000000))",
            "steps": [
              {
                "transformation": "equality_propagation",
                "resulting_condition": "((`country`.`Population` > 5000000) and multiple equal('Asia',
`country`.`Continent`))"
              },
              {
                "transformation": "constant_propagation",
                "resulting_condition": "((`country`.`Population` > 5000000) and multiple equal('Asia',
`country`.`Continent`))"
              },
              {
                "transformation": "trivial_condition_removal",
                "resulting_condition": "((`country`.`Population` > 5000000) and multiple equal('Asia',
`country`.`Continent`))"
              }
            ]
          }
        },
        {
          "substitute_generated_columns": {
          }
        },
        {
          "table_dependencies": [
            {
              "table": "`country`",
              "row_may_be_null": false,
              "map_bit": 0,
```

```
        "depends_on_map_bits": [
        ]
      }
    ]
  },
  {
    "ref_optimizer_key_uses": [
    ]
  },
  {
    "rows_estimation": [
      {
        "table": "`country`",
        "range_analysis": {
          "table_scan": {
            "rows": 239,
            "cost": 27.5
          },
          "potential_range_indexes": [
            {
              "index": "PRIMARY",
              "usable": false,
              "cause": "not_applicable"
            },
            {
              "index": "p",
              "usable": true,
              "key_parts": [
                "Population",
                "Code"
              ]
```

```
      }
    ],
    "setup_range_conditions": [
    ],
    "group_index_range": {
      "chosen": false,
      "cause": "not_group_by_or_distinct"
    },
    "analyzing_range_alternatives": {
      "range_scan_alternatives": [
        {
          "index": "p",
          "ranges": [
            "5000000 < Population"
          ],
          "index_dives_for_eq_ranges": true,
          "rowid_ordered": false,
          "using_mrr": false,
          "index_only": false,
          "rows": 108,
          "cost": 38.06,
          "chosen": false,
          "cause": "cost"
        }
      ],
      "analyzing_roworder_intersect": {
        "usable": false,
        "cause": "too_few_roworder_scans"
      }
    }
}
```

```
            }
          ]
        },
        {
          "considered_execution_plans": [
            {
              "plan_prefix": [
              ],
              "table": "`country`",
              "best_access_path": {
                "considered_access_paths": [
                  {
                    "rows_to_scan": 239,
                    "access_type": "scan",
                    "resulting_rows": 239,
                    "cost": 25.4,
                    "chosen": true
                  }
                ]
              },
              "condition_filtering_pct": 100,
              "rows_for_plan": 239,
              "cost_for_plan": 25.4,
              "chosen": true
            }
          ]
        },
        {
          "attaching_conditions_to_tables": {
            "original_condition": "((`country`.`Continent` =
'Asia') and (`country`.`Population` > 5000000))",
```

```
            "attached_conditions_computation": [
            ],
            "attached_conditions_summary": [
              {
                "table": "`country`",
                "attached": "((`country`.`Continent` = 'Asia') and (`country`.`Population` > 5000000))"
              }
            ]
          }
        },
        {
          "refine_plan": [
            {
              "table": "`country`"
            }
          ]
        }
      ]
    }
  },
  {
    "join_execution": {
      "select#": 1,
      "steps": [
      ]
    }
  }
]
}
```

Why would an index not be used?

OPTIMIZER TRACE:

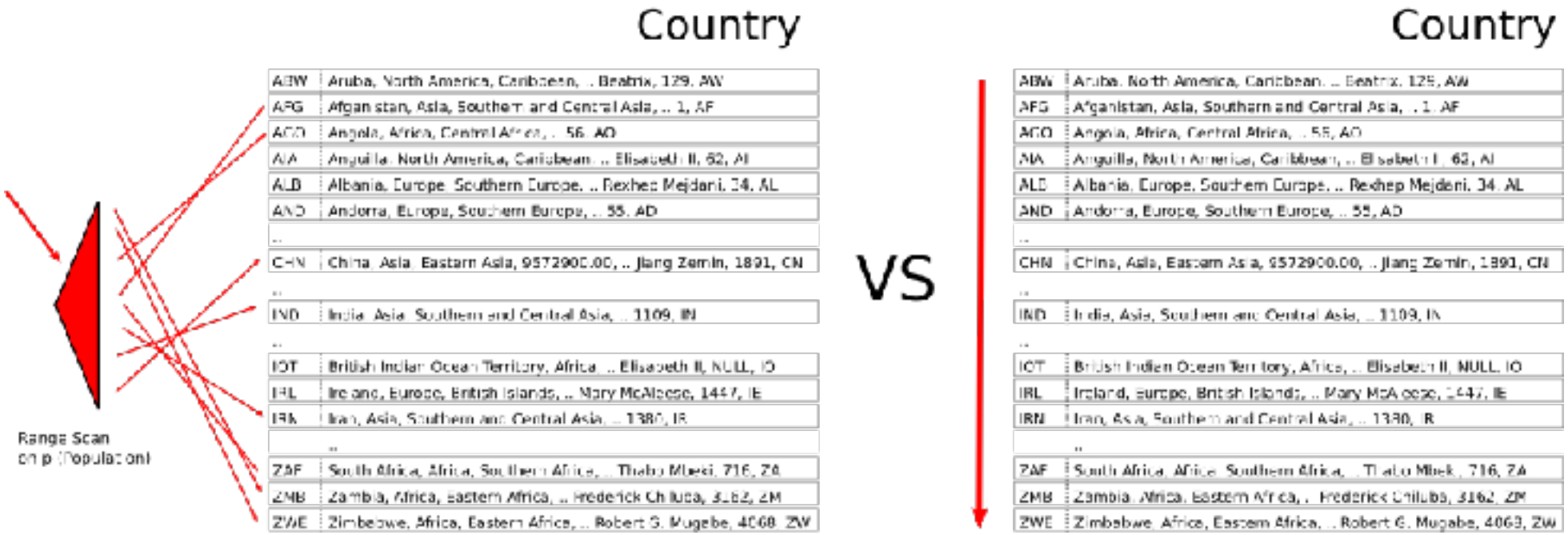
```
"analyzing_range_alternatives": {
  "range_scan_alternatives": [
    {
      "index": "p",
      "ranges": [
        "5000000 < Population"
      ],
      "index_dives_for_eq_ranges": true,
      "rowid_ordered": false,
      "using_mrr": false,
      "index_only": false,
      "rows": 108,
      "cost": 38.06,
      "chosen": false,
      "cause": "cost"
    }
  ],
```

FORCE INDEX (p):

```
..
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "48.86"
    },
    "table": {
      "table_name": "Country",
      "access_type": "range",
      "possible_keys": [
        "p"
      ],
      "key": "p",
..
```

Reason again…

```
SELECT * FROM Country
WHERE continent='Asia'  AND population > 5000000;
```

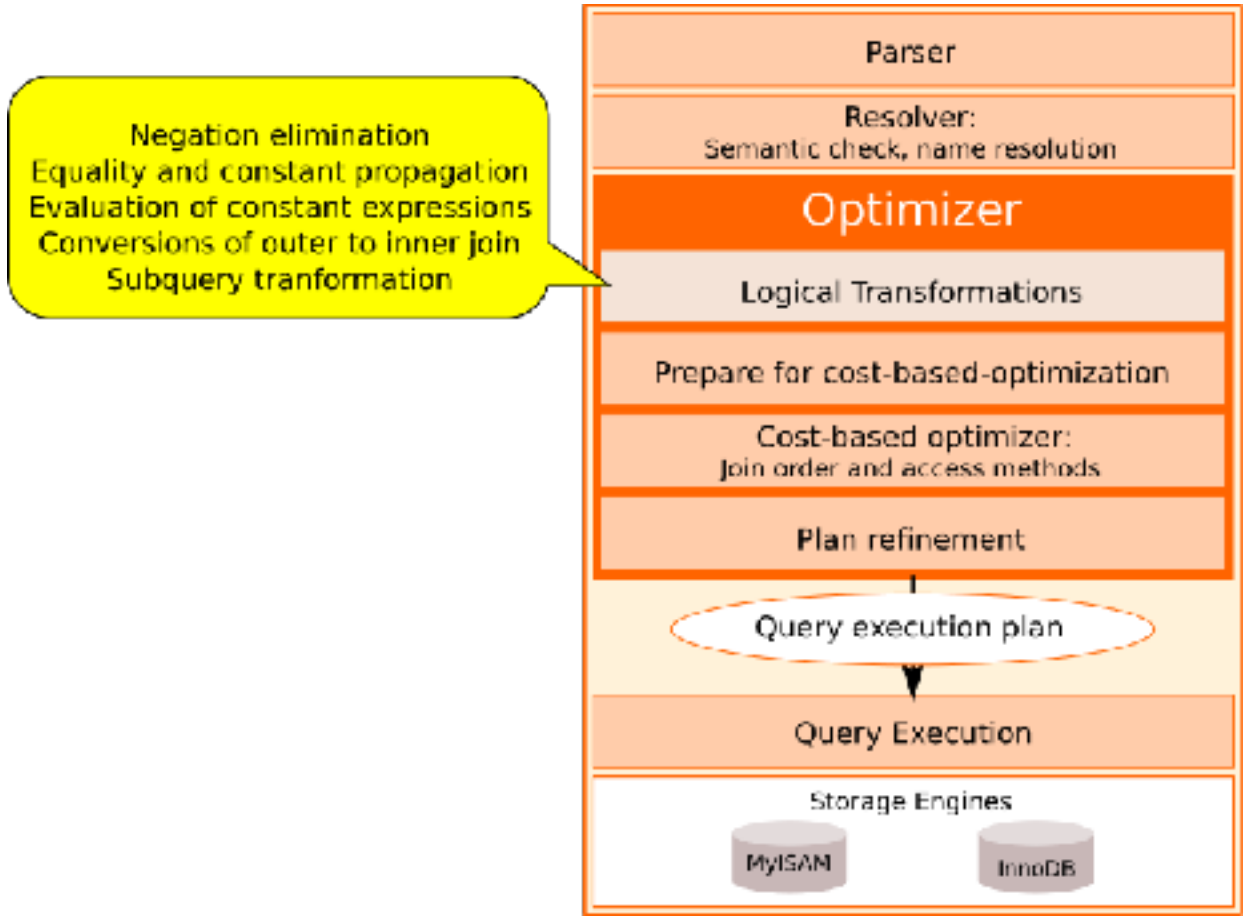


Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. **Logical Transformations**
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Logical Transformations

- First part of optimization is eliminating unnecessary work



Why eliminate unnecessary work?

- Short-cut/reduce number of execution plans that need to be evaluated
- Transform parts of queries to take advantage of better execution strategies
- Think of a how a compiler transforms code to be more efficient
  - MySQL does similar at runtime

Example:

```
SELECT * FROM Country
WHERE population > 5000000 AND continent='Asia'
AND 1=1;
```

SHOW WARNINGS says:

```
EXPLAIN FORMAT=JSON SELECT * FROM Country WHERE population > 5000000 AND 1=1;
```

```
SHOW WARNINGS;
```

```
/* select#1 */ select
`world`.`Country`.`Code` AS `Code`,
`world`.`Country`.`Name` AS `Name`,
`world`.`Country`.`Continent` AS `Continent`,
`world`.`Country`.`Region` AS `Region`,
`world`.`Country`.`SurfaceArea` AS `SurfaceArea`,
`world`.`Country`.`IndepYear` AS `IndepYear`,
`world`.`Country`.`Population` AS `Population`,
`world`.`Country`.`LifeExpectancy` AS `LifeExpectancy`,
`world`.`Country`.`GNP` AS `GNP`,
`world`.`Country`.`GNPOld` AS `GNPOld`,
`world`.`Country`.`LocalName` AS `LocalName`,
`world`.`Country`.`GovernmentForm` AS `GovernmentForm`,
`world`.`Country`.`HeadOfState` AS `HeadOfState`,
`world`.`Country`.`Capital` AS `Capital`,
`world`.`Country`.`Code2` AS `Code2`
from `world`.`Country`
where (`world`.`Country`.`Population` > 5000000)
```

OPTIMIZER TRACE says:

```
..
        "steps": [
          {
            "condition_processing": {
              "condition": "WHERE",
              "original_condition": "((`Country`.`Population` > 5000000) and (1 = 1))",
              "steps": [
                {
                  "transformation": "equality_propagation",
                  "resulting_condition": "((`Country`.`Population` > 5000000) and (1 = 1))"
                },
                {
                  "transformation": "constant_propagation",
                  "resulting_condition": "((`Country`.`Population` > 5000000) and (1 = 1))"
                },
                {
                  "transformation": "trivial_condition_removal",
                  "resulting_condition": "(`Country`.`Population` > 5000000)"
           ..
```

What sort of transformations can occur?

- Merging views back with definition of base tables
- Derived table in FROM clause merged back into base tables
- Unique subqueries converted directly to INNER JOIN statements
- Primary key lookup converted to constant values.
  - Shortcut plans that will need to be evaluated.

Primary Key Lookup

```
SELECT * FROM Country WHERE code='CAN'
/* select#1 */ select
'CAN' AS `Code`,
'Canada' AS `Name`,
'North America' AS `Continent`,
'North America' AS `Region`,
'9970610.00' AS `SurfaceArea`,
'1867' AS `IndepYear`,
'31147000' AS `Population`,
'79.4' AS `LifeExpectancy`,
'598862.00' AS `GNP`,
'625626.00' AS `GNPOld`,
'Canada' AS `LocalName`,
'Constitutional Monarchy, Federation' AS `GovernmentForm`,
'Elisabeth II' AS `HeadOfState`,
'1822' AS `Capital`,
'CA' AS `Code2`
from `world`.`Country` where 1
```

Primary key does not exist

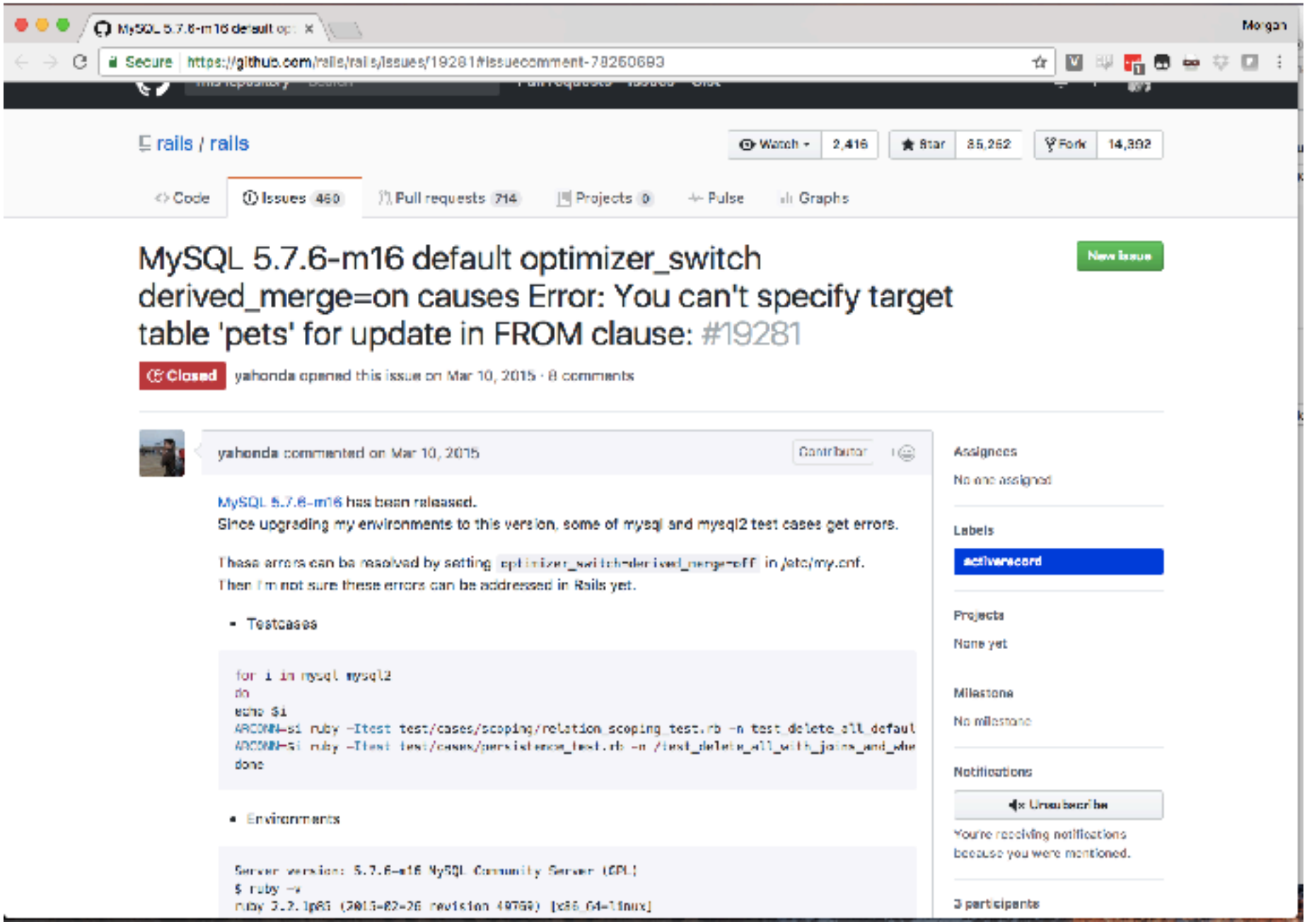
```
SELECT * FROM Country WHERE code='XYZ'

/* select#1 */ select NULL AS `Code`,NULL AS
`Name`,NULL AS `Continent`,NULL AS `Region`, NULL AS
`SurfaceArea`,NULL AS `IndepYear`,NULL AS
`Population`,NULL AS `LifeExpectancy`,NULL AS `GNP`,
NULL AS `GNPOld`,NULL AS `LocalName`,NULL AS
`GovernmentForm`,NULL AS `HeadOfState`,NULL AS
`Capital`, NULL AS `Code2` from `world`.`Country`
where multiple equal('XYZ', NULL)
```

Impossible WHERE

```
SELECT * FROM Country WHERE code='CAN' AND 1=0

/* select#1 */ select `world`.`Country`.`Code` AS
`Code`,`world`.`Country`.`Name` AS `Name`,
`world`.`Country`.`Continent` AS `Continent`,`world`.`Country`.`Region`
AS `Region`, `world`.`Country`.`SurfaceArea` AS
`SurfaceArea`,`world`.`Country`.`IndepYear` AS `IndepYear`,
`world`.`Country`.`Population` AS
`Population`,`world`.`Country`.`LifeExpectancy` AS `LifeExpectancy`,
`world`.`Country`.`GNP` AS `GNP`,`world`.`Country`.`GNPOld` AS
`GNPOld`, `world`.`Country`.`LocalName` AS
`LocalName`,`world`.`Country`.`GovernmentForm` AS `GovernmentForm`,
`world`.`Country`.`HeadOfState` AS
`HeadOfState`,`world`.`Country`.`Capital` AS `Capital`,
`world`.`Country`.`Code2` AS `Code2` from `world`.`Country` where 0
```

Are transformations always safe?

- Yes they should be
- New transformations (and execution strategies) may return *non deterministic* queries in a different order
- Some illegal statements as a result of derived_merge transformation

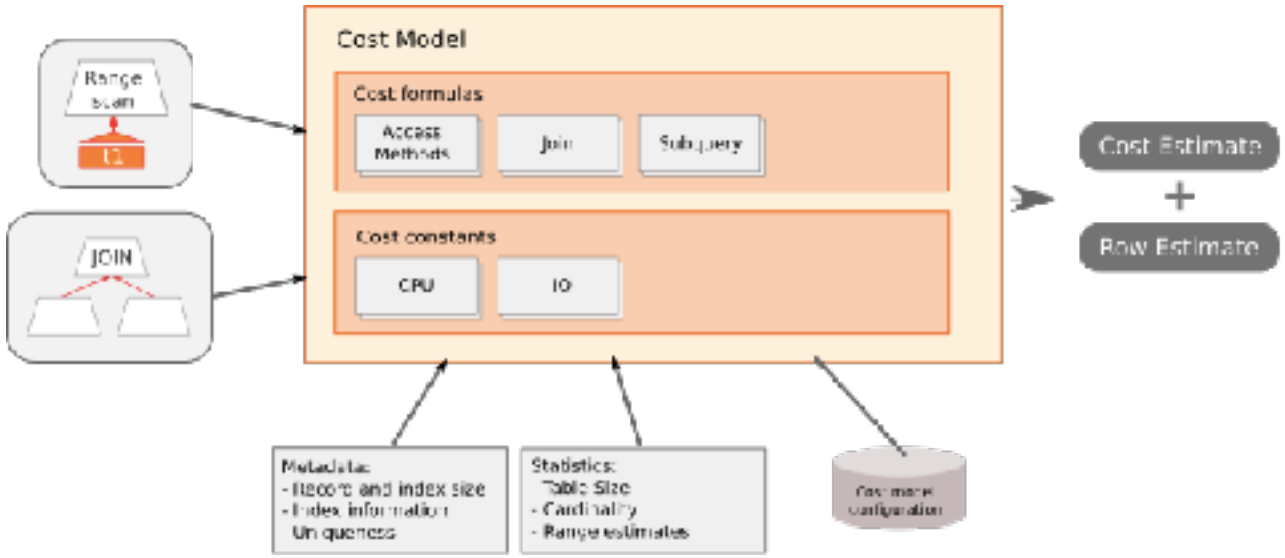

MySQL 5.7.6-m16 default op
Secure
https://github.com/rails/rails/issues/19281#issuecomment-78260693
rails / rails
Watch 2,416
Star 85,262
Fork 14,392
Code
Issues 460
Pull requests 714
Projects 0
Pulse
Graphs
MySQL 5.7.6-m16 default optimizer_switch derived_merge=on causes Error: You can't specify target table 'pets' for update in FROM clause: #19281
New issue
Closed
yahonda opened this issue on Mar 10, 2015 · 8 comments
yahonda commented on Mar 10, 2015
Contributor
MySQL 5.7.6-m16 has been released.
Since upgrading my environments to this version, some of mysql and mysql2 test cases get errors.
These errors can be resolved by setting optimizer_switch=derived_merge=off in /etc/my.cnf.
Then I'm not sure these errors can be addressed in Rails yet.
Testcases
for i in mysql mysql2
do
echo $i
ARCONN=$i ruby -Itest test/cases/scoping/relation_scoping_test.rb -n test_delete_all_defaul
ARCONN=$i ruby -Itest test/cases/persistence_test.rb -n /test_delete_all_with_joins_and_whe
done
Environments
Server version: 5.7.6-m16 MySQL Community Server (GPL)
$ ruby -v
ruby 2.2.1p85 (2015-02-26 revision 49769) [x86_64-linux]
Assignees
No one assigned
Labels
activerecord
Projects
None yet
Milestone
No milestone
Notifications
Unsubscribe
You're receiving notifications because you were mentioned.
3 participants

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. **Cost Based Optimization**
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Query Optimizer Strategy

- Model each of the possible execution plans (using support from statistics and meta data)
- Pick the plan with the lowest cost



Model you say?

1. Assign a cost to each operation
2. Evaluate how many operations each possible plan would take
3. Sum up the total
4. Choose the plan with the lowest overall cost

How are statistics calculated?

- Dictionary Information
- Cardinality Statistics
- Records In Range Dynamic Sampling
- Table Size

Example Model: Table Scan

*SELECT * FROM Country WHERE continent='Asia' and population > 5000000;*

IO Cost:
pages in table * (IO_BLOCK_READ_COST | MEMORY_BLOCK_READ_COST)

CPU Cost:
records * ROW_EVALUATE_COST

Defaults:
IO_BLOCK_READ_COST = 1
MEMORY_BLOCK_READ_COST = 0.25
ROW_EVALUATE_COST=0.1

Values:
pages in table = 6
records = 239

100% on Disk:
= (6 * 1) + (0.1 * 239)
= 29.9

100% in Memory:
= (6 * 0.25) + (0.1 * 239)
= 25.4

New! MySQL 8.0 estimates how many of the pages will be in memory.

SELECT clust_index_size from INNODB_SYS_TABLESTATS WHERE name='world/country'

EXPLAIN said cost was 25.40

Example Model: Range Scan

*SELECT * FROM Country WHERE continent='Asia' and* ***population > 5000000;***

IO Cost:
records_in_range * (IO_BLOCK_READ_COST | MEMORY_BLOCK_READ_COST)

CPU Cost:
records_in_range * ROW_EVALUATE_COST
+ # records_in_range * ROW_EVALUATE_COST

Evaluate range condition

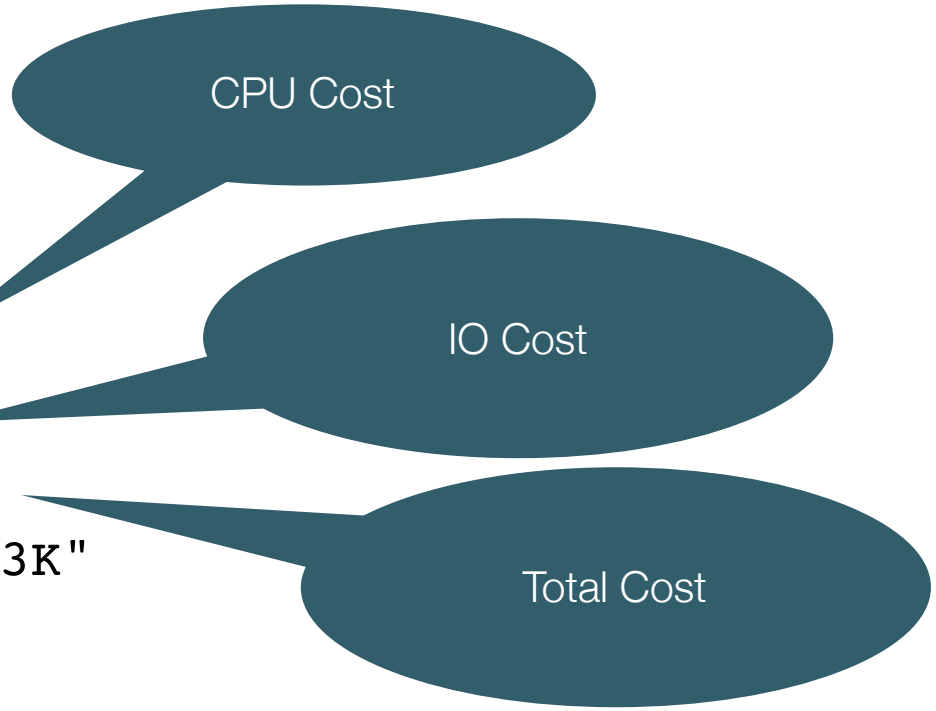
Evaluate WHERE condition

= (108 * 0.25) + ((108 * 0.1) + (108 * 0.1))
= 48.6

Compares to "query_cost": "48.86" in EXPLAIN.

100% in memory.
On disk = 129.6

```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "25.40"
    },
    "table": {
      "table_name": "country",
      "access_type": "ALL",
      "possible_keys": [
        "p"
      ],
..
      "cost_info": {
        "read_cost": "23.86",
        "eval_cost": "1.54",
        "prefix_cost": "25.40",
        "data_read_per_join": "3K"
      },
..
```



Cost Constant Refinement

```
select * from mysql.server_cost;
+------------------------------+------------+---------------------+---------+---------------+
| cost_name                    | cost_value | last_update         | comment | default_value |
+------------------------------+------------+---------------------+---------+---------------+
| disk_temptable_create_cost   |       NULL | 2017-04-14 16:01:42 | NULL    |            20 |
| disk_temptable_row_cost      |       NULL | 2017-04-14 16:01:42 | NULL    |           0.5 |
| key_compare_cost             |       NULL | 2017-04-14 16:01:42 | NULL    |          0.05 |
| memory_temptable_create_cost |       NULL | 2017-04-14 16:01:42 | NULL    |             1 |
| memory_temptable_row_cost    |       NULL | 2017-04-14 16:01:42 | NULL    |           0.1 |
| row_evaluate_cost            |       NULL | 2017-04-14 16:01:42 | NULL    |           0.1 |
+------------------------------+------------+---------------------+---------+---------------+
6 rows in set (0.00 sec)

select * from mysql.engine_cost\G
*************************** 1. row ***************************
  engine_name: default
  device_type: 0
    cost_name: io_block_read_cost
   cost_value: NULL
  last_update: 2017-04-14 16:01:42
      comment: NULL
default_value: 1
```

Cost Constant Refinement

```
UPDATE mysql.server_cost SET cost_value=1 WHERE cost_name='row_evaluate_cost';
UPDATE mysql.engine_cost set cost_value = 1;
FLUSH OPTIMIZER_COSTS;
EXPLAIN FORMAT=JSON SELECT * FROM Country WHERE continent='Asia' and population > 5000000;

{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "245.00"
    },
    "table": {
      "table_name": "Country",
      "access_type": "ALL",
..
```

Increase row evaluate cost from 0.1 to 1. Make memory and IO block read cost the same.

New Table Scan Cost:
= (6 * 1) + (1 * 239)
= 245

Are plans exhaustively evaluated?

- Short cuts are taken to not spend too much time in planning:
  - Some parts of queries may be transformed to limit plans evaluated
  - The optimizer will by default limit the search depth of bad plans:

```
optimizer_search_depth=64
optimizer_prune_level=1
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. **Hints and Switches**
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

How often is the query optimizer wrong?

- Yes it happens
- Similar to GPS; you may not have traffic data available for all streets
- The model may be incomplete or imperfect
- There exist method(s) to overwrite it

Hints and Switches

- Typically a better level of override to modifying cost constants
- Come in three varieties:
  - Old Style Hints
  - New Comment-Style Hints
  - Switches

Old Style Hints

- Have SQL and Hint intermingled
- Cause errors when indexes don't exist

```
SELECT * FROM Country FORCE INDEX (p) WHERE population >
5000000;
SELECT * FROM Country IGNORE INDEX (p) WHERE population >
5000000;
SELECT * FROM Country USE INDEX (p) WHERE population > 5000000;
SELECT STRAIGHT_JOIN ..;
SELECT * FROM Country STRAIGHT_JOIN ..;
```

New Comment-Style Hints

- Can be added by a system that doesn't understand SQL
- Clearer defined semantics as a hint not a directive
- Fine granularity

```
SELECT
/*+ NO_RANGE_OPTIMIZATION (Country) */
* FROM Country
WHERE Population > 1000000000 AND Continent='Asia';
```

Switches

- As new optimizations are added, some cause regressions
- Allow the specific optimization to be disabled (SESSION or GLOBAL)

```
SELECT @@optimizer_switch;
```

```
index_merge=on,index_merge_union=on,index_merge_sort_union=on,
index_merge_intersection=on,engine_condition_pushdown=on,
index_condition_pushdown=on,mrr=on,mrr_cost_based=on,
block_nested_loop=on,batched_key_access=off,materialization=on,
semijoin=on,loosescan=on,firstmatch=on,duplicateweedout=on,
subquery_materialization_cost_based=on,use_index_extensions=on,
condition_fanout_filter=on,derived_merge=on
```

How to consider hints and switches

- They provide immediate pain relief to production problems at the cost of maintenance
- They add technical debt to your applications

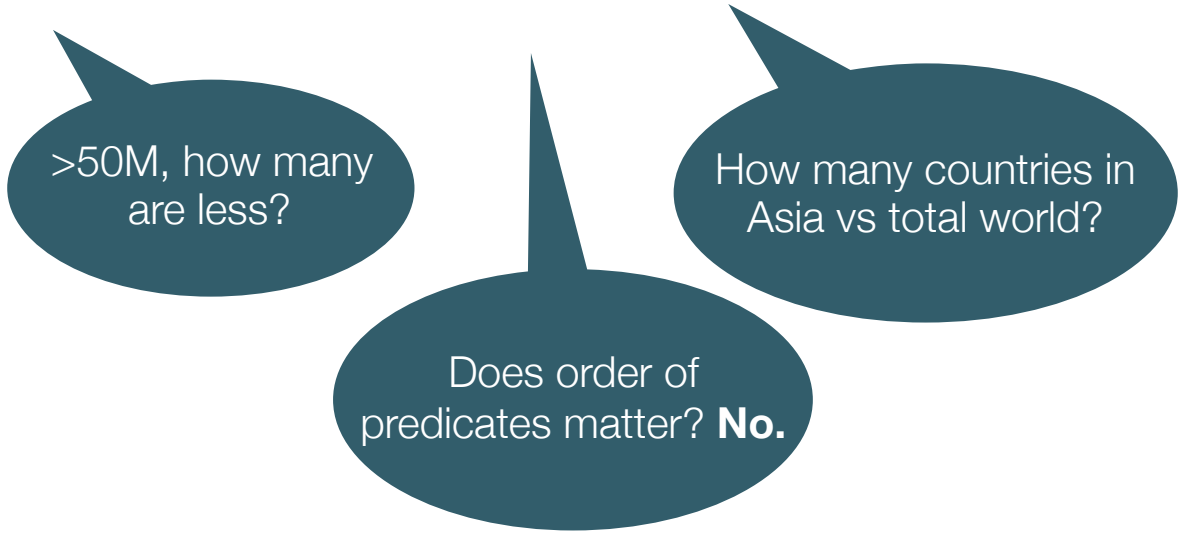
Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. **Comparing Plans**
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Our simple query with *n* candidate indexes

- Indexes exist on `p(population)` and `c(continent)`:

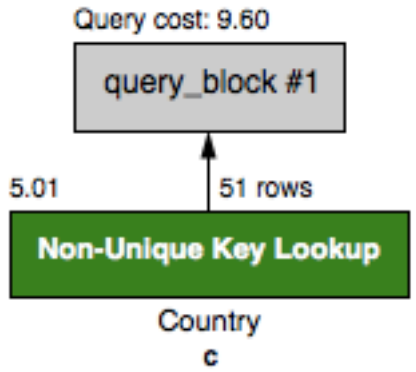
```
SELECT * FROM Country
WHERE population > 50000000 AND continent='Asia';
```



Role of the Optimizer

- Given these many choices, which is the best choice?
- A good GPS navigator finds the fastest route!
- We can expect a good query optimizer to do similar

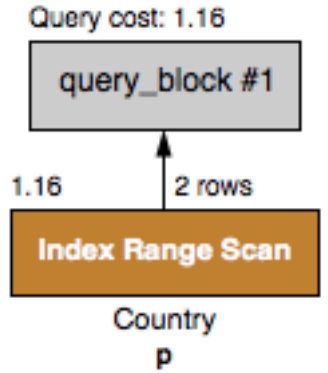
```
ALTER TABLE Country ADD INDEX c (continent);
EXPLAIN FORMAT=JSON # 50M
SELECT * FROM Country WHERE population > 50000000 AND continent='Asia';
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "9.60"
    },
    "table": {
      "table_name": "Country",
      "access_type": "ref",
      "possible_keys": [
        "p",
        "c"
      ],
      "key": "c",
      "used_key_parts": [
        "Continent"
      ],
      "key_length": "1",
      "ref": [
        "const"
      ],
..
      "attached_condition": "(`world`.`country`.`Population` > 50000000)"
..
```



Continent is determined to be lower cost.

```
EXPLAIN FORMAT=JSON # 500M
SELECT * FROM Country WHERE continent='Asia' and population > 500000000;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1.16"
    },
    "table": {
      "table_name": "Country",
      "access_type": "range",
      "possible_keys": [
        "p",
        "c"
      ],
      "key": "p",
      "used_key_parts": [
        "Population"
      ],
      "key_length": "4",
..
      "attached_condition": "(`world`.`country`.`Continent` = 'Asia')"
..
```

Change the predicate, the query plan changes.



Query Plan Evaluation

- Evaluated for each query, and thus each set of predicates
- Currently not cached*
- For prepared statements, permanent transformations are cached

* Cardinality statistics are cached. Don't get confused.

Cost Estimates

	p>5M c='Asia'	p>50M, c='Asia'	p>500M, c='Asia'
p	48.86	11.06	1.16
c	9.60	9.60	9.60
ALL	25.40	25.40	25.40

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. **Composite Indexes**
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

The role of composite indexes

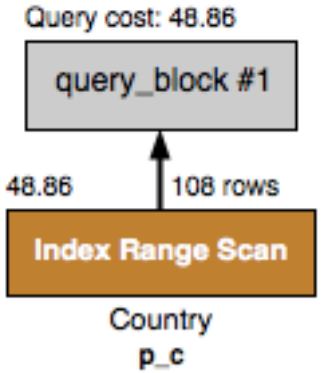
- Useful when two or more predicates combined improves filtering effect. i.e.

  Not all countries with a population > 5M are in Asia

Composite Indexes

- p_c (population, continent)
- c_p (continent, population)

```
ALTER TABLE Country ADD INDEX p_c (Population, Continent);
EXPLAIN FORMAT=JSON
SELECT * FROM Country FORCE INDEX (p_c) WHERE continent='Asia' and population >
5000000;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "48.86"
    },
    "table": {
      "table_name": "Country",
      "access_type": "range",
      "possible_keys": [
        "p_c"
      ],
      "key": "p_c",
      "used_key_parts": [
        "Population"
      ],
      "key_length": "4",
 ..
```



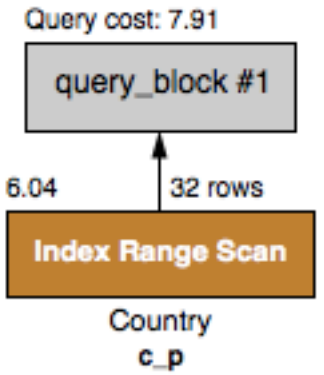
Only part of the key is used!

Rule of Thumb

- Index on (const, range) instead of (range, const)
- Applies to all databases

```
ALTER TABLE Country ADD INDEX c_p (Continent, Population);
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE continent='Asia' and population > 5000000;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "7.91"
    },
    "table": {
      "table_name": "Country",
      "access_type": "range",
      "possible_keys": [
        "p",
        "c",
        "p_c",
        "c_p"
      ],
      "key": "c_p",
      "used_key_parts": [
        "Continent",
        "Population"
      ],
      "key_length": “5”,
..
```

All of the key is used



Composite Left-most Rule

- An index on (Continent, Population) can also be used as an index on (Continent)
- It can not be used as an index on (Population)

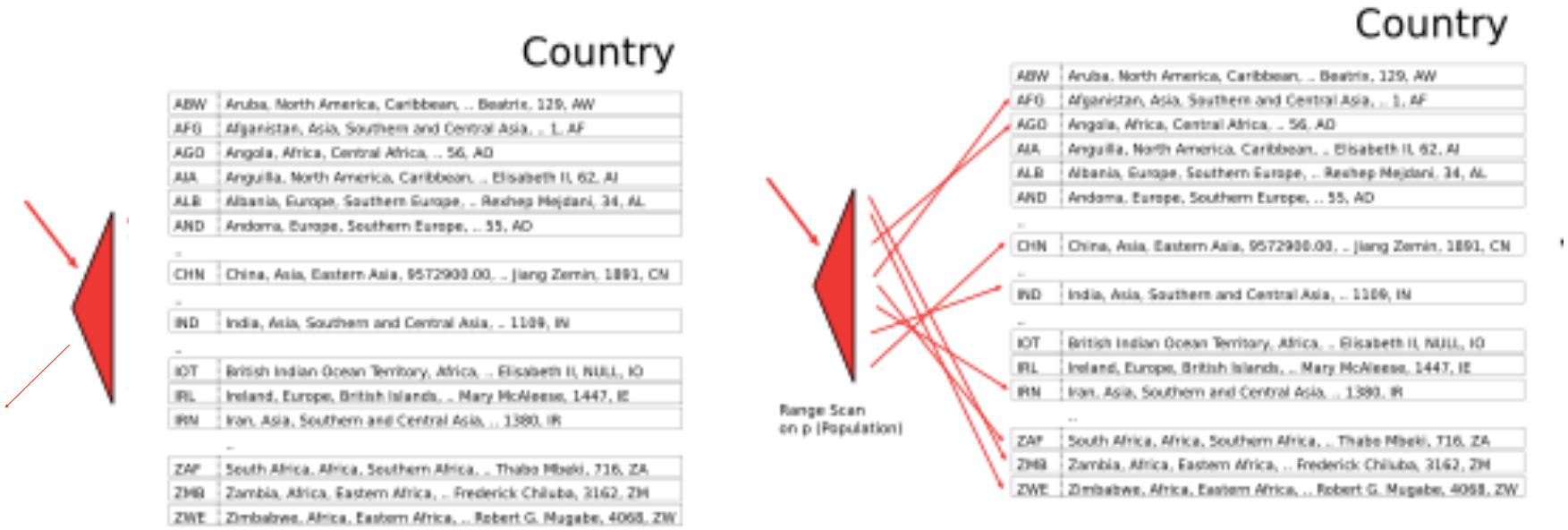
```
EXPLAIN FORMAT=JSON
SELECT * FROM Country FORCE INDEX (c_p) WHERE population >
500000000;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "83.90"
    },
    "table": {
      "table_name": "Country",
      "access_type": "ALL",
      "rows_examined_per_scan": 239,
      "rows_produced_per_join": 79,
      "filtered": "33.33",
..
      "attached_condition": "(`world`.`country`.`Population` >
500000000)"
..
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. **Covering Indexes**
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

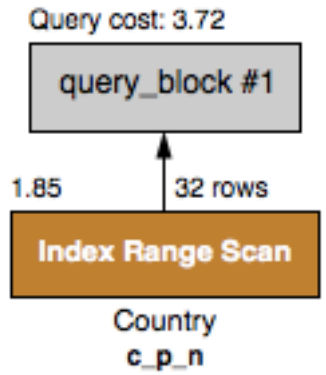
Covering Indexes

- A special kind of composite index
- All information returned just by accessing the index



```
ALTER TABLE Country ADD INDEX c_p_n (Continent,Population,Name);
EXPLAIN FORMAT=JSON
SELECT Name FROM Country WHERE continent='Asia' and population > 5000000;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "3.72"
    },
    "table": {
      "table_name": "Country",
      "access_type": "range",
      "possible_keys": [
..
        "c_p_n"
      ],
      "key": "c_p_n",
      "used_key_parts": [
        "Continent",
        "Population"
      ],
      "key_length": "5",
..
      "filtered": "100.00",
      "using_index": true,
    ..
```

Cost is reduced by 53%



Using index means "covering index"

Use cases

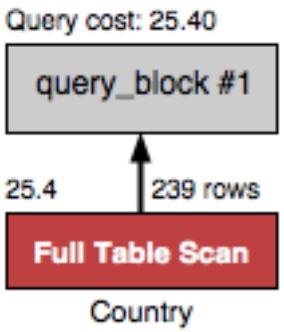
- Can be used as in this example
- Also beneficial in join conditions (join through covering index on intermediate table)
- Useful in aggregate queries

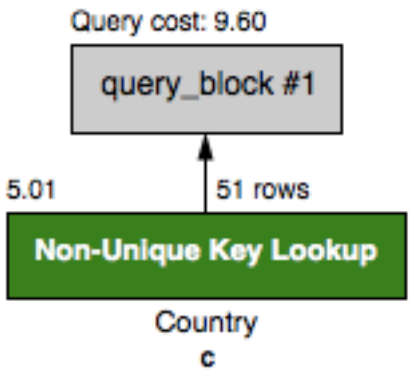
Agenda

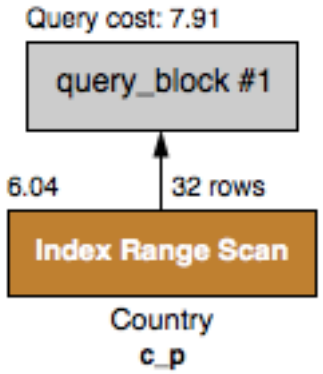
1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. **Visual Explain**
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

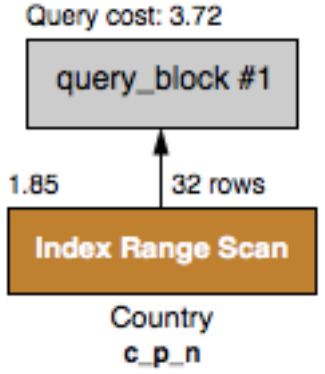
Visual Explain

- For complex queries, it is useful to see visual representation
- Visualizations in this deck are produced by MySQL Workbench.









Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. **Transient Plans**
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

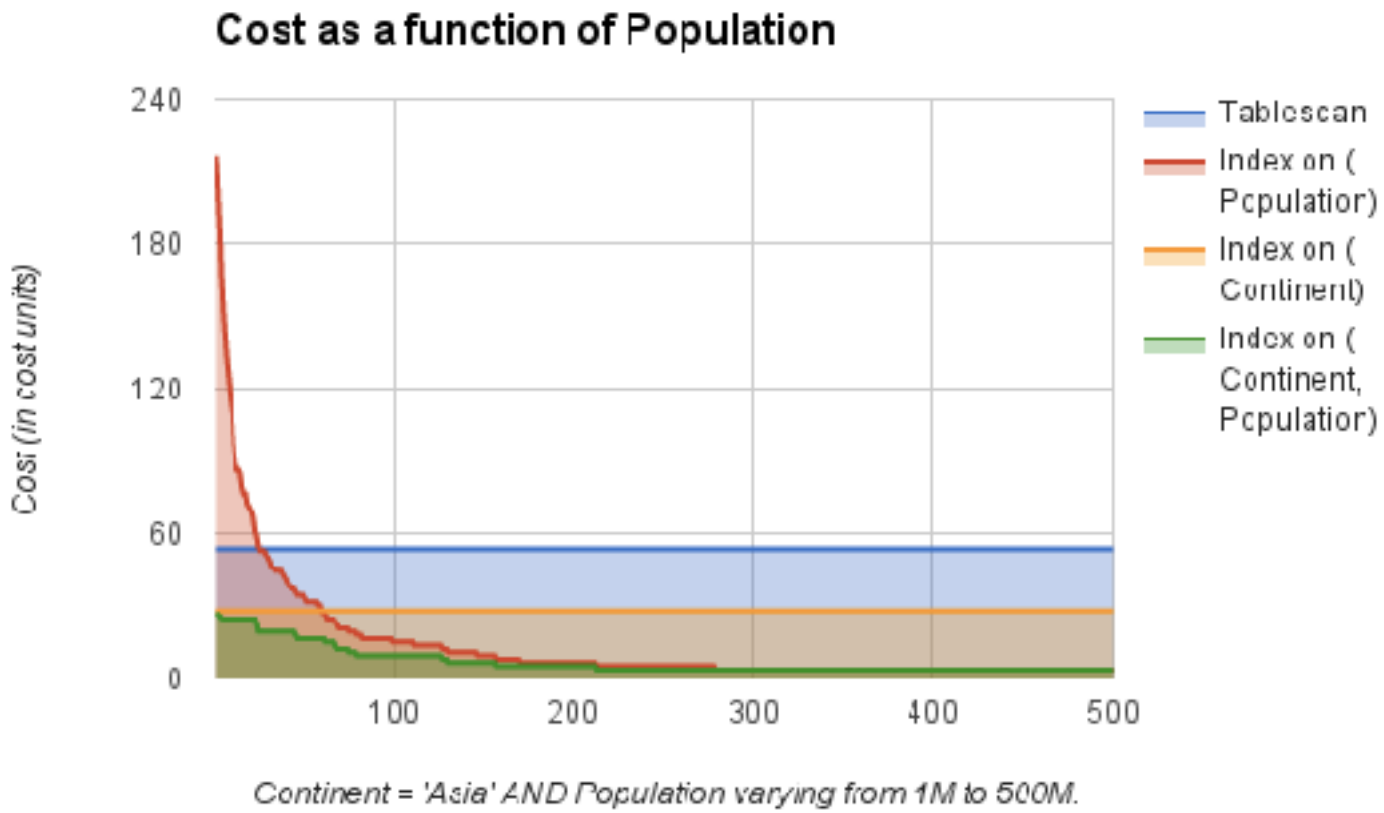
A quick recap:

- So far we've talked about 4 candidate indexes:
  - p (population)
  - c (continent)
  - p_c (population, continent)
  - c_p (continent, population)
- We've always used c='Asia' and p > 5M

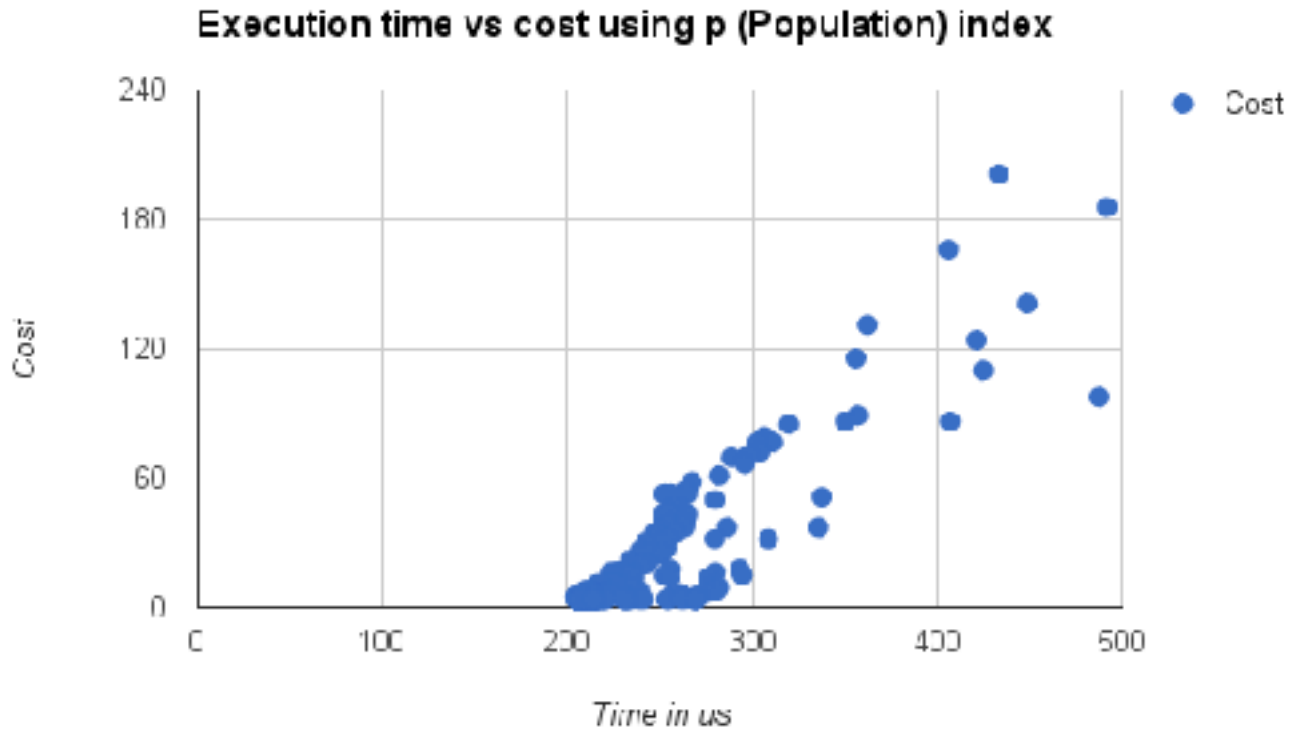
Cost Estimates

	p>5M c='Asia'	p>5M c='Antarctica'	p>50M, c='Asia'	p>50M c='Antarctica'	p>500M, c='Asia'	p>500M c='Antarctica'
p	48.86	48.86	11.06	11.06	1.16	1.16
c	9.60	1.75	9.60	1.75	9.60	1.75
c_p	7.91	0.71	5.21	0.71	1.16	0.71
p_c	48.86	48.86	11.06	11.06	1.16	1.16
ALL	25.40	25.40	25.40	25.40	25.40	25.40

Cost Estimates



Actual Execution Time



Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. **Subqueries**
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Subquery (Scalar)

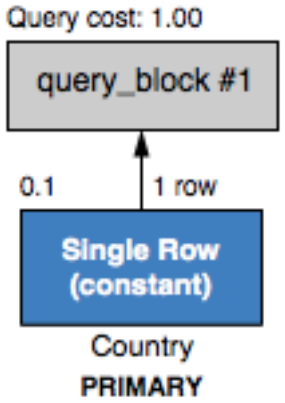
- Can optimize away the inner part first and then cache it.
- This avoids re-executing the inner part for-each-row

```
SELECT * FROM Country WHERE
Code = (SELECT CountryCode FROM City WHERE
name='Toronto');
```

```
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE Code = (SELECT CountryCode FROM City WHERE name='Toronto');
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1.00"
    },
    "table": {
      "table_name": "Country",
      "access_type": "const",
..
      "key": "PRIMARY",
..
    },
    "optimized_away_subqueries": [
      {
        "dependent": false,
        "cacheable": true,
        "query_block": {
          "select_id": 2,
          "cost_info": {
            "query_cost": "425.05"
          },
          "table": {
            "table_name": "City",
            "access_type": "ALL",
..
```

First query + its cost

Second query + its cost

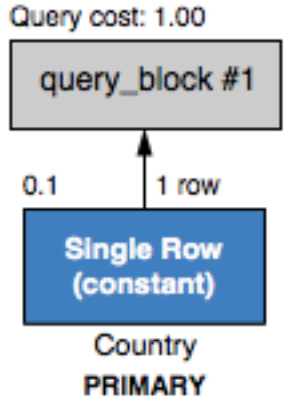


(misleading visualization)

```
ALTER TABLE city ADD INDEX n (name);
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE Code = (SELECT CountryCode FROM City WHERE name='Toronto');
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1.00"
    },
    "table": {
      "table_name": "Country",
      "access_type": "const",
..
      "key": "PRIMARY",
..
    },
    "optimized_away_subqueries": [
      {
        "dependent": false,
        "cacheable": true,
        "query_block": {
          "select_id": 2,
          "cost_info": {
            "query_cost": "0.35"
          },
          "table": {
            "table_name": "City",
            "access_type": "ref",
            "possible_keys": [
              "n"
            ],
            "key": "n",
..
```

First query + its cost

Second query + its cost



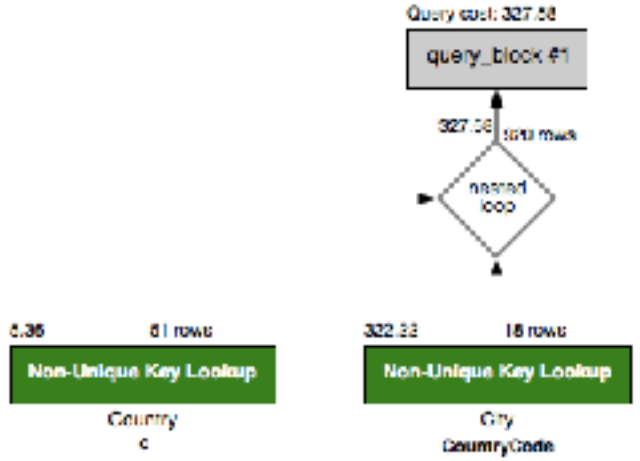
(misleading visualization)

Subquery (IN list)

- When the result inner subquery returns unique results it can safely be transformed to an inner join:

```
EXPLAIN FORMAT=JSON SELECT * FROM City WHERE CountryCode IN
(SELECT Code FROM Country WHERE Continent = 'Asia');


show warnings;
/* select#1 */ select `world`.`city`.`ID` AS `ID`,`world`.`city`.`Name` AS
`Name`,`world`.`city`.`CountryCode` AS `CountryCode`,`world`.`city`.`District`
AS `District`,`world`.`city`.`Population` AS `Population` from
`world`.`country` join `world`.`city` where ((`world`.`city`.`CountryCode` =
`world`.`country`.`Code`) and (`world`.`country`.`Continent` = 'Asia'))
1 row in set (0.00 sec)
```

```
EXPLAIN FORMAT=JSON
SELECT * FROM City WHERE CountryCode IN
(SELECT Code FROM Country WHERE Continent = 'Asia');
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "327.58"
    },
    "nested_loop": [
      {
        "table": {
          "table_name": "Country",
          "access_type": "ref",
..
          "key": "c",
..
          "using_index": true,
..
          "used_columns": [
            "Code",
            "Continent"
 ..
```

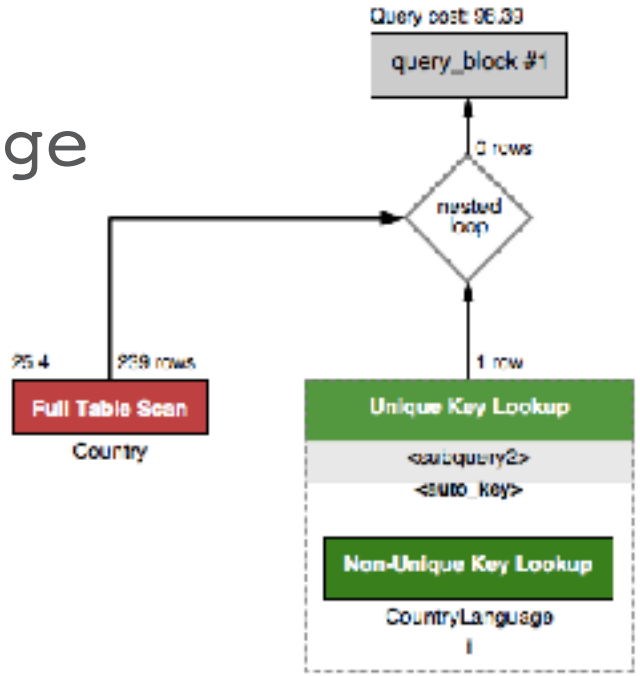



```
      {
        "table": {
          "table_name": "City",
          "access_type": "ref",
          "possible_keys": [
            "CountryCode"
          ],
          "key": "CountryCode",
..
          "ref": [
            "world.Country.Code"
          ],
..
```

Subquery (cont.)

- When non-unique the optimizer needs to pick a semi-join strategy
- Multiple options: `FirstMatch, MaterializeLookup, DuplicatesWeedout`

```
SELECT * FROM Country WHERE Code IN

(SELECT CountryCode FROM CountryLanguage
WHERE isOfficial=1);
```



```
ALTER TABLE CountryLanguage ADD INDEX i (isOfficial);
EXPLAIN FORMAT=JSON SELECT * FROM Country WHERE Code IN
(SELECT CountryCode FROM CountryLanguage WHERE isOfficial=1);
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "98.39"
    },
    "nested_loop": [
      {
        "table": {
          "table_name": "Country",
          "access_type": "ALL",
          "possible_keys": [
            "PRIMARY"
          ],
..
          "filtered": "100.00",
..
```

```
"table": {
  "table_name": "<subquery2>",
  "access_type": "eq_ref",
  "key": "<auto_key>",
  "key_length": "3",
  "ref": [
    "world.Country.Code"
  ],
  "rows_examined_per_scan": 1,
  "materialized_from_subquery": {
    "using_temporary_table": true,
    "query_block": {
      "table": {
        "table_name": "CountryLanguage",
        "access_type": "ref",
..
        "key": "i",
..
        "using_index": true,
..
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. **CTEs and Views**
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

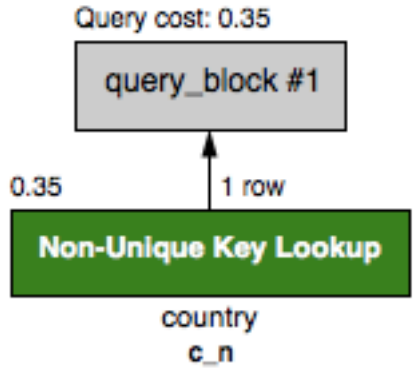
Views

- A way of saving a SELECT statement as a table
- Allows for simplified queries
- Processed using one of two methods internally:
  - **Merge** - transform the view to be combined with the query.
  - **Materialize** - save the contents of the view in a temporary table, then begin querying

```
ALTER TABLE country ADD INDEX c_n (continent, name);
CREATE VIEW vCountry_Asia AS SELECT * FROM Country WHERE Continent='Asia';
EXPLAIN FORMAT=JSON
SELECT * FROM vCountry_Asia WHERE Name='China';
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "0.35"
    },
    "table": {
      "table_name": "country",
      "access_type": "ref",
      "possible_keys": [
..
        "c_n"
      ],
      "key": "c_n",
      "used_key_parts": [
        "Continent",
        "Name"
      ],
      "key_length": "53",
      "ref": [
        "const",
        "const"
      ],
..
```

This is the base table

Predicates from the view definition and query combined



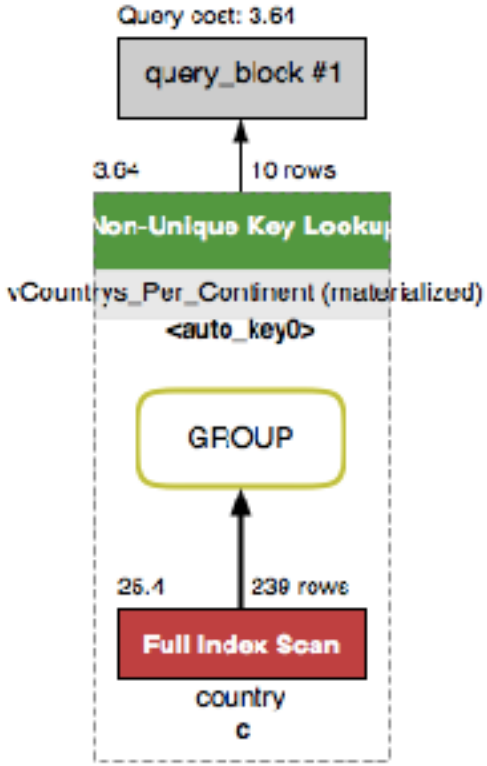
```
SHOW WARNINGS;
/* select#1 */ select
`world`.`Country`.`Code` AS `Code`,
`world`.`Country`.`Name` AS `Name`,
`world`.`Country`.`Continent` AS `Continent`,
`world`.`Country`.`Region` AS `Region`,
`world`.`Country`.`SurfaceArea` AS `SurfaceArea`,
`world`.`Country`.`IndepYear` AS `IndepYear`,
`world`.`Country`.`Population` AS `Population`,
`world`.`Country`.`LifeExpectancy` AS `LifeExpectancy`,
`world`.`Country`.`GNP` AS `GNP`,
`world`.`Country`.`GNPOld` AS `GNPOld`,
`world`.`Country`.`LocalName` AS `LocalName`,
`world`.`Country`.`GovernmentForm` AS `GovernmentForm`,
`world`.`Country`.`HeadOfState` AS `HeadOfState`,
`world`.`Country`.`Capital` AS `Capital`,
`world`.`Country`.`Code2` AS `Code2`
from `world`.`Country`
where
((`world`.`Country`.`Continent` = 'Asia')
and (`world`.`Country`.`Name` = 'China'))
```

```
CREATE VIEW vCountrys_Per_Continent AS
SELECT Continent, COUNT(*) as Count FROM Country
GROUP BY Continent;
EXPLAIN FORMAT=JSON
SELECT * FROM vCountrys_Per_Continent WHERE Continent='Asia';
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "3.64"
    },
    "table": {
      "table_name": "vCountrys_Per_Continent",
      "access_type": "ref",
      "possible_keys": [
        "<auto_key0>"
      ],
      "key": "<auto_key0>",
      "used_key_parts": [
        "Continent"
      ],
      "key_length": "1",
      "ref": [
        "const"
      ],
..
      "used_columns": [
        "Continent",
        "Count"
      ],
..
```

This is only the cost of accessing the materialized table

This is the view name

This step happens first.




```
..
"materialized_from_subquery": {
  "using_temporary_table": true,
  "dependent": false,
  "cacheable": true,
  "query_block": {
    "select_id": 2,
    "cost_info": {
      "query_cost": "25.40"
    },
```

```
SHOW WARNINGS;
/* select#1 */ select
`vCountrys_Per_Continent`.`Continent` AS `Continent`,
`vCountrys_Per_Continent`.`Count` AS `Count`
from `world`.`vCountrys_Per_Continent`
where (`vCountrys_Per_Continent`.`Continent` = 'Asia')
```

WITH (CTE)

- A view for query-only duration
- Same optimizations available as views:
  - **Merge** - transform the CTE to be combined with the query.
  - **Materialize** - save the contents of the CTE in a temporary table, then begin querying

Identical Queries - CTE and VIEW

```
WITH vCountry_Asia AS (SELECT * FROM Country WHERE
Continent='Asia')
SELECT * FROM vCountry_Asia WHERE Name='China';
```

```
CREATE VIEW vCountry_Asia AS SELECT * FROM Country WHERE
Continent='Asia';
SELECT * FROM vCountry_Asia WHERE Name='China';
```

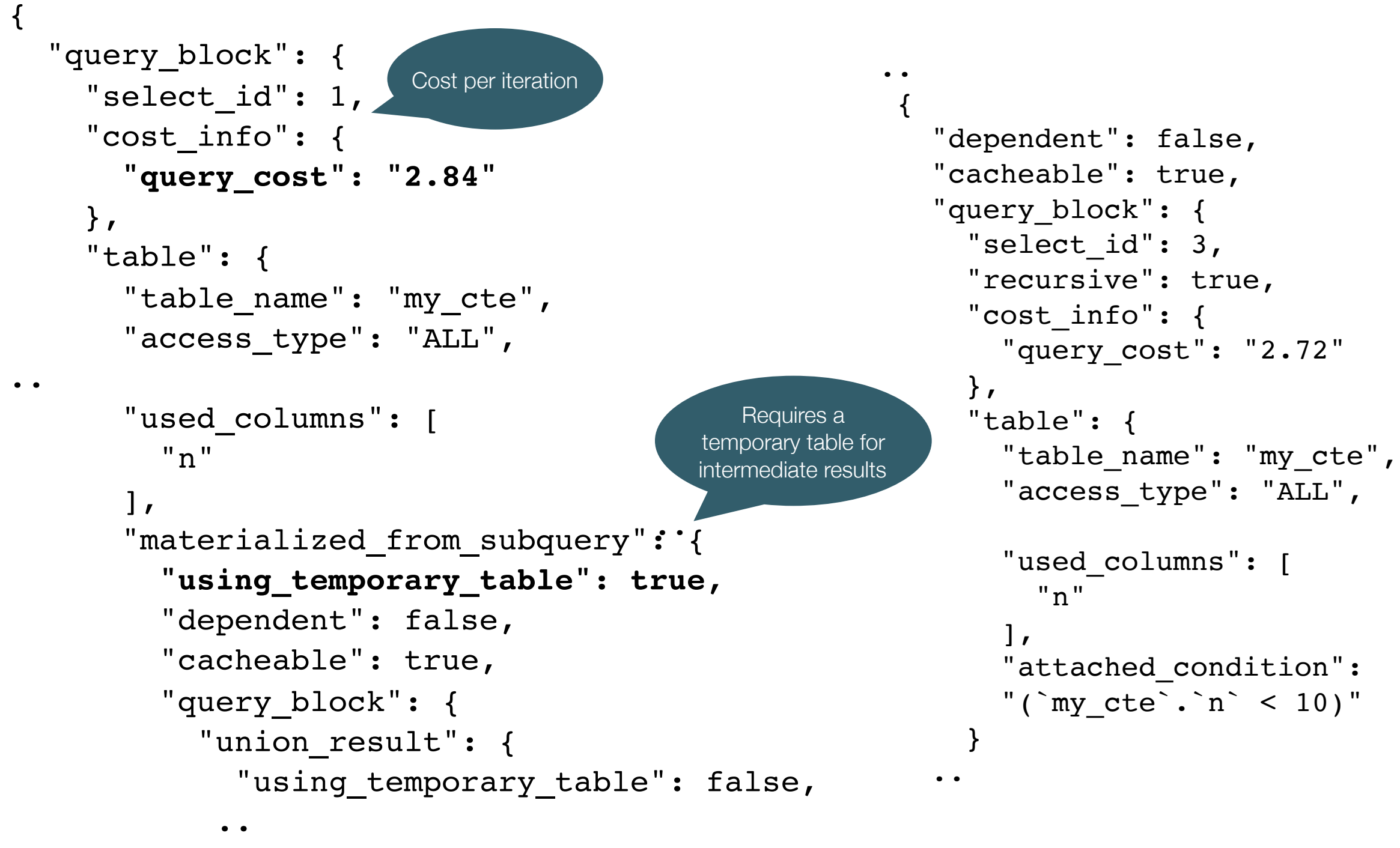
CTEs are new!

- May provide performance enhancements over legacy code using temporary tables - which never merge.
- Derived tables may need to materialize more than once. A CTE does not! i.e.

```
SELECT * FROM my_table, (SELECT ... ) as t1 ...

UNION ALL

SELECT * FROM my_table, (SELECT ... ) as t1 ...
```

WITH RECURSIVE - new!

```
WITH RECURSIVE my_cte AS (
 SELECT 1 AS n
 UNION ALL
 SELECT 1+n FROM my_cte WHERE n<10
)
SELECT * FROM my_cte;
+------+
| n    |
+------+
|    1 |
|    2 |
..
|    9 |
|   10 |
+------+
10 rows in set (0.01 sec)
```


{
"query_block": {
"select_id": 1,
"cost_info": {
"query_cost": "2.84"
},
"table": {
"table_name": "my_cte",
"access_type": "ALL",
..
"used_columns": [
"n"
],
"materialized_from_subquery": {
"using_temporary_table": true,
"dependent": false,
"cacheable": true,
"query_block": {
"union_result": {
"using_temporary_table": false,
..
Cost per iteration
Requires a temporary table for intermediate results
..
{
"dependent": false,
"cacheable": true,
"query_block": {
"select_id": 3,
"recursive": true,
"cost_info": {
"query_cost": "2.72"
},
"table": {
"table_name": "my_cte",
"access_type": "ALL",
"used_columns": [
"n"
],
"attached_condition":
"(`my_cte`.`n` < 10)"
}
..

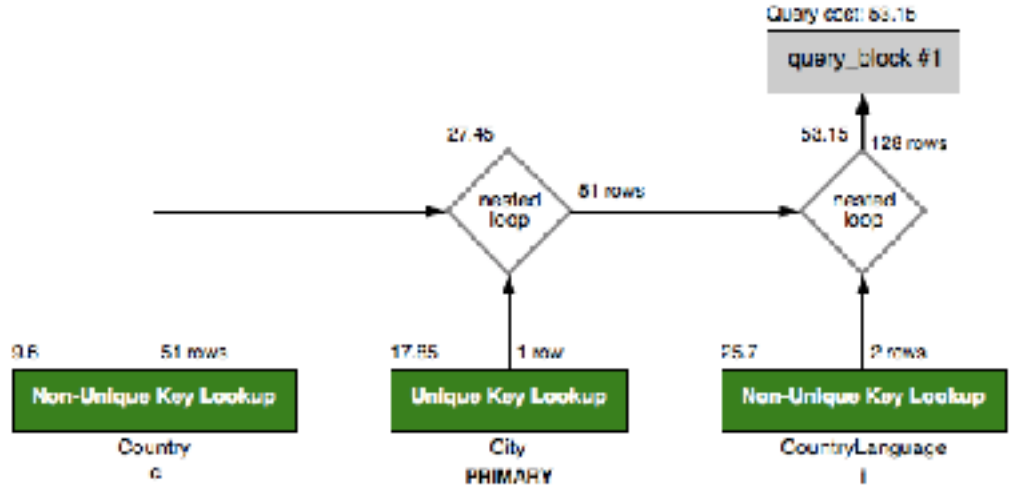
Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. **Joins**
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

```
SELECT
  Country.Name as Country, City.Name as Capital,
Language
FROM
  City
  INNER JOIN Country ON Country.Capital=City.id
  INNER JOIN CountryLanguage ON
CountryLanguage.CountryCode=Country.code
WHERE
  Country.Continent='Asia' and
CountryLanguage.IsOfficial='T';
```

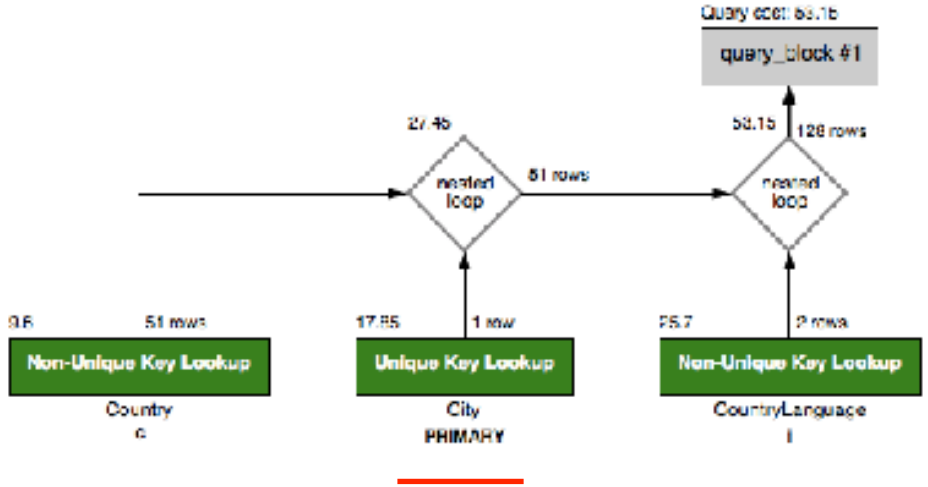
Join Strategy (Nested Loop Join)

1. Pick Driving Table (Country)
2. For each row in Country step through to City table
3. For each row in City table step through to CountryLanguage table
4. Repeat



Join efficiency

- Important to eliminate work before accessing other tables (WHERE clause should have lots of predicates that filter driving table)
- Indexes are required on the columns that connect between driving table, and subsequent tables:



ON Country.Capital=**City.id**

INNER JOIN vs LEFT JOIN

- LEFT JOIN semantically says “right row is optional”.
  - Forces JOIN order to be left side first.
  - Reduces possible ways to join tables

Join Order Hints

- One of the most frequent types of hints to apply
- New join order hints in 8.0:
  - `JOIN_FIXED_ORDER`
  - `JOIN_ORDER`
  - `JOIN_PREFIX`
  - `JOIN_SUFFIX`

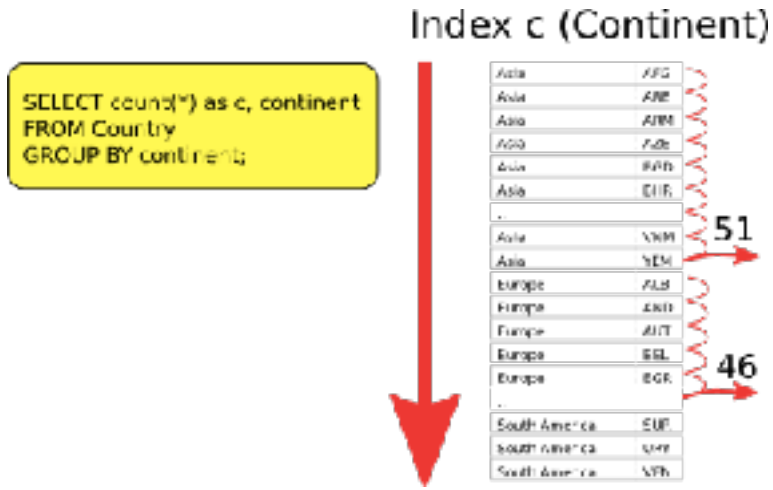
Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. **Aggregation**
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Group By - Loose Index Scan

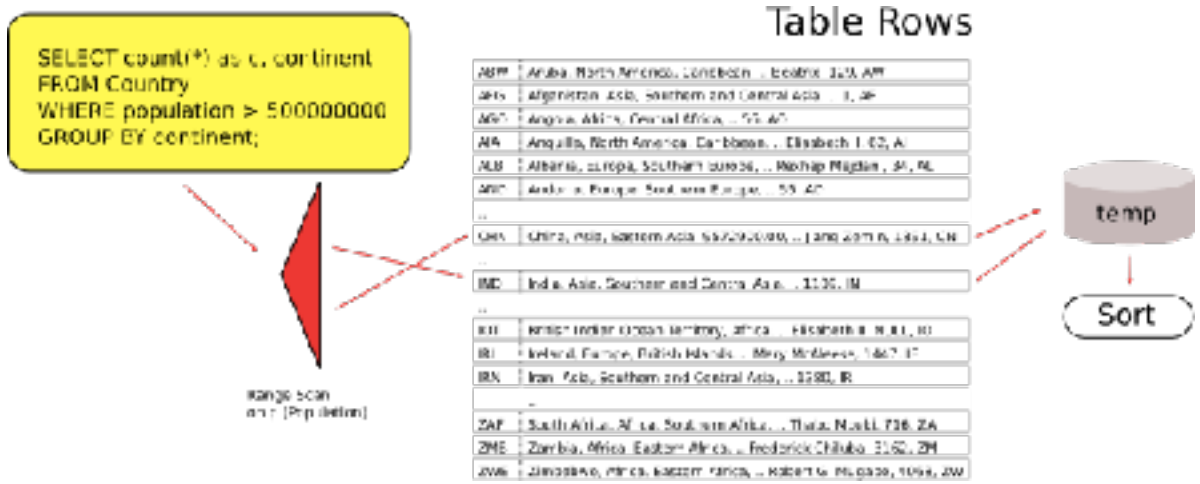
- Scan the index from start to finish without buffering. Results are pipelined to client:

```
SELECT count(*) as c, continent FROM Country
GROUP BY continent;
```



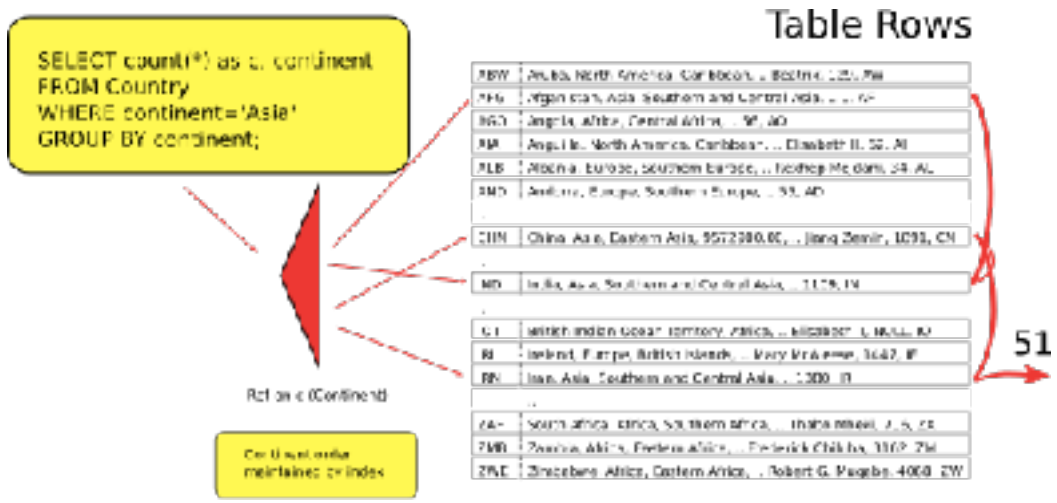
Group By - Index Filtering Rows

- Use the index to eliminate as much work as possible
- Store rows in intermediate temporary file and then sort



Group By - Index Filtering + Guaranteed Order

- Use the index to eliminate as much work as possible
- The index also maintains order



UNION

- Requires an intermediate temporary table to weed out duplicate rows
- The optimizer does not really have any optimizations for UNION (such as a merge with views)

```
EXPLAIN FORMAT=JSON
SELECT * FROM City WHERE CountryCode = 'CAN'
UNION
SELECT * FROM City WHERE CountryCode = 'USA'
{
    "union_result": {
      "using_temporary_table": true,
      "table_name": "<union1,2>",
      "access_type": "ALL",
      "query_specifications": [
        {
          "dependent": false,
          "cacheable": true,
          "query_block": {
            "select_id": 1,
            "cost_info": {
              "query_cost": "17.15"
            },
            "table": {
              "table_name": "City",
              "access_type": "ref",
..
              "key": "CountryCode",
..
```

Temporary table to de-duplicate

```
{
  "dependent": false,
  "cacheable": true,
  "query_block": {
    "select_id": 2,
    "cost_info": {
      "query_cost": "46.15"
    },
    "table": {
      "table_name": "City",
      "access_type": "ref",
      "possible_keys": [
        "CountryCode"
      ],
      "key": "CountryCode",
      "used_key_parts": [
        "CountryCode"
      ],
      "key_length": "3",
      "ref": [
        "const"
      ],
```

UNION ALL

- Results may contain duplicate rows
- Does not require an intermediate temporary table in simple use cases. i.e. no result ordering.
- Otherwise similar to UNION

```
EXPLAIN FORMAT=JSON
SELECT * FROM City WHERE CountryCode = 'CAN'
UNION ALL
SELECT * FROM City WHERE CountryCode = 'USA'
{
  "query_block": {
    "union_result": {
      "using_temporary_table": false,
      "query_specifications": [
        {
          "dependent": false,
          "cacheable": true,
          "query_block": {
            "select_id": 1,
            "cost_info": {
              "query_cost": "17.15"
            },
            "table": {
              "table_name": "City",
              "access_type": "ref",
..
              "key": "CountryCode",
..
```

No temporary table

```
{
  "dependent": false,
  "cacheable": true,
  "query_block": {
    "select_id": 2,
    "cost_info": {
      "query_cost": "46.15"
    },
    "table": {
      "table_name": "City",
      "access_type": "ref",
      "possible_keys": [
        "CountryCode"
      ],
      "key": "CountryCode",
      "used_key_parts": [
        "CountryCode"
      ],
      "key_length": "3",
      "ref": [
        "const"
      ],
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. **Descending Indexes**
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Descending Indexes

- B+tree indexes are ordered
- In 8.0 you can specify the order
- Use cases:
  - Faster to scan in order
  - Can’t change direction in a composite index

```
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE continent='Asia' AND population > 5000000
ORDER BY population DESC;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "7.91"
    },
    "ordering_operation": {
      "using_filesort": false,
      "table": {
        "table_name": "Country",
        "access_type": "range",
..
        "key": "c_p",
..
        "backward_index_scan": true,
..
```

Still uses the index, but about 15% slower

```
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE continent IN ('Asia', 'Oceania') AND population > 5000000
ORDER BY continent ASC, population DESC
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "48.36"
    },
    "ordering_operation": {
      "using_filesort": true,
      "cost_info": {
        "sort_cost": "33.00"
      },
      "table": {
        "table_name": "Country",
        "access_type": "range",
        "key": "c_p",
..
        "rows_examined_per_scan": 33,
        "rows_produced_per_join": 33,
        "filtered": "100.00",
..
```

```
ALTER TABLE Country DROP INDEX c_p, DROP INDEX c_p_n,
ADD INDEX c_p_desc (continent ASC, population DESC);
EXPLAIN FORMAT=JSON
SELECT * FROM Country  WHERE continent IN ('Asia', 'Oceania') AND population > 5000000
ORDER BY continent ASC, population DESC;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "15.36"
    },
    "ordering_operation": {
      "using_filesort": false,
      "table": {
        "table_name": "Country",
        "access_type": "range",
..
        "key": "c_p_desc",
        "used_key_parts": [
          "Continent",
          "Population"
        ],
        "key_length": "5",
..
```

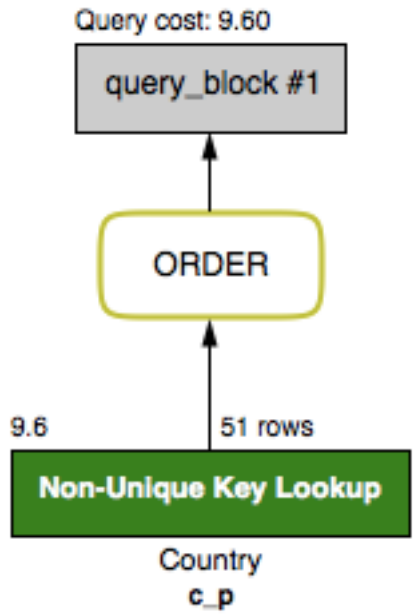
TIP: The optimizer does not consider sort cost in evaluating plans. You may need to FORCE INDEX or DROP similar ascending indexes to use it.

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. **Sorting**
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

How is ORDER BY optimized?

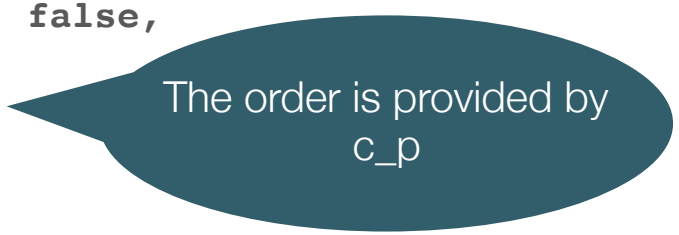
1. Via an Index
2. Top N Buffer (“priority queue”)
3. Using temporary files



Via an Index

- B+tree indexes are ordered
- Some ORDER BY queries do not require sorting at all

```
EXPLAIN FORMAT=JSON
SELECT * FROM Country WHERE continent='Asia' ORDER BY population;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "9.60"
    },
    "ordering_operation": {
      "using_filesort": false,
..
        "key": "c_p",
```



Via a Priority Queue

- Special ORDER BY + small limit optimization
- Keeps top N records in an in memory buffer
- Usage is **NOT** shown in EXPLAIN

```
SELECT * FROM Country IGNORE INDEX (p, p_c)
ORDER BY population LIMIT 10;
```

```
"select#": 1,
"steps": [
  {
    "filesort_information": [
      {
        "direction": "asc",
        "table": "`country` IGNORE INDEX (`p_c`) IGNORE INDEX (`p`)",
        "field": "Population"
      }
    ],
    "filesort_priority_queue_optimization": {
      "limit": 10,
      "chosen": true
    },
    "filesort_execution": [
    ],
    "filesort_summary": {
      "memory_available": 262144,
      "key_size": 4,
      "row_size": 272,
      "max_rows_per_buffer": 11,
      "num_rows_estimate": 587,
      "num_rows_found": 11,
      "num_examined_rows": 239,
      "num_tmp_files": 0,
      "sort_buffer_size": 3080,
      "sort_algorithm": "std::sort",
      "unpacked_addon_fields": "using_priority_queue",
      "sort_mode": "<fixed_sort_key, additional_fields>”
..
```

Using Temporary Files

- Either “Alternative Sort Algorithm” (no blobs present) or “Original Sort Algorithm”

```
SELECT * FROM Country IGNORE INDEX (p, p_c)
ORDER BY population;
```

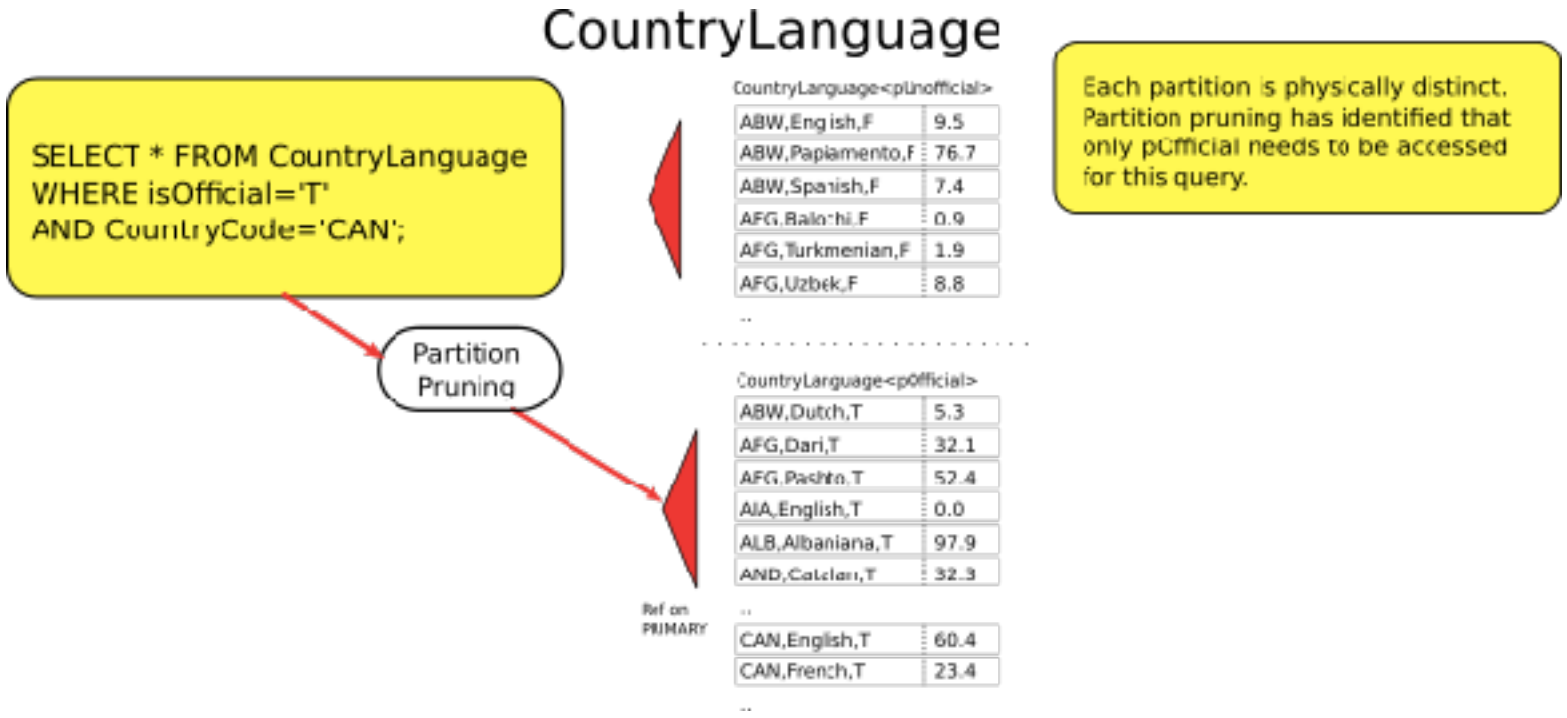
```
"select#": 1,
"steps": [
  {
    "filesort_information": [
      {
        "direction": "asc",
        "table": "`country` IGNORE INDEX (`p_c`) IGNORE INDEX (`p`)",
        "field": "Population"
      }
    ],
    "filesort_priority_queue_optimization": {
      "usable": false,
      "cause": "not applicable (no LIMIT)"
    },
    "filesort_execution": [
    ],
    "filesort_summary": {
      "memory_available": 262144,
      "key_size": 4,
      "row_size": 274,
      "max_rows_per_buffer": 587,
      "num_rows_estimate": 587,
      "num_rows_found": 239,
      "num_examined_rows": 239,
      "num_tmp_files": 0,
      "sort_buffer_size": 165536,
      "sort_algorithm": "std::stable_sort",
      "sort_mode": "<fixed_sort_key, packed_additional_fields>"
..
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. **Partitioning**
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Partitioning

- Split a table physically into smaller tables
- At the user-level make it still appear as one table



Use Cases

- Can be a better fit low cardinality columns than indexing
- Useful for time series data with retention scheme
  - i.e. drop data older than 3 months
- Data where queries always have some locality
  - i.e. store_id, region

Partition Pruning

- Optimizer looks at query and identifies which partitions need to be accessed

```
ALTER TABLE CountryLanguage MODIFY IsOfficial CHAR(1) NOT NULL DEFAULT 'F', DROP
PRIMARY KEY, ADD PRIMARY KEY(CountryCode, Language, IsOfficial);

ALTER TABLE CountryLanguage PARTITION BY LIST COLUMNS (IsOfficial) (
 PARTITION pUnofficial VALUES IN ('F'),
 PARTITION pOfficial VALUES IN ('T')
);
```

```
EXPLAIN FORMAT=JSON
SELECT * FROM CountryLanguage WHERE isOfficial='T' AND
CountryCode='CAN';
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "2.40"
    },
    "table": {
      "table_name": "CountryLanguage",
      "partitions": [
        "pOfficial"
      ],
      "access_type": "ref",
..
      "key": "PRIMARY",
..
```

Explicit Partition Selection

- Also possible to “target” a partition
- Consider this similar to query hints

```
SELECT * FROM CountryLanguage PARTITION (pOfficial)
WHERE CountryCode='CAN';
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. **Query Rewrite**
22. Invisible Indexes
23. Profiling
24. JSON
25. Character Sets

Query Rewrite

- MySQL allows you to change queries before they are executed
- Insert a hint, or remove a join that is not required

```
mysql -u root -p < install_rewriter.sql


INSERT INTO query_rewrite.rewrite_rules(pattern_database, pattern,
replacement) VALUES (
"world",
"SELECT * FROM Country WHERE population > ? AND continent=?",
"SELECT * FROM Country WHERE population > ? AND continent=? LIMIT 1"
);
CALL query_rewrite.flush_rewrite_rules();
```

```
SELECT * FROM Country WHERE population > 5000000 AND
continent='Asia';

SHOW WARNINGS;

*************************** 1. row ***************************

  Level: Note

   Code: 1105

Message: Query 'SELECT * FROM Country WHERE population >
5000000 AND continent='Asia'' rewritten to 'SELECT *
FROM Country WHERE population > 5000000 AND
continent='Asia' LIMIT 1' by a query rewrite plugin

1 row in set (0.00 sec)
```

```
SELECT * FROM query_rewrite.rewrite_rules\G
*************************** 1. row ***************************
                    id: 1
               pattern: SELECT * FROM Country WHERE
population > ? AND continent=?
  pattern_database: world
       replacement: SELECT * FROM Country WHERE
population > ? AND continent=? LIMIT 1
               enabled: YES
               message: NULL
    pattern_digest: 88876bbb502cef6efddcc661cce77deb
normalized_pattern: select `*` from `world`.`country`
where ((`population` > ?) and (`continent` = ?))
1 row in set (0.00 sec)
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. **Invisible Indexes**
23. Profiling
24. JSON
25. Character Sets

Changing Indexes is a *Destructive Operation*

- Removing an index can make some queries much slower
- Adding can cause some existing query plans to change
- Old-style hints will generate errors if indexes are removed

Invisible Indexes, the “Recycle Bin”

- Hide the indexes from the optimizer
- Will no longer be considered as part of query execution plans
- Still kept up to date and are maintained by insert/update/delete statements

Invisible Indexes: Soft Delete

```
ALTER TABLE Country ALTER INDEX c INVISIBLE;
SELECT * FROM information_schema.statistics WHERE is_visible='NO';
*************************** 1. row ***************************
TABLE_CATALOG: def
 TABLE_SCHEMA: world
   TABLE_NAME: Country
   NON_UNIQUE: 1
 INDEX_SCHEMA: world
   INDEX_NAME: c
 SEQ_IN_INDEX: 1
  COLUMN_NAME: Continent
    COLLATION: A
  CARDINALITY: 7
     SUB_PART: NULL
       PACKED: NULL
     NULLABLE:
   INDEX_TYPE: BTREE
      COMMENT: disabled
INDEX_COMMENT:
   IS_VISIBLE: NO
```

Invisible Indexes: Staged Rollout

```
ALTER TABLE Country ADD INDEX c (Continent)
INVISIBLE;


# after some time

ALTER TABLE Country ALTER INDEX c VISIBLE;
```

Finding Unused Indexes

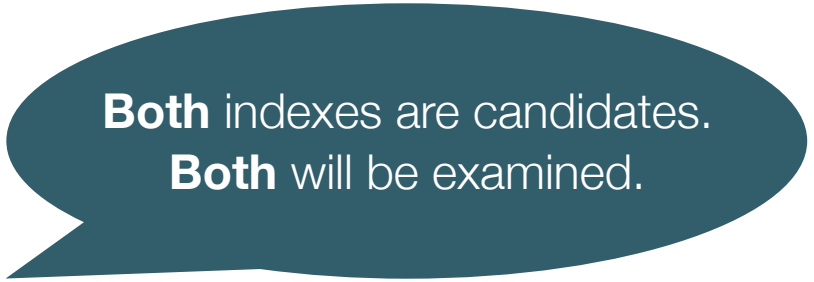
```
SELECT * FROM sys.schema_unused_indexes;
+---------------+-------------+------------+
| object_schema | object_name | index_name |
+---------------+-------------+------------+
| world         | Country     | p          |
| world         | Country     | p_c        |
+---------------+-------------+------------+
2 rows in set (0.01 sec)
```

Do indexes hurt reads or writes?

- They can have some impact on both:
  - On writes, indexes need to space, and to be maintained
  - On reads, lets use an example…

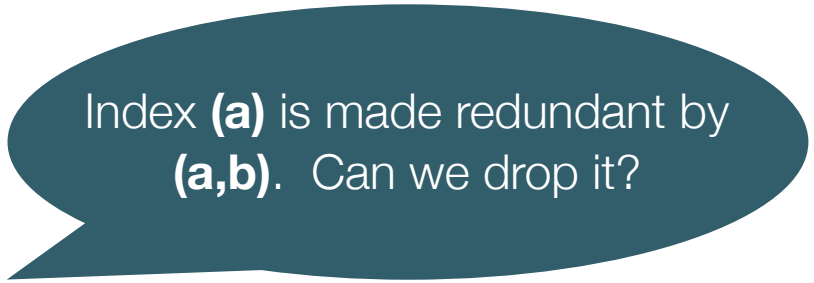
Indexes Hurting Reads

```
CREATE TABLE t1 (
 id INT NOT NULL primary key auto_increment,
 a VARCHAR(255) NOT NULL,
 b VARCHAR(255) NOT NULL,
 c TEXT,
 d TEXT,
INDEX a (a),
INDEX ab (a,b));

# Sample Query
SELECT * FROM t1 WHERE a = 'abc' AND b = 'bcd';
```



A use case for invisible indexes!

```
CREATE TABLE t1 (
 id INT NOT NULL primary key auto_increment,
 a VARCHAR(255) NOT NULL,
 b VARCHAR(255) NOT NULL,
 c TEXT,
 d TEXT,
INDEX a (a),
INDEX ab (a,b));

# Consider:
SELECT count(*) FROM t1 FORCE INDEX (a)
WHERE a='1234' AND id=1234;
```



No, due to clustered Index!

```
FORCE INDEX (a) WHERE a='1234' AND id=1234;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "0.35"
    },
    "table": {
      "table_name": "t1",
      "access_type": "const",
      "possible_keys": [
        "a"
      ],
      "key": "a",
      "used_key_parts": [
        "a",
        "id"
      ],
..
```

```
FORCE INDEX (ab) WHERE a='1234' AND id=1234;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "11.80"
    },
    "table": {
      "table_name": "t1",
      "access_type": "ref",
      "possible_keys": [
        "ab"
      ],
      "key": "ab",
      "used_key_parts": [
        "a"
      ],
..
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. **Profiling**
24. JSON
25. Character Sets

Profiling

- Optimizer only shows estimates from pre-execution view
- Can be useful to know actual time spent
- Support for profiling is only very basic

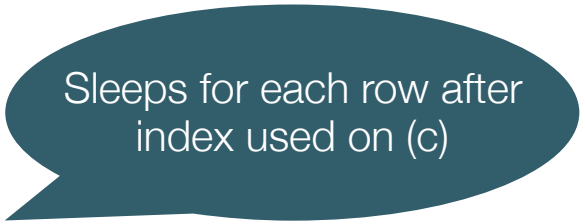
```
wget http://www.tocker.ca/files/ps-show-profiles.sql
mysql -u root -p < ps-show-profiles.sql
```

```
CALL sys.enable_profiling();
CALL sys.show_profiles;
*************************** 1. row ***************************
Event_ID: 22
Duration: 495.02 us
   Query: SELECT * FROM Country WHERE co ... Asia' and population > 5000000
1 row in set (0.00 sec)

CALL sys.show_profile_for_event_id(22);
+----------------------+-----------+
| Status               | Duration  |
+----------------------+-----------+
| starting             | 64.82 us  |
| checking permissions | 4.10 us   |
| Opening tables       | 11.87 us  |
| init                 | 29.74 us  |
| System lock          | 5.63 us   |
| optimizing           | 8.74 us   |
| statistics           | 139.38 us |
| preparing            | 11.94 us  |
| executing            | 348.00 ns |
| Sending data         | 192.59 us |
| end                  | 1.17 us   |
| query end            | 4.60 us   |
| closing tables       | 4.07 us   |
| freeing items        | 13.60 us  |
| cleaning up          | 734.00 ns |
+----------------------+-----------+
15 rows in set (0.00 sec)
```

```
SELECT * FROM Country WHERE Continent='Antarctica' and SLEEP(5);
CALL sys.show_profiles();
CALL sys.show_profile_for_event_id(<event_id>);
```

Status	Duration
starting	103.89 us
checking permissions	4.48 us
Opening tables	17.78 us
init	45.75 us
System lock	8.37 us
optimizing	11.98 us
statistics	144.78 us
preparing	15.78 us
executing	634.00 ns
Sending data	116.15 us
User sleep	**5.00 s**
User sleep	**5.00 s**
User sleep	**5.00 s**
User sleep	**5.00 s**
User sleep	**5.00 s**
end	2.05 us
query end	5.63 us
closing tables	7.30 us
freeing items	20.19 us
cleaning up	1.20 us

20 rows in set (0.01 sec)



```
SELECT region, count(*) as c FROM Country GROUP BY region;
CALL sys.show_profiles();
CALL sys.show_profile_for_event_id(<event_id>);
+----------------------+-----------+
| Status               | Duration  |
+----------------------+-----------+
| starting             | 87.43 us  |
| checking permissions | 4.93 us   |
| Opening tables       | 17.35 us  |
| init                 | 25.81 us  |
| System lock          | 9.04 us   |
| optimizing           | 3.37 us   |
| statistics           | 18.31 us  |
| preparing            | 10.94 us  |
| Creating tmp table   | 35.57 us  |
| Sorting result       | 2.38 us   |
| executing            | 741.00 ns |
| Sending data         | 446.03 us |
| Creating sort index  | 49.45 us  |
| end                  | 1.71 us   |
| query end            | 4.85 us   |
| removing tmp table   | 4.71 us   |
| closing tables       | 6.12 us   |
| freeing items        | 17.17 us  |
| cleaning up          | 1.00 us   |
+----------------------+-----------+
19 rows in set (0.01 sec)
```

```
SELECT * FROM performance_schema.events_statements_history_long
WHERE event_id=<event_id>\G
*************************** 1. row ***************************
            THREAD_ID: 3062
             EVENT_ID: 1566
         END_EVENT_ID: 1585
           EVENT_NAME: statement/sql/select
               SOURCE: init_net_server_extension.cc:80
          TIMER_START: 5888838695662770000
            TIMER_END: 5888838703176830000
           TIMER_WAIT: 751406000
            LOCK_TIME: 132000000
              SQL_TEXT: SELECT region,
count(*) as c FROM Country GROUP BY region
                DIGEST: d3a04b346fe48da4f1f5c2e06628a245
           DIGEST_TEXT: SELECT `region` ,
COUNT ( * ) AS `c` FROM `Country`
GROUP BY `region`
        CURRENT_SCHEMA: world
           OBJECT_TYPE: NULL
         OBJECT_SCHEMA: NULL
           OBJECT_NAME: NULL
  OBJECT_INSTANCE_BEGIN: NULL
           MYSQL_ERRNO: 0
     RETURNED_SQLSTATE: NULL
          MESSAGE_TEXT: NULL
                ERRORS: 0
              WARNINGS: 0
..
..
            ROWS_AFFECTED: 0
                ROWS_SENT: 25
            ROWS_EXAMINED: 289
  CREATED_TMP_DISK_TABLES: 0
       CREATED_TMP_TABLES: 1
         SELECT_FULL_JOIN: 0
   SELECT_FULL_RANGE_JOIN: 0
             SELECT_RANGE: 0
       SELECT_RANGE_CHECK: 0
              SELECT_SCAN: 1
        SORT_MERGE_PASSES: 0
               SORT_RANGE: 0
                SORT_ROWS: 25
                SORT_SCAN: 1
            NO_INDEX_USED: 1
       NO_GOOD_INDEX_USED: 0
         NESTING_EVENT_ID: NULL
       NESTING_EVENT_TYPE: NULL
      NESTING_EVENT_LEVEL: 0
```

For non-aggregate queries rows sent vs. rows examined helps indicate index effectiveness.

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. **JSON**
25. Character Sets

JSON

- Optimizer has native support for JSON with indexes on generated columns used for matching JSON path expressions

```
CREATE TABLE CountryJson (Code char(3) not null primary key, doc JSON NOT NULL);
INSERT INTO CountryJSON SELECT code,
 JSON_OBJECT(
  'Name', Name,
  'Continent', Continent,
..
  'HeadOfState',HeadOfState,
  'Capital', Capital,
  'Code2', Code2
 ) FROM Country;
```

```
EXPLAIN FORMAT=JSON
SELECT * FROM CountryJSON where doc->>"$.Name" = 'Canada';
 {
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "48.80"
    },
    "table": {
      "table_name": "CountryJSON",
      "access_type": "ALL",
      "rows_examined_per_scan": 239,
      "rows_produced_per_join": 239,
      "filtered": "100.00",
      "cost_info": {
        "read_cost": "1.00",
        "eval_cost": "47.80",
        "prefix_cost": "48.80",
        "data_read_per_join": "3K"
      },
..
```

```
ALTER TABLE CountryJSON ADD Name char(52) AS (doc->>"$.Name"),
ADD INDEX n (Name);
EXPLAIN FORMAT=JSON
SELECT * FROM CountryJSON where doc->>"$.Name" = 'Canada';
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1.20"
    },
    "table": {
      "table_name": "CountryJSON",
      "access_type": "ref"
..
      "key": "n",
..
      "key_length": "53",
      "ref": [
        "const"
      ],
..
```

Matches expression from indexed virtual column

Key from virtual column

JSON Comparator

- JSON types compare to MySQL types

```
SELECT CountryJSON.* FROM CountryJSON
INNER JOIN Country ON CountryJSON.doc->>"$.Name" = Country.Name WHERE
Country.Name='Canada';
*********************** 1. row ***********************
Code: CAN
 doc: {"GNP": 598862, "Name": "Canada", "Code2": "CA", "GNPOld": 625626, "Region":
"North America", "Capital": 1822, "Continent": "North America", "IndepYear": 1867,
"LocalName": "Canada", "Population": 31147000, "HeadOfState": "Elisabeth II",
"SurfaceArea": 9970610, "GovernmentForm": "Constitutional Monarchy, Federation",
"LifeExpectancy": 79.4000015258789}
Name: Canada
```

Agenda

1. Introduction
2. Server Architecture
3. B+trees
4. EXPLAIN
5. Optimizer Trace
6. Logical Transformations
7. Cost Based Optimization
8. Hints and Switches
9. Comparing Plans
10. Composite Indexes
11. Covering Indexes
12. Visual Explain
13. Transient Plans
14. Subqueries
15. CTEs and Views
16. Joins
17. Aggregation
18. Descending Indexes
19. Sorting
20. Partitioning
21. Query Rewrite
22. Invisible Indexes
23. Profiling
24. JSON
25. **Character Sets**

Character Sets

- The default character set in MySQL 8.0 is utf8mb4
- Utf8mb4 is variable length (1-4 bytes)
- InnoDB will always store as variable size for both CHAR and VARCHAR
- Some buffers inside MySQL may require the fixed length (4 bytes)

Character Sets (cont.)

- CHAR(n) or VARCHAR(n) refers to n characters - x4 for maximum length
- EXPLAIN will always show the maximum length
- Mysqldump will preserve character set

```
ALTER TABLE City DROP FOREIGN KEY city_ibfk_1;

ALTER TABLE CountryLanguage DROP FOREIGN KEY
countryLanguage_ibfk_1;

ALTER TABLE Country CONVERT TO CHARACTER SET
utf8mb4;
```

```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "0.35"
    },
    "table": {
      "table_name": "Country",
      "access_type": "ref",
      "possible_keys": [
        "n"
      ],
      "key": "n",
      "used_key_parts": [
        "Name"
      ],
      "key_length": "52",
..
      "rows_examined_per_scan": 1,
      "rows_produced_per_join": 1,
      "filtered": "100.00",
      "cost_info": {
        "read_cost": "0.25",
        "eval_cost": "0.10",
        "prefix_cost": "0.35",
        "data_read_per_join": "264"
      },
..
```

Key length as latin1

```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "0.35"
    },
    "table": {
      "table_name": "Country",
      "access_type": "ref",
      "possible_keys": [
        "n"
      ],
      "key": "n",
      "used_key_parts": [
        "Name"
      ],
      "key_length": "208",
..
      "rows_examined_per_scan": 1,
      "rows_produced_per_join": 1,
      "filtered": "100.00",
      "cost_info": {
        "read_cost": "0.25",
        "eval_cost": "0.10",
        "prefix_cost": "0.35",
        "data_read_per_join": "968"
      },
..
```

Key length as utf8

Conclusion

- Thank you for coming!
- This presentation is available as a website: www.unofficialmysqlguide.com


ORACLE®