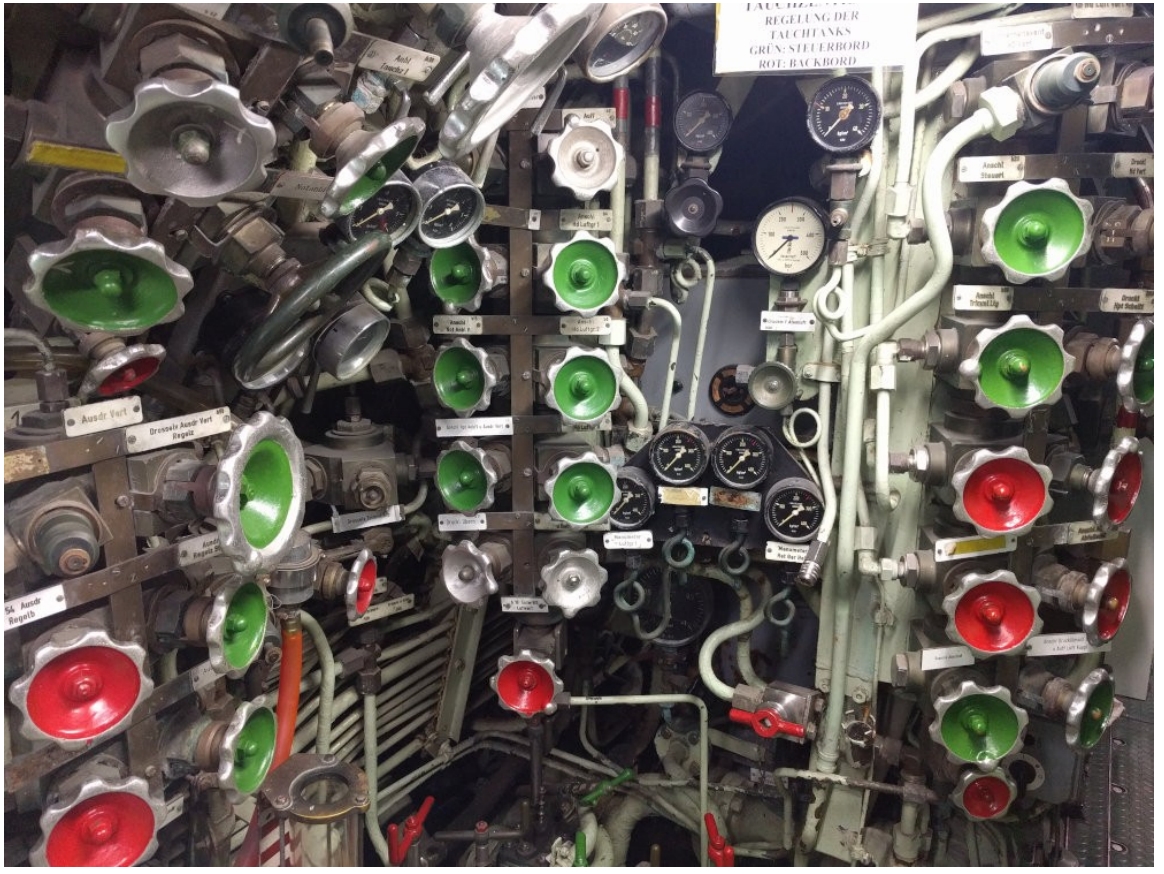
How Optimizer Works



Oleksandr „Sanja“ Byelkin, MariaDB, 2019

Query execution

- *Parsing*
- *Preparing*
- **Optimizing**
- *Executing*

How make optimizations visible

- `EXPALIN EXTENDED …; SHOW WARNINGS;`
  - Very old, shows only results of the optimisation
- `EXPLAIN/ANALYZE FORMAT=JSON …;`
  - 10.1, more info
- `set optimizer_trace=1; …;`

  `select * from information_schema.optimizer_trace;`
  - 10.4 explains decisions taking process

Pre-Optimizer changes (parsing)

- `Exp IN (single_val) → Exp = single_val`
- `Exp IN NOT (single_val) → Exp <> single_val`
- `WHERE col → WHERE col <> 0`
- `WHERE NOT col → WHERE col = 0`

Optimization steps

- Query transformations (non-cost-based, identical, open ways for other optimizations)
- Join optimization - preliminary phase (info collection)
- Join optimization (greedy search)
- Join optimization - plan refinement
- Other kinds of plan refinement

Query transformations:

- Derived/view merge
- IN predicate → IN subquery
- EXISTS → IN (opens EXISTS → IN or Materialization)
- MIN/MAX subquery
- Semi-join
- IN materialization
- outer → inner joins
- Condition optimization

Query transformations: Derived/view merge

```
select * from
  (select * from t1 where a>1)
  as tt;
```

If it can not be merged then cause will be mentioned

```
"join_preparation": {
  "select_id": 1,
  "steps": [
    {
      "derived": {
        "table": "tt",
        "select_id": 2,
        "algorithm": "merged"
      }
    },
```

Query transformations: IN predicate → IN subquery

```
set @@in_predicate_conversion_threshold= 2;
select * from t1 where a in (1,2);

explain extended
select * from t1 where a in (1,2);
id  select_type  table   type possible_keys   key key_len  ref  rows filtered Extra

…

Warnings:
Note 1003 /* select#1 */ select `test`.`t1`.`a` AS `a`,`test`.`t1`.`b` AS `b` from
`test`.`t1` semi join ((values (1),(2)) `tvc_0`) where 1
```

Semi-join is a result of conversion to IN-Subquery

Query transformations: EXISTS → IN

```
EXPLAIN EXTENDED
SELECT Name FROM Country
           WHERE (EXISTS (select 1 from City where City.Population > 100000 and
                          Code = Country) OR
                  Name LIKE 'L%') AND
                 Surfacearea > 1000000;
+--+------------+-------+----+------------------+----+-------+----+----+---------+-----------+
|Id|select_type |table  |type|possible_keys     |key |key_len|ref |rows|filtered |Extra      |
|1 |PRIMARY     |Country|ALL |Name,SurfaceArea  |NULL|NULL   |NULL|239 |29.71    |Using where|
|2 |MATERIALIZED|City   |ALL |Population,Country|NULL|NULL   |NULL|4079|87.45    |Using where|
+--+------------+-------+----+------------------+----+-------+----+----+---------+-----------+
Warnings:
Note 1276 Field or reference 'world.Country.Code' of SELECT #2 was resolved in SELECT #1
Note 1003 /* select#1 */ select `world`.`Country`.`Name` AS `Name` from `world`.`Country` where
(<expr_cache><`world`.`Country`.`Code`>(<in_optimizer>(`world`.`Country`.`Code`,`world`.`Countr
y`.`Code` in ( <materialize> (/* select#2 */ select `world`.`City`.`Country` from
`world`.`City` where `world`.`City`.`population` > 100000 ),
<primary_index_lookup>(`world`.`Country`.`Code` in <temporary table> on distinct_key where
`world`.`Country`.`Code` = `<subquery2>`.`Country`)))) or `world`.`Country`.`Name` like 'L%')
and `world`.`Country`.`SurfaceArea` > 1000000
```

Query transformations: MIN/MAX Subquery

```
explain extended

SELECT a FROM t1 WHERE b < ANY ( SELECT b FROM t1 GROUP BY b );

id  select_type table   type    possible_keys   key key_len ref rows
filtered    Extra

...

Warnings:

Note    1003    /* select#1 */ select `test`.`t1`.`a` AS `a` from
`test`.`t1` where <nop>(<in_optimizer>(`test`.`t1`.`b`,(/* select#2 */
select max(`test`.`t1`.`b`) from `test`.`t1`) > <cache>(`test`.`t1`.`b`)))
```

Query transformations: semi-join & IN materialization

```
select * from t1
  where a in (select pk from t2);
```

```
"steps": [
 {
   "transformation": {
     "select_id": 2,
     "from": "IN (SELECT)",
     "to": "materialization",
     "sjm_scan_allowed": true,
     "possible": true
   }
 },
 {
   "transformation": {
     "select_id": 2,
     "from": "IN (SELECT)",
     "to": "semijoin",
     "chosen": true
   }
 },
```

Query transformations: outer → inner joins

```
explain extended
SELECT * FROM t1 LEFT JOIN t2 ON t2.a=t1.a WHERE t2.b < 5;
id select_type  table  type   possible_keys key key_len   ref
rows   filtered  Extra
...
Warnings:
Note   1003   select `test`.`t1`.`a` AS `a`,`test`.`t1`.`b`
AS `b`,`test`.`t2`.`a` AS `a`,`test`.`t2`.`b` AS `b` from
`test`.`t1` join `test`.`t2` where `test`.`t2`.`a` =
`test`.`t1`.`a` and `test`.`t2`.`b` < 5
```

“t2.b < 5“ is null rejecting

Query transformations: Condition optimization

- Multiple-equality part #1: Equality list building
- Constant propagation
- Trivial condition detection

Query transformations: Condition optimization

```
create table t1
  (a int primary key, b int);
select * from t1 where t1.a=0;
```

```
          {
            "rows_estimation": [
              {
                "table": "t1",
                "rows": 1,
                "cost": 1,
                "table_type": "const"
              }
            ]
          },
```

```
explain extended
select * from t1 where t1.a=0;
...
Warnings:
Note 1003 select 0 AS `a`,0 AS `b` from `test`.`t1` where 1
```

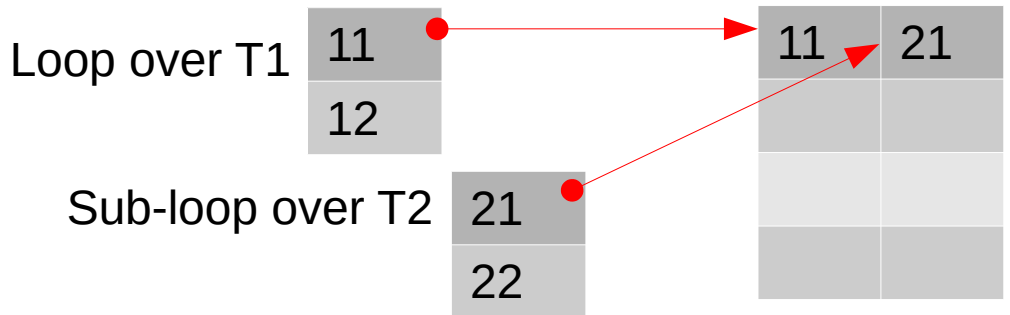
```
"condition_processing": {
  "condition": "WHERE",
  "original_condition": "t1.a = 0",
  "steps": [
    {
      "transformation": "equality_propagation",
      "resulting_condition": "multiple equal(0, t1.a)"
    },
    {
      "transformation": "constant_propagation",
      "resulting_condition": "multiple equal(0, t1.a)"
    },
    {
      "transformation": "trivial_condition_removal",
      "resulting_condition": "multiple equal(0, t1.a)"
    }
  ]
```

Optimization steps

- Query transformations (non-cost-based identical, open way for other optimizations)
- Join optimization - preliminary phase (info collection)
- Join optimization (greedy search)
- Join optimization - plan refinement
- Other kinds of plan refinement

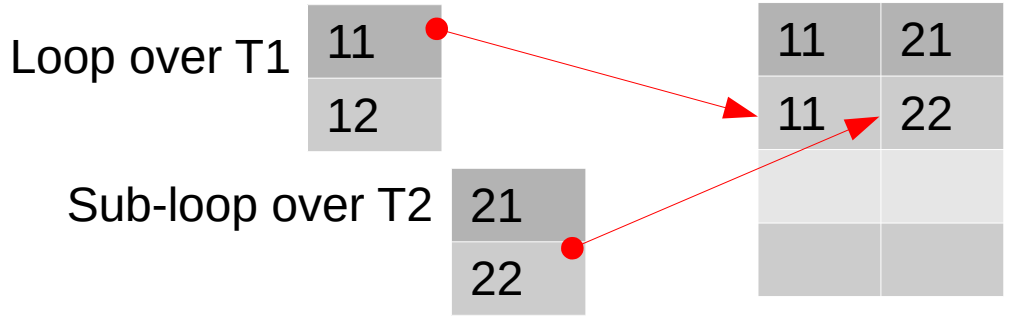
JOIN

MariaDB uses Nested loops



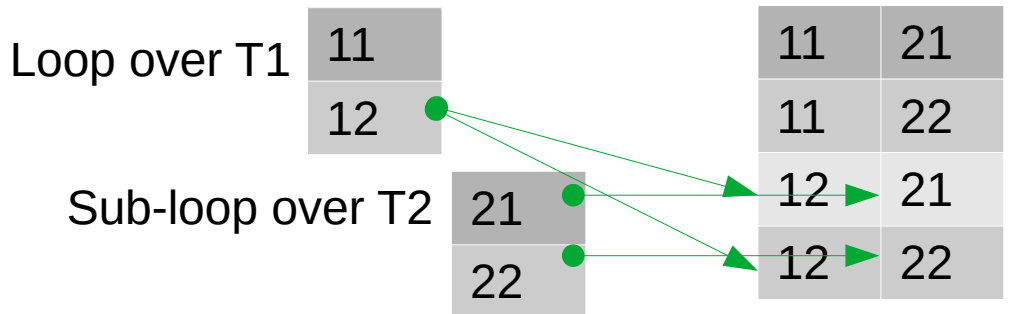
JOIN

MariaDB uses Nested loops



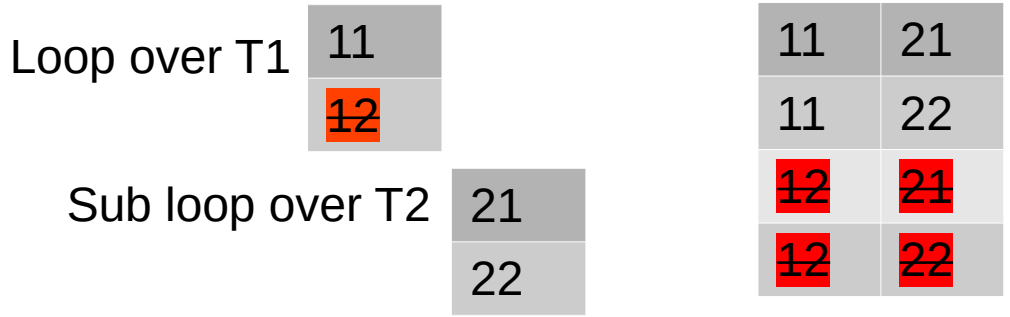
JOIN

MariaDB uses Nested loops



JOIN

MariaDB uses Nested loops



Each filtered row upper level multiply gain on lower level, so then more we filter on upper level then better.

Push Model (others)

Pull Model:

```
class query_plan_node
 { ...
  virtual int get_next();
 ...}
```

Upper level call get_next()

Pull Model (MariaDB)

```
Loop 1 (do_subselect())
  Loop 2 (do_subselect())
    …
      Loop N (do_subselect())
        send_data()
```

Join optimization - preliminary phase

- Check conditions/tables for possible “ref” access
- Get information about each table
  - Range analysis: produces possible range access
  - Partition pruning finds used partitions
  - Condition selectivity estimation (Use the range analysis with histograms instead of indexes)

Join optimization - preliminary phase

```
create table t1 (a int primary key, ...
create table t2 (a int primary key, …

select * from t1,t2
  where
    t1.a=t2.a and
    t2.a>1 and t2.a < 5;
```

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

Join optimization - preliminary phase

```
"rows_estimation": [
  {
    "table": "t1",
    "range_analysis": {
      "table_scan": {
        "rows": 3,
        "cost": 4.7066
      },
```

```
"potential_range_indexes": [
  {
    "index": "PRIMARY",
    "usable": true,
    "key_parts": ["a"]
  }
],
"setup_range_conditions": [],
"group_index_range": {
  "chosen": false,
  "cause": "not single_table"
},
```

```
"analyzing_range_alternatives": {
 "range_scan_alternatives": [
   {
     "index": "PRIMARY",
     "ranges": ["(1) < (a) <
                (5)"],
     "rowid_ordered": false,
     "using_mrr": false,
     "index_only": false,
     "rows": 1,
     "cost": 2.5021,
     "chosen": true
   }
 ],
```

```
"analyzing_roworder_intersect": {
     "cause": "too few roworder scans"
   },
```

Join optimization - preliminary phase

t1	t2

<pre>"chosen_range_access_summary": {
  "range_access_plan": {
  "type": "range_scan",
  "index": "PRIMARY",
  "rows": 1,
  "ranges": ["(1) < (a) < (5)"]
  },
  "rows_for_plan": 1,
  "cost_for_plan": 2.5021,
  "chosen": true</pre>
	<pre>"chosen_range_access_summary": {
	  "range_access_plan": {
	  "type": "range_scan",
	  "index": "PRIMARY",
	  "rows": 3,
	  "ranges": ["(1) < (a) < (5)"]
	  },
	  "rows_for_plan": 3,
	  "cost_for_plan": 5.0064,
	  "chosen": true</pre>

Optimization steps

- Query transformations (non-cost-based identical, open way for other optimizations)
- Join optimization - preliminary phase (info collection)
- Join optimization (greedy search)
- Join optimization - plan refinement
- Other kinds of plan refinement

Join optimization (greedy search)

- Limited by depth greedy search (recursive search)
  - use best variant on each step
  - reject branch (pruning) if we see that there is better
  - If depth is bigger then number of tables we get exhaustive search
- Work with prefixes

Join optimization (greedy search)

```
"considered_execution_plans": [
  {
    "plan_prefix": [],
    "table": "t1",
...
    "rows_for_plan": 1,
    "cost_for_plan": 2.7021,
    "rest_of_plan": [
      {
        "plan_prefix": ["t1"],
        "table": "t2",
...
        "rows_for_plan": 1,
        "cost_for_plan": 3.9021,
        "estimated_join_cardinality": 1
      }
...
```

```
  {
    "plan_prefix": [],
    "table": "t2",
...
    "rows_for_plan": 3,
    "cost_for_plan": 5.6064,
    "pruned_by_cost": true
  }
```

```
{
  "best_join_order": ["t1", "t2"]
},
```

Optimization steps

- Query transformations (non-cost-based identical, open way for other optimizations)
- Join optimization - preliminary phase (info collection)
- Join optimization (greedy search)
- Join optimization - plan refinement
- Other kinds of plan refinement

Join optimization - plan refinement

- Take apart WHERE/ON conditions and attach them to tables
  - Multiple-Equality Part #2: Take Item_equal(a,b,c) and generate pair like a=b, a=c.
  - Multiple-Equality Part #3: Equality Substitution.
- Index condition push down
- Join buffering
- ORDER BY … LIMIT Optimization

Take apart WHERE/ON conditions

```
  select * from t1,t2
    where t1.a=t2.a and t2.a>1 and
       t2.a < 5 and cos(t1.b) > 0 and sin(t2.b) < 0;
```


```
{
  "attaching_conditions_to_tables": {
    "original_condition": "t2.a = t1.a and t1.a > 1 and t1.a < 5 and cos(t1.b) > 0 and sin(t2.b) <
0",
    "attached_conditions_computation": [],
    "attached_conditions_summary": [
      {
        "table": "t1",
        "attached": "t1.a > 1 and t1.a < 5 and cos(t1.b) > 0"
      },
      {
        "table": "t2",
        "attached": "sin(t2.b) < 0"
      }
    ]
  }
}
```

Multiple-Equality Part #2

- Item_equal(a,b,c) can be:
  - a=b and b=c
  - a=c and c=b

  we choose that which will be calculated earlier

Multiple-Equality Part #3

- With equality substitution we can have limitations like for varchars: X=Y but LENGTH(X) <> LENGTH(Y)

ORDER BY … LIMIT Optimization

Check how to satisfy ORDER BY LIMIT:

- Are we using an index that produces data in the required ordering
- If yes: disable optimizations that “break ordering” (e.g. join buffer)
- If not: check if we can use a different index to produce data in the required ordering (and if this would be cheaper, but we can only change the index for the first table in the join)

Optimization steps

- Query transformations (non-cost-based identical, open way for other optimizations)
- Join optimization - preliminary phase (info collection)
- Join optimization (greedy search)
- Join optimization - plan refinement
- Other kinds of plan refinement

Other kinds of plan refinement

- Distinct, Window functions.
- Convert DISTINCT into GROUP BY if possible
- ...