MariaDB Fest 2020

Optimizer Trace walkthrough

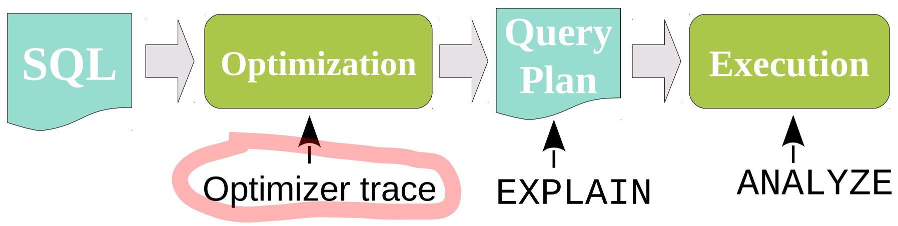
Sergei Petrunia
MariaDB

Optimizer Trace

- Prototyped before MariaDB existed
- Initial release: MySQL 5.6
- Release in MariaDB: 10.4 (June 2019)
  - After that, added more tracing
  - Got experience with using it

Optimizer Trace goals

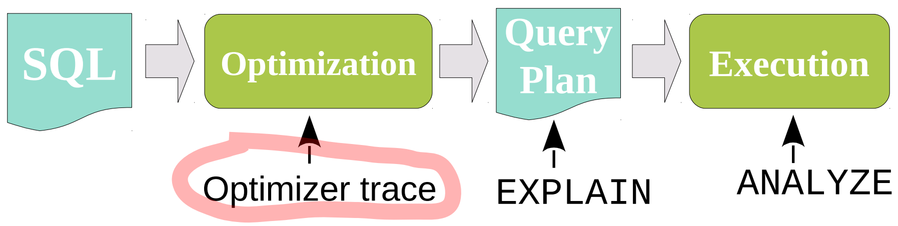
- “Show details about what goes on in the optimizer”



- There is a lot going on there
  - rewrites (e.g. view merging)
  - WHERE analysis, finding ways to read rows (t.key_column < 'abc')
  - Search for query plan
    - *Some* of possible plans are considered
  - Plan refinement

Optimizer Trace goals

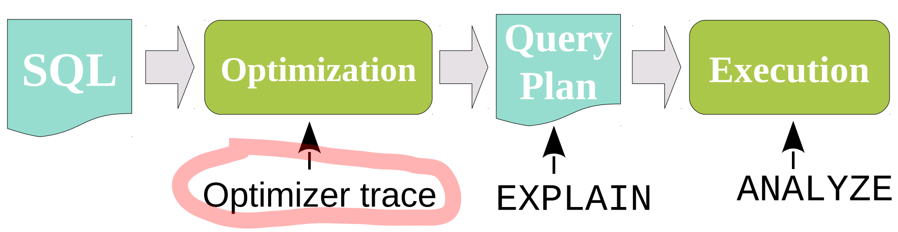
- "Show details about what goes on in the optimizer"



- Answers questions
  - Why query plan X is [not] chosen
    - Can strategy Y be used for index Z?
    - Is restriction on indexed column C sargable?
    - ...
  - Why are query plans different across db instances
    - different db version (upgrades)
    - different statistics or "minor" DDL changes

Optimizer Trace goals (2)

- "Show details about what goes on in the optimizer"



- Answers questions (2)
  - Asking for help or reporting an issue to the development team
    - Uploading the whole dataset is not always possible
    - EXPLAIN, Table definitions, etc – are not always sufficient
    - The trace is text, so one can remove any sensitive info.

Getting the Optimizer Trace

Getting Optimizer Trace

```
MariaDB> set optimizer_trace=1;

MariaDB> <query>;

MariaDB> select * from
      ->   information_schema.optimizer_trace;
```

- Enable trace
- Run the query
  - Its trace is kept in memory, per-session.
- Examine the trace


```
TRACE: {
  "steps": [
    {
      "join_preparation": {
        "select_id": 1,
        "steps": [
          {
            "expanded_query": "select orders.o_orderkey AS o_orderkey,...
          }
        ]
      }
    },
    {
      "join_optimization": {
        "select_id": 1,
        "steps": [
          {
            "condition_processing": {
              "condition": "WHERE",
              "original_condition": "orders.o_orderDATE =
lineitem.l_shipDATE and orders.o_orderDATE = '1995-01-01' and
orders.o_orderkey = lineitem.l_orderkey",
              "steps": [
                {
                  "transformation": "equality_propagation",
                  "resulting_condition": "multiple equal(DATE'1995-01-01',
orders.o_orderDATE, lineitem.l_shipDATE) and multiple
equal(orders.o_orderkey, lineitem.l_orderkey)"
                },
                {
                  "transformation": "constant_propagation",
                  "resulting_condition": "multiple equal(DATE'1995-01-01',
orders.o_orderDATE, lineitem.l_shipDATE) and multiple
equal(orders.o_orderkey, lineitem.l_orderkey)"
                },
                {
                  "transformation": "trivial_condition_removal",
                  "resulting_condition": "multiple equal(DATE'1995-01-01',
orders.o_orderDATE, lineitem.l_shipDATE) and multiple
equal(orders.o_orderkey, lineitem.l_orderkey)"
                }
              ]
            }
          },
          {
            "table_dependencies": [
              {
                "table": "orders",
                "row_may_be_null": false,
                "map_bit": 0,
                "depends_on_map_bits": []
              },
              {
                "table": "lineitem",
                "row_may_be_null": false,
                "map_bit": 1,
                "depends_on_map_bits": []
              }
            ]
          },
```

Optimizer trace contents

- It’s a huge JSON document
- “A log of actions done by the optimizer”
  - I would like to say “all actions”
  - But this isn’t the case.
  - A good subset of all actions.


```
TRACE: {
  "steps": [
    {
      "join_preparation": {
        "select_id": 1,
        "steps": [
          {
            "expanded_query": "select orders.o_orderkey AS o_orderkey,...
          }
        ]
      }
    },
    {
      "join_optimization": {
        "select_id": 1,
        "steps": [
          {
            "condition_processing": {
              "condition": "WHERE",
              "original_condition": "orders.o_orderDATE =
lineitem.l_shipDATE and orders.o_orderDATE = '1995-01-01' and
orders.o_orderkey = lineitem.l_orderkey",
              "steps": [
                {
                  "transformation": "equality_propagation",
                  "resulting_condition": "multiple equal(DATE'1995-01-01',
orders.o_orderDATE, lineitem.l_shipDATE) and multiple
equal(orders.o_orderkey, lineitem.l_orderkey)"
                },
                {
                  "transformation": "constant_propagation",
                  "resulting_condition": "multiple equal(DATE'1995-01-01',
orders.o_orderDATE, lineitem.l_shipDATE) and multiple
equal(orders.o_orderkey, lineitem.l_orderkey)"
                },
                {
                  "transformation": "trivial_condition_removal",
                  "resulting_condition": "multiple equal(DATE'1995-01-01',
orders.o_orderDATE, lineitem.l_shipDATE) and multiple
equal(orders.o_orderkey, lineitem.l_orderkey)"
                }
              ]
            }
          },
          {
            "table_dependencies": [
              {
                "table": "orders",
                "row_may_be_null": false,
                "map_bit": 0,
                "depends_on_map_bits": []
              },
              {
                "table": "lineitem",
                "row_may_be_null": false,
                "map_bit": 1,
                "depends_on_map_bits": []
              }
            ]
          },
```

Why JSON?

- It's human-readable
- It's easy to extend
- It's machine-readable

```sql
select
  JSON_DETAILED(JSON_EXTRACT(trace, '$**.rows_estimation'))
from
  information_schema.optimizer_trace;
```

```json
      ...
      },
      {
        "rows_estimation": [
          {
            "selectivity_for_indexes": [],
            "selectivity_for_columns": [],
            "cond_selectivity": 1
          },
          {
            "table": "t1",
            "table_scan": {
              "rows": 1000,
              "cost": 4
            }
          }
        ]
      },
      {
        "considered_execution_plans": [
          {
           ...
```

→

```json
[
    [
        {
            "selectivity_for_indexes": [],
            "selectivity_for_columns": [],
            "cond_selectivity": 1
        },
        {
            "table": "t1",
            "table_scan":
            {
                "rows": 1000,
                "cost": 4
            }
        }
    ]
]
```

Insert: JSON 101

- value:

  object | array |

  "string" | number | true | false | null

- array:

  [value, ...]

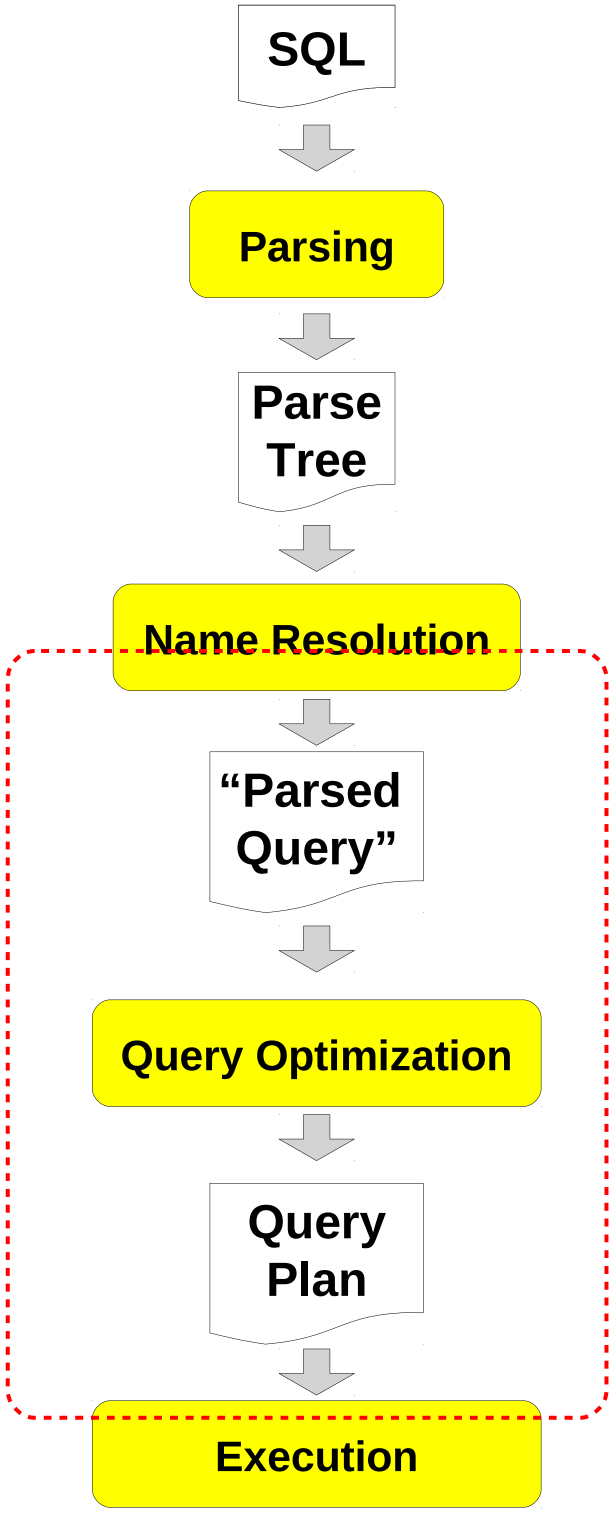
- object:

  { "name" : value, ... }

Interpreting

Optimizer Trace

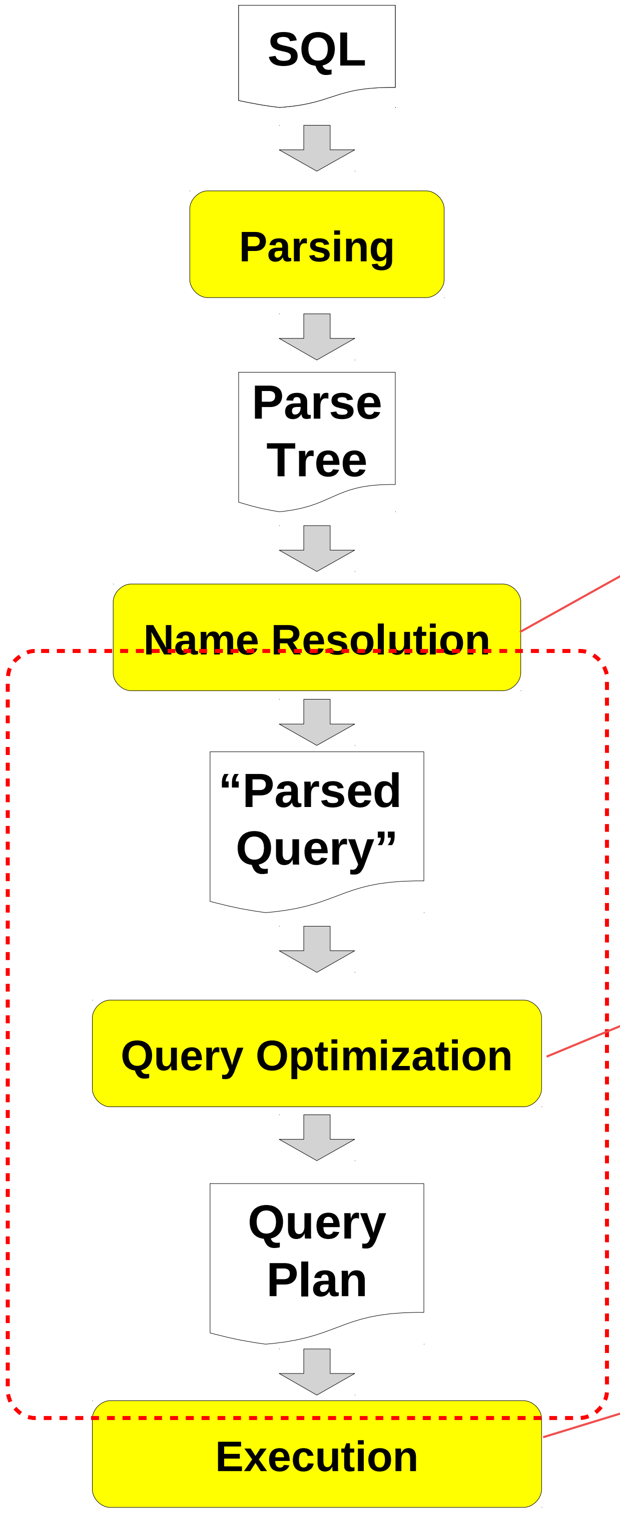
Query processing



Optimizer trace

- Mostly covers Query Optimization
- Covers a bit of Name Resolution
  - Because some optimizations are or were done there
- Covers a bit of execution
  - (More of it in MySQL)

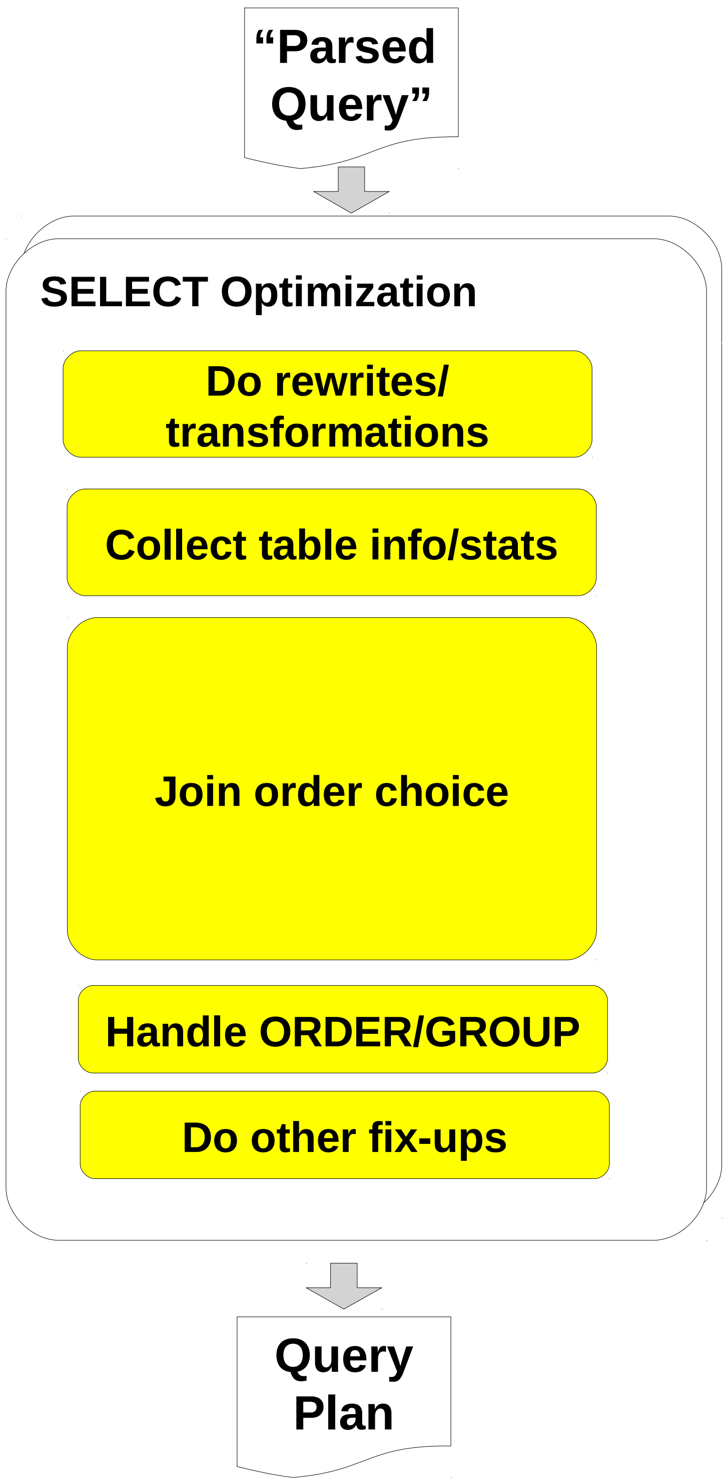
Query processing in optimizer trace




```
select * from t1 where col1 <3 and col2 ='foo'
```

```
TRACE: {
  "steps": [
    {
      "join_preparation": {
        "select_id": 1,
        "steps": [
          {
            "expanded_query": "select t1.col1 AS
col1,t1.col2 AS col2,t1.col3 AS col3 from t1 where t1.col1
< 3 and t1.col2 = 'foo'"
          }
        ]
      }
    },
    {
      "join_optimization": {
        "select_id": 1,
        "steps": [
           ...
           ...
           ...
        ]
      }
    },
    {
      "join_execution": {
        "select_id": 1,
        "steps": []
      }
    }
  ]
}
```

Query Optimization



- Each SELECT is optimized separately
  - Except when it is merged, converted to semi-join, etc
- Optimization phases are roughly as shown in the diagram
  - With some exceptions

Query Optimization in optimizer trace


```
  "join_optimization": {
    "select_id": 1,
    "steps": [
      {
        "condition_processing": { }
      },
      {
        "table_dependencies": [... ]
      },
      ...
      {
        "ref_optimizer_key_uses": [
          ...
        ]
      },
      {
        "rows_estimation": [
          ...
        ]
      },
      {
        "considered_execution_plans": [
          ...
        ]
      },
      {
        "best_join_order": ["table1", "table2", ...]
      },
      ...
      {
        "attaching_conditions_to_tables": { ... }
      }
    ]
  }
}
```

ref optimizer

- Try a join

```
explain select * from t1, t2 where t1.key_col=t2.key_col
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t2	ALL	key_col	NULL	NULL	NULL	1000	Using where
1	SIMPLE	t1	ref	key_col	key_col	5	t2.key_col	1	Using index condition

- Could this use join order t1->t2 and use ref(t2.key_col) ?
  - t1.key_col is INT
  - t2.key_col is VARCHAR(100) collate utf8_general_ci
- Look at ref_optimizer_keyuses
  - Can make lookups using t1.key_col
  - Not vice-versa.

```
"ref_optimizer_key_uses": [
  {
    "table": "t1",
    "field": "key_col",
    "equals": "t2.key_col",
    "null_rejecting": true
  }
]
```

ref optimizer (2)

- Change table t1 for t3:
  - t3.key_col is VARCHAR(100) collate utf8_unicode_ci
  - t2.key_col is VARCHAR(100) collate utf8_general_ci

```
explain select * from t1, t2 where t3.key_col=t2.key_col
```

- Check the trace
  - Now, lookup is possible in both directions.
- Take-away:
  - **ref_optimizer_keyuses** lists potential ref accesses.

```
"ref_optimizer_key_uses": [
  {
    "table": "t3",
    "field": "key_col",
    "equals": "t2.key_col",
    "null_rejecting": true
  },
  {
    "table": "t2",
    "field": "key_col",
    "equals": "t3.key_col",
    "null_rejecting": true
  }
]
```

Range Optimizer

Range optimizer: ranges (1)

- Inference of ranges from WHERE/ON conditions can get complex
  - Multi-part keys
  - Inference using equalities (col1=col2 AND col1<10 -> col2<10)
  - EXPLAIN [FORMAT=JSON] just shows used_key_parts
- Example with multi-part keys:

```
create table some_events (
  start_date DATE,
  end_date   DATE,
  ...
  KEY (start_date, end_date)
);
```

```
select ...
from some_events as TBL
where
  start_date >= '2019-06-24'
and
  end_date   <= '2019-06-28'
```

```
"rows_estimation": [
  {
    "table": "some_events",
    "range_analysis": {
      ...
      "analyzing_range_alternatives": {
        "range_scan_alternatives": [
          {
            "index": "start_date",
            "ranges": ["(2019-06-24,NULL) < (start_date,end_date)"],
            "rowid_ordered": false,
            "using_mrr": false,
            "index_only": false,
            "rows": 4503,
            "cost": 5638.8,
            "chosen": true
          }
        ]
```

Range optimizer: ranges (2)

- Inference of ranges from WHERE/ON conditions can get complex
  - Multi-part keys
  - Inference using equalities (col1=col2 AND col1<10 -> col2<10)
  - EXPLAIN [FORMAT=JSON] just shows used_key_parts
- Example with multi-part keys:

```
create table some_events (
  start_date DATE,
  end_date   DATE,
  ...
  KEY (start_date, end_date)
);
```

```
select ...
from some_events as TBL
where
  start_date >= '2019-06-24'
and
  end_date   <= '2019-06-28'
```

start_date <= end_date, so:

```
and
  start_date <= '2019-06-28'
and
  end_date   >= '2019-06-24'
```

```
"rows_estimation": [
  {
    "table": "some_events",
    "range_analysis": {
      ...
      "analyzing_range_alternatives": {
        "range_scan_alternatives": [
          {
            "index": "start_date",
            "ranges": [
              "(2019-06-24,2019-06-24) <= (start_date,end_date) <=
                                            (2019-06-28,2019-06-28)"
            ],
            "rowid_ordered": false,
            "using_mrr": false,
            "index_only": false,
            "rows": 1,
            "cost": 1.345170888,
            "chosen": true
          }
        ],
```

Range optimizer: multiple ranges

- Another example: multiple ranges

```
select *
from some_events
where start_date in ('2019-06-01','2019-06-02','2019-06-03') and end_date='2019-06-10'
```

```
 "rows_estimation": [
   {
     "table": "some_events",
     "range_analysis": {
       ...
       "analyzing_range_alternatives": {
         "range_scan_alternatives": [
           {
             "index": "start_date",
             "ranges": [
               "(2019-06-01,2019-06-10) <= (start_date,end_date) <= (2019-06-01,2019-06-10)",
               "(2019-06-02,2019-06-10) <= (start_date,end_date) <= (2019-06-02,2019-06-10)",
               "(2019-06-03,2019-06-10) <= (start_date,end_date) <= (2019-06-03,2019-06-10)"
             ],
             "rowid_ordered": false,
             "using_mrr": false,
             "index_only": false,
             "rows": 3,
             "cost": 3.995512664,
             "chosen": true
           }
         ],
```

Loose index scan

- Loose index scan is a useful optimization
  - The choice whether to use it is cost-based (depends also on ANALYZE TABLE)
  - It has complex applicability criteria

```
create table t (
  kp1 INT,
  kp2 INT,
  kp3 INT,
  ...
  KEY (kp1, kp2, kp3)
);
```

- Try two very similar queries

```
select
  min(kp2)
from t
where
  kp2>=10
group by
  kp1
```

  - This one is using Loose Scan

```
select
  min(kp2)
from t
where
  kp2>=10 and kp3<10
group by
  kp1
```

  - This one is NOT using Loose Scan

Loose Index Scan (2)

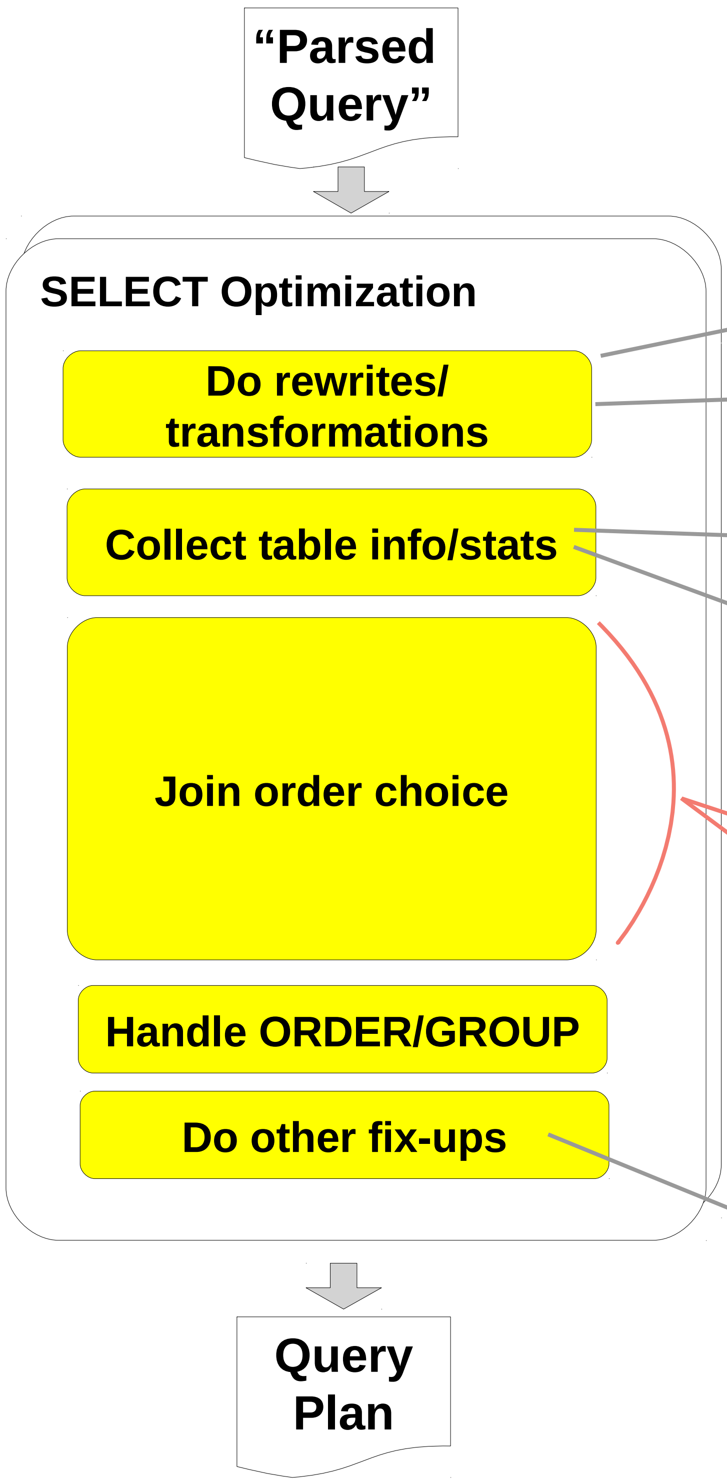
- Compare optimizer traces:

```
"rows_estimation": [
  {
    "table": "t",
    "range_analysis": {
      ...
      "group_index_range": {
        "potential_group_range_indexes": [
          {
            "index": "kp1",
            "covering": true,
            "ranges": ["(10) <= (kp2)"],
            "rows": 67,
            "cost": 90.45
          }
        ]
      },
      "best_group_range_summary": {
        "type": "index_group",
        "index": "kp1",
        "min_max_arg": "kp2",
        "min_aggregate": true,
        "max_aggregate": false,
        "distinct_aggregate": false,
        "rows": 67,
        "cost": 90.45,
        "key_parts_used_for_access": ["kp1"],
        "ranges": ["(10) <= (kp2)"],
        "chosen": true
      },
```

```
"rows_estimation": [
  {
    "table": "t23",
    "range_analysis": {
      ...
      "group_index_range": {
        "potential_group_range_indexes": [
          {
            "index": "kp1",
            "covering": true,
            "usable": false,
            "cause": "keypart after infix in query"
          }
        ]
```

Join Optimizer

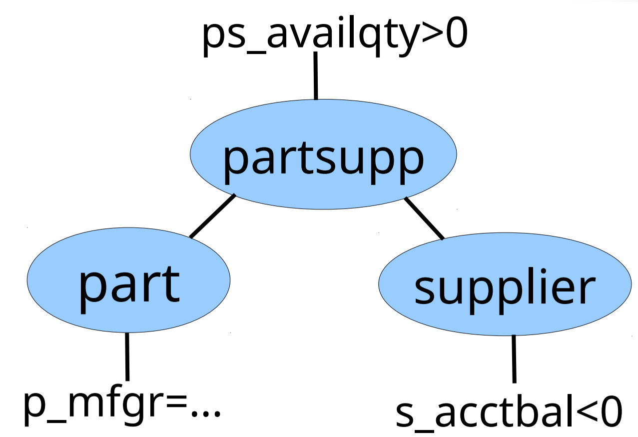
Tracing JOIN Optimizer




```
"join_optimization": {
  "select_id": 1,
  "steps": [
    {
      "condition_processing": { }
    },
    {
      "table_dependencies": [... ]
    },
    ...
    {
      "ref_optimizer_key_uses": [
         ...
      ]
    },
    {
      "rows_estimation": [
         ...
      ]
    },
    {
      "considered_execution_plans": [
         ...
      ]
    },
    {
      "best_join_order": ["table1", "table2", ...]
    },
    ...
    {
      "attaching_conditions_to_tables": { ... }
    }
  ]
 }
}
```

A join query

```
select *
from
  part, supplier, partsupp
where
  p_partkey=ps_partkey and ps_suppkey=s_suppkey and
  s_acctbal<0 and
  ps_availqty > 0 and
  p_mfgr='Manufacturer#3'
```



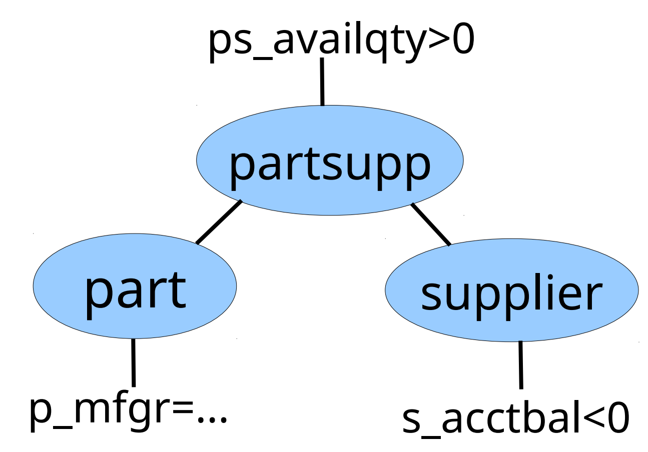
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	supplier	range	PRIMARY,s_acctbal	s_acctbal	9	NULL	886	Using index condition
1	SIMPLE	partsupp	ref	PRIMARY,i_ps_par...	i_ps_suppkey	4	supplier.s_suppkey	45	Using where
1	SIMPLE	part	eq_ref	PRIMARY	PRIMARY	4	partsupp.ps_partkey	1	Using where

```
"join_optimization": {
  ...
  {
    "considered_execution_plans": [
      ...
      ...
      ...
    ]
  },
  {
    "best_join_order": ["supplier", "partsupp", "part"]
  },
```

- **best_join_order** shows the final picked join order
- **considered_execution_plans** is a log of join order choice.
  - it can have *a lot* of content

Join order enumeration

```
+------+-------------+----------+-------+---------------------+--------------+-----
| id   | select_type | table    | type  | possible_keys       | key          | key_
+------+-------------+----------+-------+---------------------+--------------+-----
|    1 | SIMPLE      | supplier | range | PRIMARY,s_acctbal   | s_acctbal    | 9
|    1 | SIMPLE      | partsupp | ref   | PRIMARY,i_ps_par... | i_ps_suppkey | 4
|    1 | SIMPLE      | part     | eq_ref| PRIMARY             | PRIMARY      | 4
+------+-------------+----------+-------+---------------------+--------------+-----
```



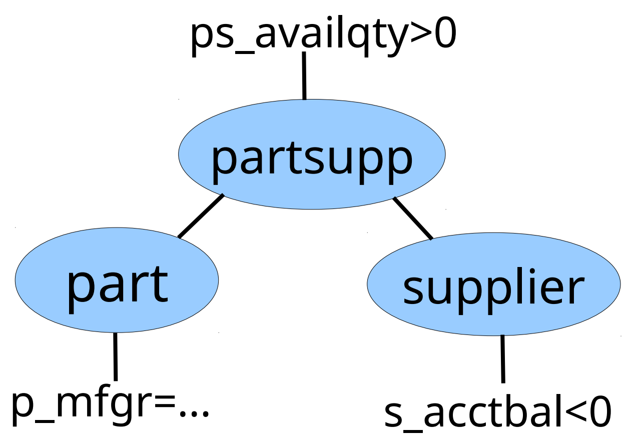

```
$ grep -A1 plan_prefix /tmp/trace.txt

          "plan_prefix": [],
          "table": "supplier",
--
              "plan_prefix": ["supplier"],
              "table": "part",
--
                  "plan_prefix": ["supplier", "part"],
                  "table": "partsupp",
--
              "plan_prefix": ["supplier"],
              "table": "partsupp",
--
                  "plan_prefix": ["supplier", "partsupp"],
                  "table": "part",
--
          "plan_prefix": [],
          "table": "part",
--
              "plan_prefix": ["part"],
              "table": "supplier",
--
              "plan_prefix": ["part"],
              "table": "partsupp",
--
          "plan_prefix": [],
          "table": "partsupp",
```

- There is so much content we'll use **grep**.
- Join order is constructed in a top-down (first-to-last) way
  - **plan_prefix –** already constructed
  - **table –** the table we're trying to add
- Shows considered join prefixes
  - Non-promising prefixes are pruned away

Join order enumeration: supplier

```
+------+-------------+----------+-------+--------------------+--------------+-----
| id   | select_type | table    | type  | possible_keys      | key          | key_
+------+-------------+----------+-------+--------------------+--------------+-----
|    1 | SIMPLE      | supplier | range | PRIMARY,s_acctbal  | s_acctbal    | 9
|    1 | SIMPLE      | partsupp | ref   | PRIMARY,i_ps_par...| i_ps_suppkey | 4
|    1 | SIMPLE      | part     | eq_ref| PRIMARY            | PRIMARY      | 4
+------+-------------+----------+-------+--------------------+--------------+-----
```



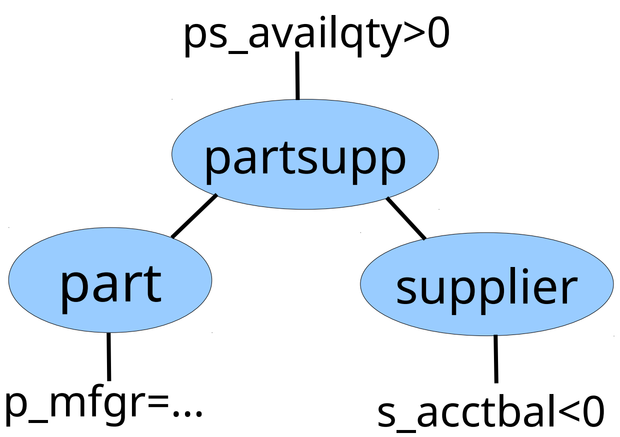

```
{
  "plan_prefix": [],
  "table": "supplier",
  "best_access_path": {
    "considered_access_paths": [
      {
        "access_type": "range",
        "resulting_rows": 886,
        "cost": 1063.485592,
        "chosen": true
      }
    ],
    "chosen_access_method": {
      "type": "range",
      "records": 886,
      "cost": 1063.485592,
      "uses_join_buffering": false
    }
  },
  "rows_for_plan": 886,
  "cost_for_plan": 1240.685592,
```

- **best_access_path –** a function adding a table the prefix.
- Shows considered ways to read the table and the picked one
  - Also #rows and costs.

Join order enumeration: supplier,partsupp

id	select_type	table	type	possible_keys	key	key_
1	SIMPLE	supplier	range	PRIMARY,s_acctbal	s_acctbal	9
1	SIMPLE	partsupp	ref	PRIMARY,i_ps_par...	i_ps_suppkey	4
1	SIMPLE	part	eq_ref	PRIMARY	PRIMARY	4

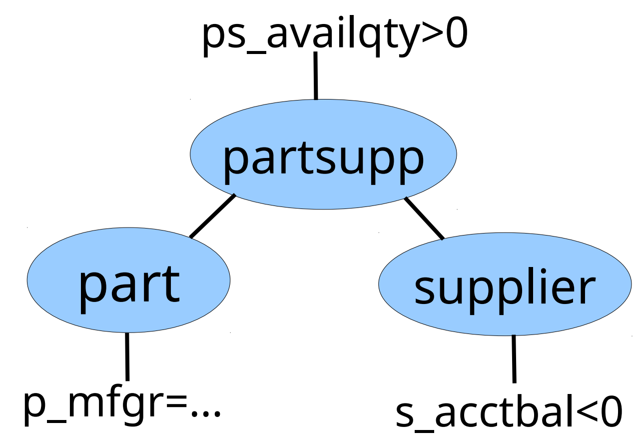
```
{
  "plan_prefix": ["supplier"],
  "table": "partsupp",
  "best_access_path": {
    "considered_access_paths": [
      {
        "access_type": "ref",
        "index": "i_ps_suppkey",
        "rows": 45,
        "cost": 40761.83998,
        "chosen": true
      },
      {
        "access_type": "scan",
        "resulting_rows": 909440,
        "cost": 12847,
        "chosen": false
      }
    ],
    "chosen_access_method": {
      "type": "ref",
      "records": 45,
      "cost": 40761.83998,
      "uses_join_buffering": false
    }
  },
  "rows_for_plan": 39870,
  "cost_for_plan": 49976.52557,
```



Join order enumeration: supplier, partsupp, part

id	select_type	table	type	possible_keys	key	key_
1	SIMPLE	supplier	range	PRIMARY,s_acctbal	s_acctbal	9
1	SIMPLE	partsupp	ref	PRIMARY,i_ps_par...	i_ps_suppkey	4
1	SIMPLE	part	eq_ref	PRIMARY	PRIMARY	4

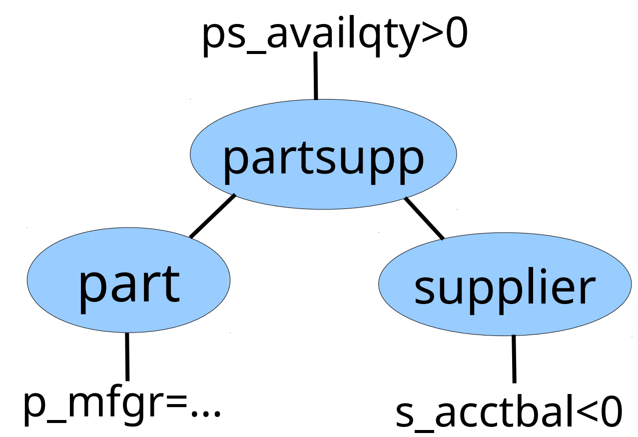
```
{
  "plan_prefix": ["supplier", "partsupp"],
  "table": "part",
  "best_access_path": {
    "considered_access_paths": [
      {
        "access_type": "eq_ref",
        "index": "PRIMARY",
        "rows": 1,
        "cost": 39870,
        "chosen": true
      },
      {
        "access_type": "scan",
        "resulting_rows": 197805,
        "cost": 131300,
        "chosen": false
      }
    ],
    "chosen_access_method": {
      "type": "eq_ref",
      "records": 1,
      "cost": 39870,
      "uses_join_buffering": false
    }
  },
  "rows_for_plan": 39870,
  "cost_for_plan": 97820.52557,
  "estimated_join_cardinality": 39870
}
```



Join order enumeration: part

id	select_type	table	type	possible_keys	key	key_
1	SIMPLE	supplier	range	PRIMARY,s_acctbal	s_acctbal	9
1	SIMPLE	partsupp	ref	PRIMARY,i_ps_par...	i_ps_suppkey	4
1	SIMPLE	part	eq_ref	PRIMARY	PRIMARY	4


```
{
  "plan_prefix": [],
  "table": "part",
  "best_access_path": {
    "considered_access_paths": [
      {
        "access_type": "scan",
        "resulting_rows": 197805,
        "cost": 2020,
        "chosen": true
      }
    ],
    "chosen_access_method": {
      "type": "scan",
      "records": 197805,
      "cost": 2020,
      "uses_join_buffering": false
    }
  },
  "rows_for_plan": 197805,
  "cost_for_plan": 41581,
```

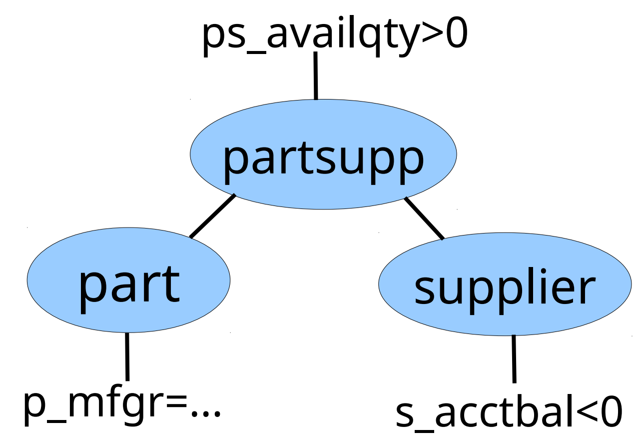



- Let's follow the other possible plan, part->partsupp->supplier.

Join order enumeration: part, partsupp

id	select_type	table	type	possible_keys	key	key_
1	SIMPLE	supplier	range	PRIMARY,s_acctbal	s_acctbal	9
1	SIMPLE	partsupp	ref	PRIMARY,i_ps_par...	i_ps_suppkey	4
1	SIMPLE	part	eq_ref	PRIMARY	PRIMARY	4

```
{
  "plan_prefix": ["part"],
  "table": "partsupp",
  "best_access_path": {
    "considered_access_paths": [
      {
        "access_type": "ref",
        "index": "PRIMARY",
        "rows": 1,
        "cost": 198088.8932,
        "chosen": true
      },
      {
        "access_type": "ref",
        "index": "i_ps_partkey",
        "rows": 2,
        "cost": 593472.9471,
        "chosen": false,
        "cause": "cost"
      },
      {
        "access_type": "scan",
        "resulting_rows": 909440,
        "cost": 1631569,
        "chosen": false
      }
    ],
    "chosen_access_method": {
      "type": "ref",
      "records": 1,
      "cost": 198088.8932,
      "uses_join_buffering": false
    }
  },
  "rows_for_plan": 197805,
  "cost_for_plan": 279230.8932,
  "pruned_by_cost": true
}
```



- **pruned_by_cost: true.**

Join optimizer take-aways

- **considered_execution_paths** traces the join order choice
  - It can get very large, use grep
    - **plan_prefix**, **table**
- **best_join_order** shows the picked order.

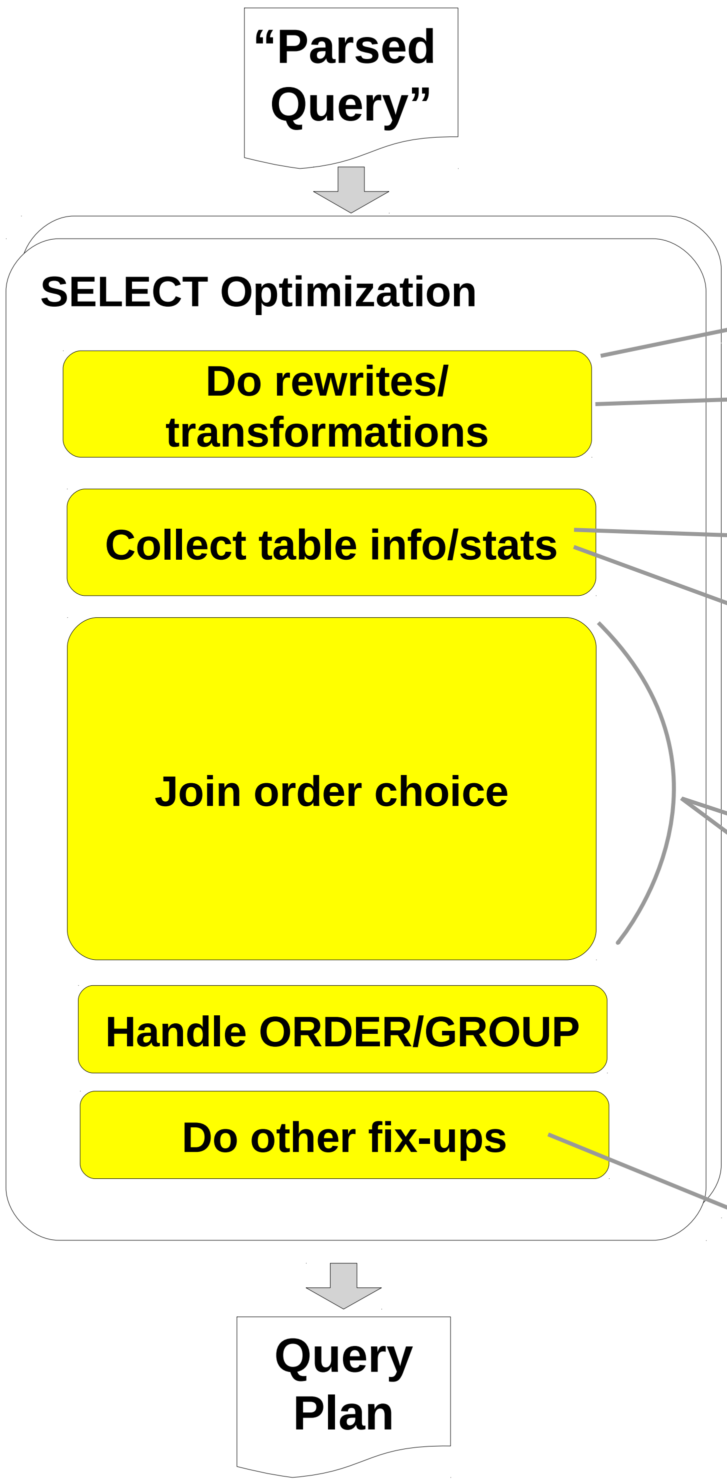
```
"join_optimization": {
  "select_id": 1,
  "steps": [
    {
      "condition_processing": { }
    },
    {
      "table_dependencies": [... ]
    },
    ...
    {
      "ref_optimizer_key_uses": [
         ...
       ]
    },
    {
      "rows_estimation": [
         ...
      ]
    },
    {
      "considered_execution_plans": [
        ...
      ]
    },
    {
      "best_join_order": ["table1", "table2", ...]
    },
    ...
    {
      "attaching_conditions_to_tables": { ... }
    }
  ]
 }
}
```

- Hardcore JSONPath users might want to use searches like

```
JSON_EXTRACT(trace,
 '$**.considered_execution_plans[?(@table="supplier")]')
```

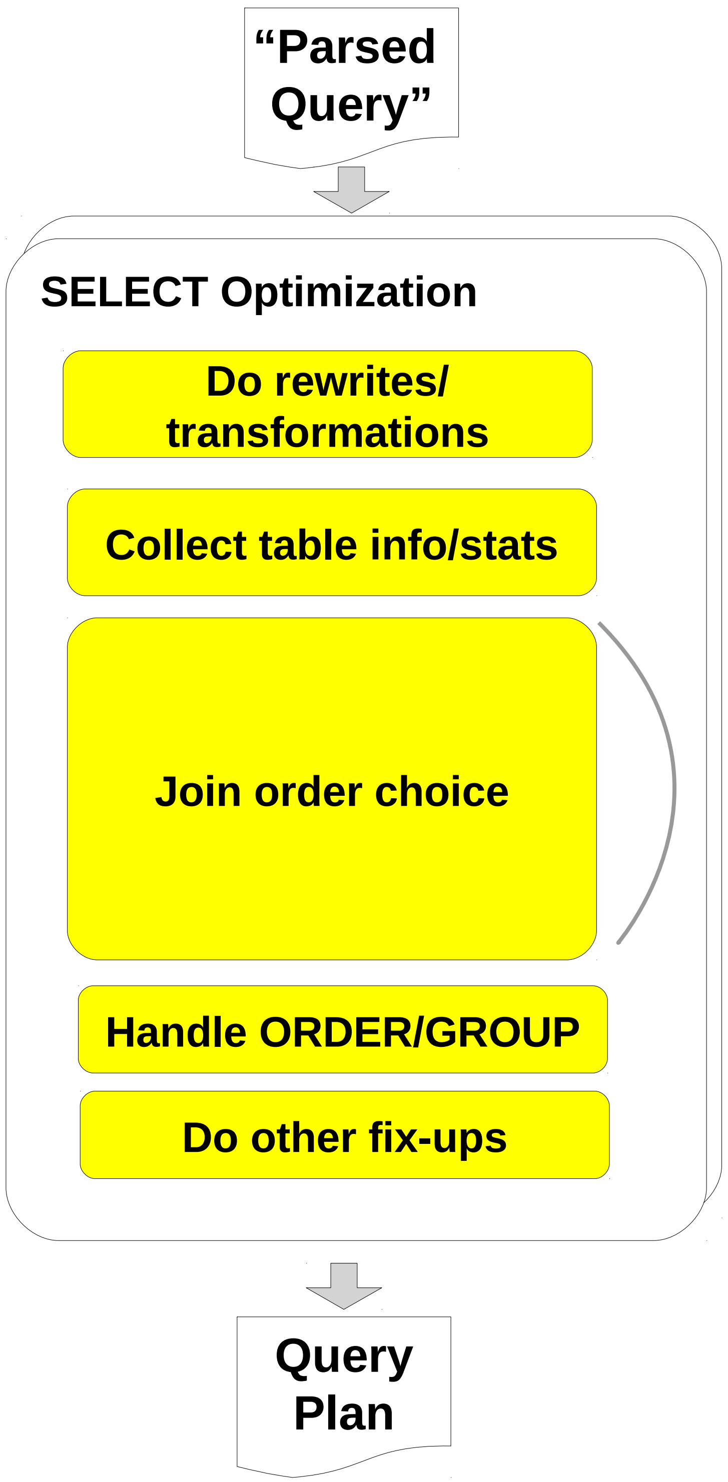
Can't do this, because MariaDB's JSONPath engine doesn't support filters.

Covered so far




```
"join_optimization": {
  "select_id": 1,
  "steps": [
    {
      "condition_processing": { }
    },
    {
      "table_dependencies": [... ]
    },
    ...
    {
      "ref_optimizer_key_uses": [
         ...
       ]
    },
    {
      "rows_estimation": [
         ...
      ]
    },
    {
      "considered_execution_plans": [
        ...
      ]
    },
    {
      "best_join_order": ["table1", "table2", ...]
    },
    ...
    {
      "attaching_conditions_to_tables": { ... }
    }
  ]
 }
}
```

There’s a lot more in the optimizer_trace



- Rewrites, transformations
  - VIEW/CTE merging
  - IN/EXISTS Subquery optimizations
  - MIN/MAX transformation
  - ...
- Collect table stats
  - EITS, histograms
  - Constant tables
  - Table elimination
  - Condition filtering
  - ...
- Fix-ups
  - ORDER/GROUP BY (not just in fix-ups)
  - ...

Optimizer Trace

in MySQL

Optimizer Trace in MySQL

- Similar to MariaDB's
  - User interface
  - Trace elements and structure
- There are differences
  - Optimizers are different
  - Slightly different set of things traced
  - The tracing module has extra features
- Let's review
  - Anything to learn?


```
{
  "steps": [
    {
      "join_preparation": {
        "select#": 1,
        "steps": [
          {
            "expanded_query": "/* select#1 */ select `t1`.`col1` AS
`col1`,`t1`.`col2` AS `col2`,`t1`.`col3` AS `col3` from `t1` where
((`t1`.`col1` < 3) and (`t1`.`col2` = 'foo'))"
          }
        ]
      }
    },
    {
      "join_optimization": {
        "select#": 1,
        "steps": [
          {
            "condition_processing": {
              "condition": "WHERE",
              "original_condition": "((`t1`.`col1` < 3) and (`t1`.`col2` =
'foo'))",
              "steps": [
                {
                  "transformation": "equality_propagation",
                  "resulting_condition": "((`t1`.`col1` < 3) and
(`t1`.`col2` = 'foo'))"
                },
                {
                  "transformation": "constant_propagation",
                  "resulting_condition": "((`t1`.`col1` < 3) and
(`t1`.`col2` = 'foo'))"
                },
                {
                  "transformation": "trivial_condition_removal",
                  "resulting_condition": "((`t1`.`col1` < 3) and
(`t1`.`col2` = 'foo'))"
                }
              ]
            }
          },
          {
            "substitute_generated_columns": {
            }
          },
          {
            "table_dependencies": [
              {
                "table": "`t1`",
                "row_may_be_null": false,
                "map_bit": 0,
                "depends_on_map_bits": [
                ]
              }
            ]
          },
          {
            "ref_optimizer_key_uses": [
            ]
          },
```

Extras in MySQL

- Can capture multiple traces for nested statements
  - Controlled with @@**optimizer_trace_{offset,limit}**
    - Can set to save first/last N traces.
  - Allows to trace statements inside stored procedure/function calls
  - The issue is that other support for nested statements is not present:
    - Can't get EXPLAINs for sub-statements.
    - Can't get statement_time* (*- in some cases, can infer #rows_examined and statement_time from PERFORMANCE_SCHEMA.
    - Is this useful?
- Can omit parts of trace
  - Controlled by @@**optimizer_trace_features**

    `greedy_search=on,range_optimizer=on,dynamic_range=on,repeated_subselect=on`
  - Why not use **JSON_EXTRACT** to get the part of trace you need, instead?
  - Would be useful for global settings e.g. "[Don't] print cost values everywhere".

Extras in MySQL (2)

- Can control JSON formatting
  - SET @@optimizer_trace='enabled=on,one_line=on'
  - SET @@end_markers_in_json=on;
    - Note: the output is not a valid JSON anymore, can't be processed.

```
"join_preparation": {
  "select#": 1,
  "steps": [
    ...
  ] /* steps */
} /* join_preparation */
```

- Tracing covers some parts of query execution
  - Because MySQL didn't have EXPLAIN ANALYZE back then?
  - Can produce a lot of repetitive JSON
  - In MariaDB, this kind of info is shown in ANALYZE FORMAT=JSON output.

```
  "filesort_priority_queue_optimization": {
    "usable": false,
    "cause": "not applicable (no LIMIT)"
  },
  "filesort_execution": [
  ],
  "filesort_summary": {
    "memory_available": 998483,
    "key_size": 1022,
    "row_size": 66561,
    "max_rows_per_buffer": 14,
    "num_rows_estimate": 215,
    "num_rows_found": 0,
    "num_initial_chunks_spilled_to_disk": 0,
    "peak_memory_used": 0,
    "sort_algorithm": "none",
    "sort_mode": "<fixed_sort_key,
packed_additional_fields>"
  }
```

Extra in MariaDB

“More polish”

- Conditions are readable
- Better JSON formatting

```
"attached_conditions_summary": [
  {
    "table": "`t1`",
    "attached": "((`t1`.`col1` < 3) and (`t1`.`col2` = 'foo'))"
  }
```

vs

```
"attached_conditions_summary": [
  {
    "table": "t1",
    "attached": "t1.col1 < 3 and t1.col2 = 'foo'"
  }
```

- Date constants are printed as dates, not hex
- etc
- The output is correct
  - Caution: https://bugs.mysql.com/bug.php?id=95824

Conclusions

- Optimizer Trace is available in MariaDB
- Shows details about query optimization
  - Analyze it yourself
  - Submit traces when discussing/reporting optimizer issues
- MySQL has a similar feature
  - Has some extras but they don't seem important
- Future
  - Print more useful info in the trace
  - ...

Thanks!