MariaDB Fest 2021

Improving MariaDB's Query Optimizer with better selectivity estimates

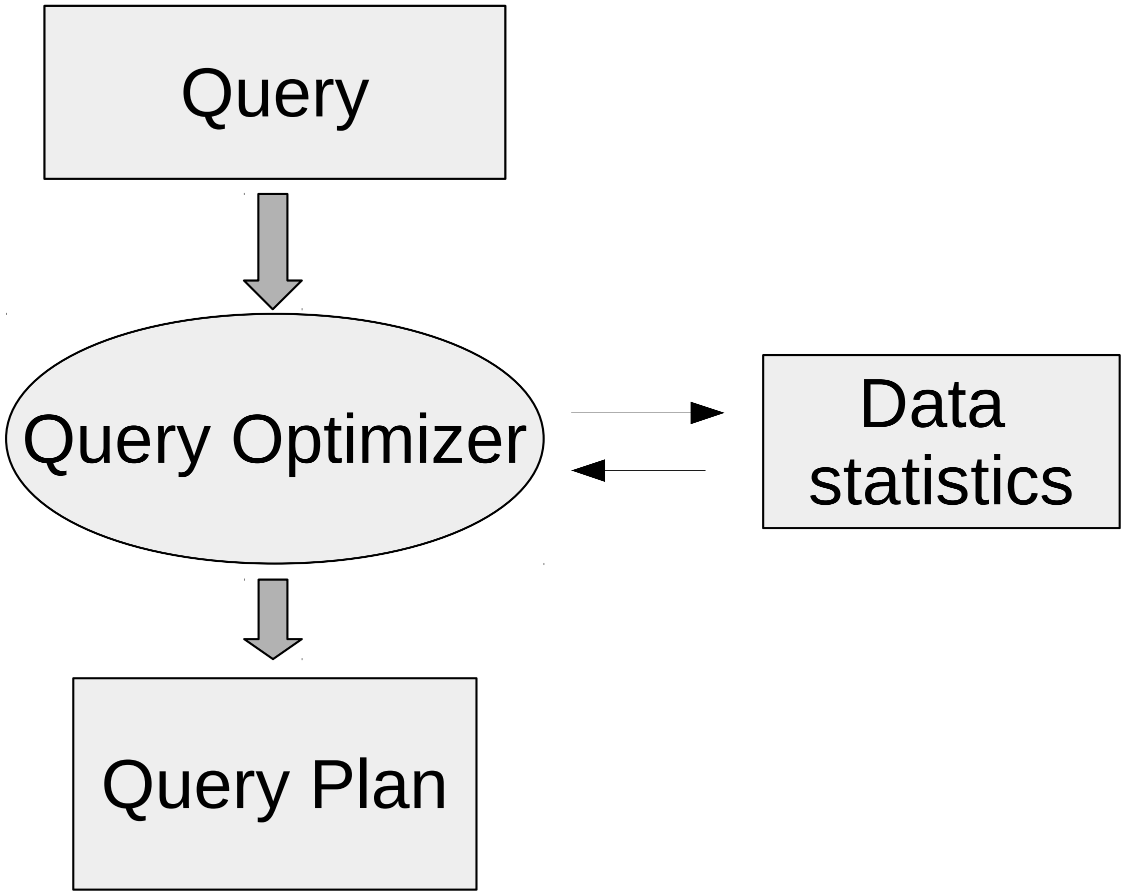
Sergei Petrunia
MariaDB developer

Background:

what are selectivity estimates

why they are important

Optimizer uses data statistics

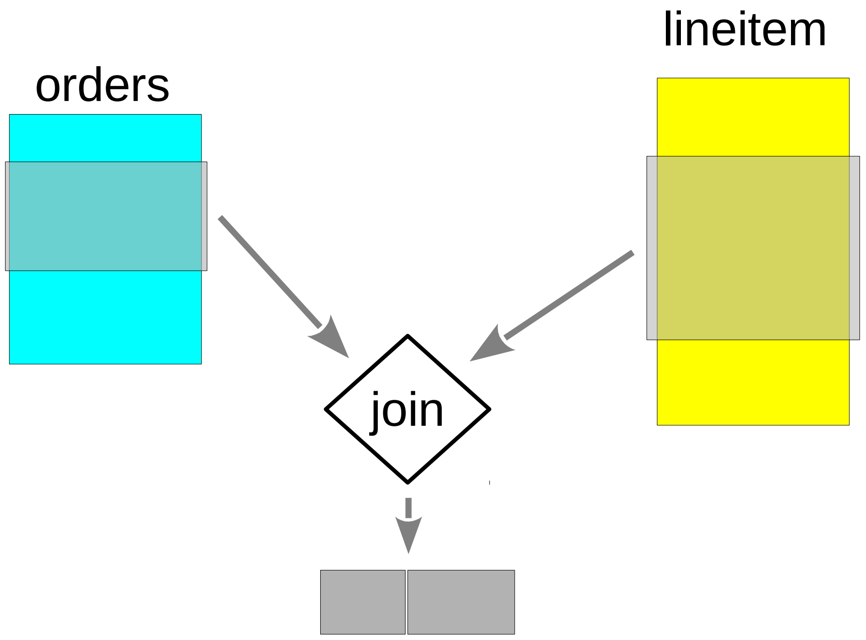


Cost-based query optimizer uses data statisistics:

- Cardinalities
- Selectivities
- Cost model
- ...

Cardinalities and selectivities

```
select *
from
  orders, lineitem
where
  o_orderkey=l_orderkey and
  o_orderdate between $DATE1 and $DATE2 and
  l_extendedprice > 1000
```



Cardinality is a number of rows

- Number of rows in the table
- Number of rows left after local condition check (the condition has “**selectivity**”)
- Number of rows after the join operation - ”join output cardinality”
- ...

Condition selectivity

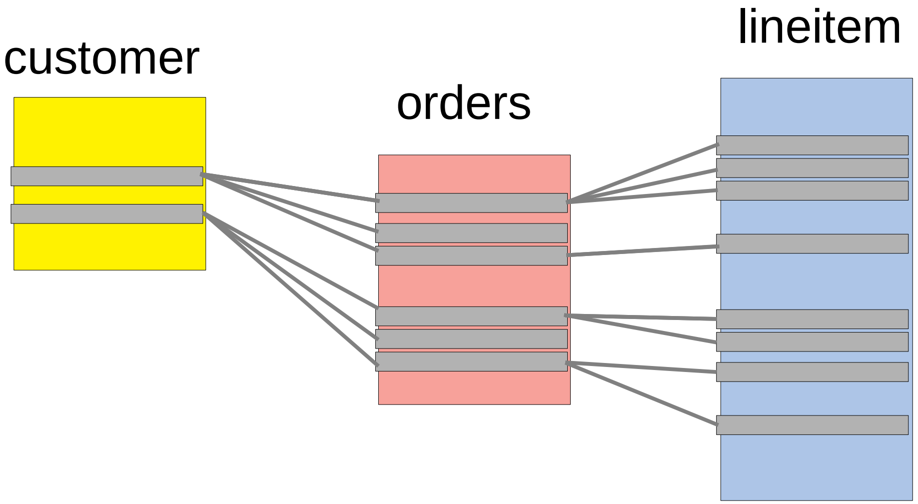
- Condition selectivity:

$$selectivity(cond) = \frac{rows\ satisfying\ cond}{total\ rows} * 100\%$$

- Selectivity
  - 0 (0%) = “no rows are accepted”
  - 1.0 (or 100%) = “all rows are accepted”
  - 0.33 (or 33%) = “one row in 3 is accepted”

Selectivity is important

```
select * from
  customer
  join orders on c_custkey=o_custkey
  join lineitem on o_orderkey=l_orderkey
where
  c_acctbal < 100 and
  o_shippriority=3 and
  l_extendedprice>1000
```



- Table access cost depends on the “incoming” cardinality
- Cardinality errors are **multiplied**, e.g.
  - 2x customers
  - 3x orders per customer
  - gives 6x lineitems
- Compare: errors in read costs are **added**.
- Wrong cardinality is a common cause of huge errors in estimates.

Computing selectivity

Computing selectivity

```
select *
from
  lineitem, orders
where
  o_orderkey=l_orderkey and
  o_orderdate between $DATE1 and $DATE2 and
  l_extendedprice > 1000000
```

- Local condition selectivity
  - Uses columns of one table
  - Typically, “column CMP const”
- Join condition selectivity

Local condition selectivity

Local condition selectivity

`o_orderdate between $DATE1 and $DATE2`

`o_shippriority=3`

`l_extendedprice > 1000000`

- Textbook:
  - “Guesstimates”
    - col=const: 10%, col < const: 50%
  - Histograms
    - or other pre-collected stats
  - Perform sampling
- MySQL/MariaDB: “records_in_range estimates”
  - Use an index as a histogram

Histograms

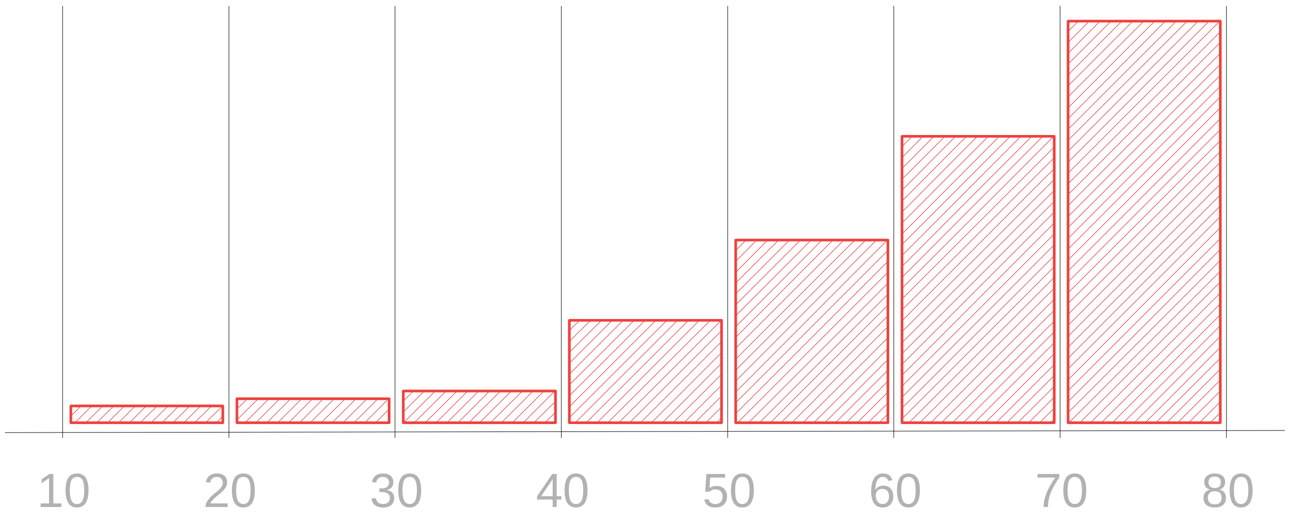
Basic Histogram

- “Value list” histogram
- List the values and their frequencies
- Works when n_values < n_buckets
- MySQL’s name: “singleton”

Value	**Cardinality**
‘foo’	100
‘bar’	200
‘baz’	300

Equi-width histogram

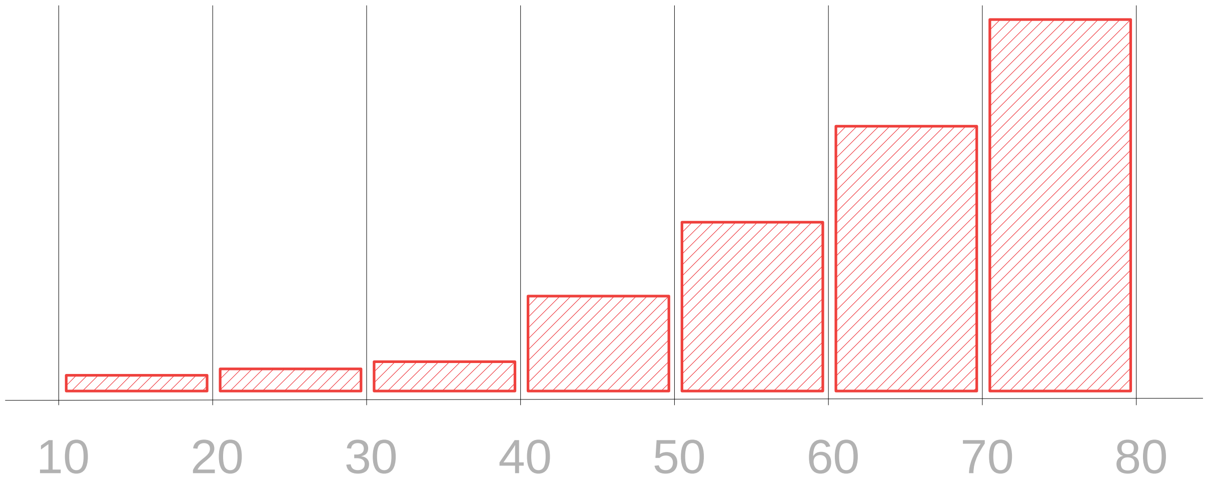
- Pre-defined bucket bounds
- Easy to collect



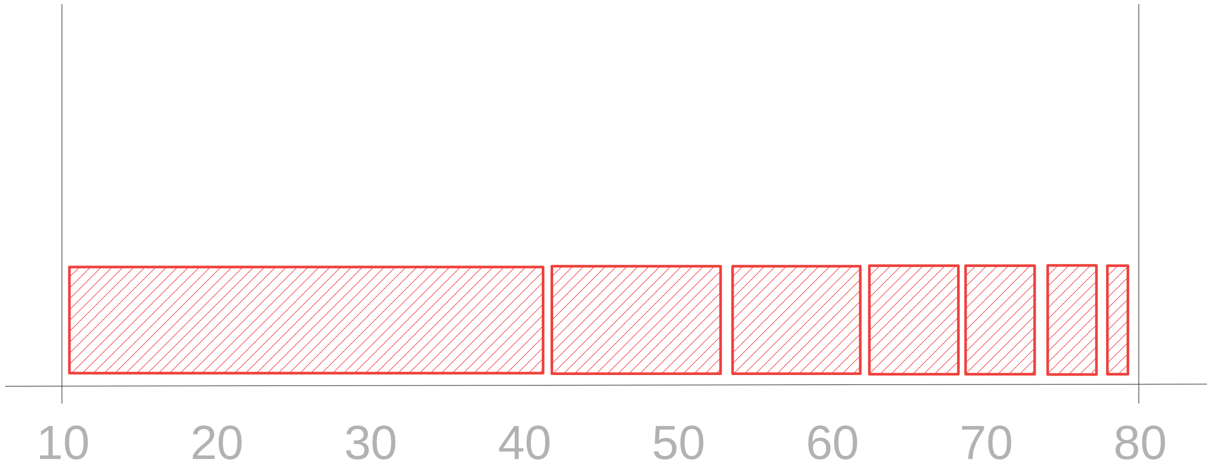
- Not always accurate:
  - A few “outlier” values in peripheral buckets
  - Most values are in a few very popular buckets
- What if densely-populated regions had more buckets?

Equi-height histogram

- Make buckets have the same number of rows
  - Densely populated areas get more buckets
  - Sparsely populated get less



- Better precision
- Can be collected with one pass.



Histogram usage

- MariaDB
  - 10.0-10.6: “Height-balanced” (aka equi-height)
  - 10.7: improved equi-height* with common values
- MySQL 8: equi-height* and “singleton”
- PostgreSQL: MostCommonValue list + equi-height
- CockroachDB: equi-height*
- TiDB: equi-height* + TopN

Histograms in MariaDB

- Introduced in MariaDB 10.0
- Not enabled by default
  - Require manual collection, update
  - Need to enable use by the optimizer
- Collection is expensive
  - Space consumption issues with VARCHAR(N>>)

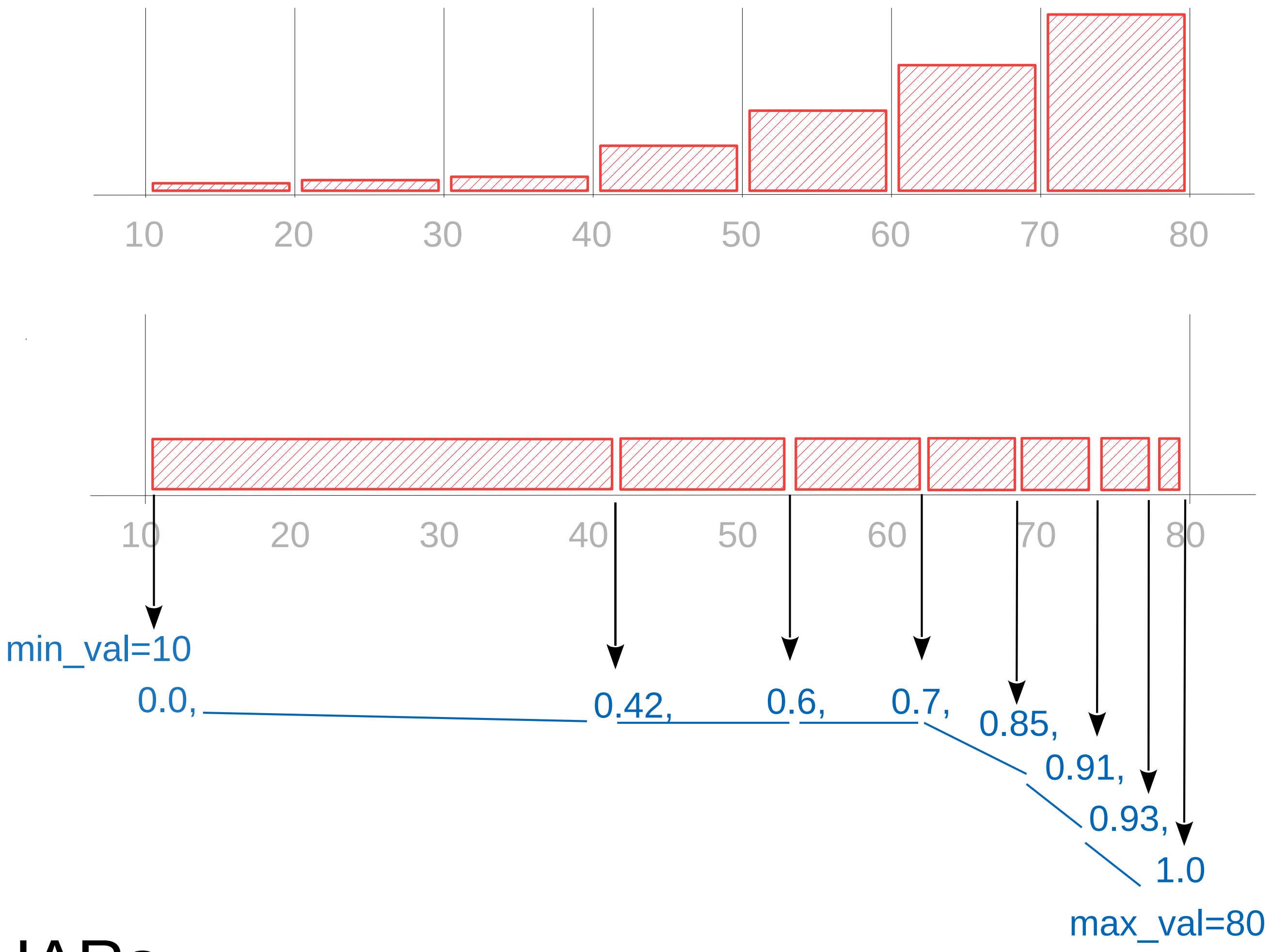
Histograms in MariaDB 10.4

- Closer to being on by default
  - ”The optimizer will use histograms if present”
  - Still, need non-default ANALYZE command to collect.

- Bernoulli sampling, @@analyze_sample_percentage
  - 100 (default): use all data
  - 0: pick the percentage automatically

Histograms in MariaDB 10.4: internals

- Height-balanced (=equi-depth)
- Store “fractions”
  - 0.0 is min_value
  - 1.0 is max_value
- Fixed-precision
  - SINGLE_PREC_HB - 1/256
  - DOUBLE_PREC_HB 1/64K
- Ok for ranges
- Poor for: Popular values, VARCHARs



Precision issue (MDEV-26125)

- 1M population, “country” matches real countries’ population.

“filtered” is the estimate

r_filted is the real value (observed)

```
analyze select * from generated_pop where country='China';
..-+---------+------------+----------+------------+-------------+
   | rows    | r_rows     | filtered | r_filtered | Extra       |
..-+---------+------------+----------+------------+-------------+
   | 1000000 | 1000000.00 |    22.66 |      21.04 | Using where |
..-+---------+------------+----------+------------+-------------+
```

Ok, for China it’s close

```
analyze select * from generated_pop where country='Chile';
..-+---------+------------+----------+------------+-------------+
   | rows    | r_rows     | filtered | r_filtered | Extra       |
..-+---------+------------+----------+------------+-------------+
   | 1000000 | 1000000.00 |    22.66 |       0.25 | Using where |
..-+---------+------------+----------+------------+-------------+
```

Very inaccurate for its neighbor Chile!

```
analyze select * from generated_pop where country='Sweden';
..-+---------+------------+----------+------------+-------------+
   | rows    | r_rows     | filtered | r_filtered | Extra       |
..-+---------+------------+----------+------------+-------------+
   | 1000000 | 1000000.00 |     4.69 |       0.15 | Using where |
..-+---------+------------+----------+------------+-------------+
```

Better for Sweden

Histograms in MariaDB 10.7

- Based on GSoC’21 project by Michael Okoko
- @@histogram_type=JSON_HB
- Stores values, not fractions
- Histogram is stored as JSON
- The histogram is height-balanced* (=equi-height)
  - Common values are in their own buckets

JSON_HB fixes MDEV-26125

- The same 1M population dataset, with JSON histogram:

```
set histogram_type=json_hb;
analyze table generated_pop persistent for all;

analyze select * from generated_pop where country='China';
..-+---------+------------+----------+------------+-------------+
   | rows    | r_rows     | filtered | r_filtered | Extra       |
..-+---------+------------+----------+------------+-------------+
   | 1000000 | 1000000.00 |    21.04 |      21.04 | Using where |
..-+---------+------------+----------+------------+-------------+

analyze select * from generated_pop where country='Chile';
..-+---------+------------+----------+------------+-------------+
   | rows    | r_rows     | filtered | r_filtered | Extra       |
..-+---------+------------+----------+------------+-------------+
   | 1000000 | 1000000.00 |     0.14 |       0.25 | Using where |
..-+---------+------------+----------+------------+-------------+

analyze select * from generated_pop where country='Sweden';
..-+---------+------------+----------+------------+-------------+
   | rows    | r_rows     | filtered | r_filtered | Extra       |
..-+---------+------------+----------+------------+-------------+
   | 1000000 | 1000000.00 |     0.17 |       0.15 | Using where |
..-+---------+------------+----------+------------+-------------+
```

Same as before for China

Much better for Chile

Also better for Sweden

JSON_HB under the hood

- A table of vehicles registered in an Australian state

```
{
  "histogram_hb_v2": [
    ...
    {
      "start": "C3",
      "size": 0.003936638,
      "ndv": 24
    },
    {
      "start": "CALAIS",
      "size": 0.002868234,
      "ndv": 10
    },
    {
      "start": "CAMRY",
      "size": 0.034154968,
      "ndv": 1
    },
    ...
```

- **start** – Start value
- **size** – fraction of table rows in the bucket
  - May vary
  - So, not really “equi-height”
  - The reason: “popular” values are in their own buckets
- **ndv**
  - Special case: ndv=1

MySQL’s equi-height histogram


```
{
  "buckets": [
    ...
    [
      "base64:type254:QlVHR1k=",
      "base64:type254:Q0FERFk=",
      0.14083891639965046,
      20
    ],
    [
      "base64:type254:Q0FMQUlT",
      "base64:type254:Q0FNQVJP",
      0.14234833037629427,
      2
    ],
    [
      "base64:type254:Q0FNUlk=",
      "base64:type254:Q0FNUlk=",
      0.1812091200381 3255,
      1
    ],
```


- Bucket is a JSON array with
  - **min**, **max**
  - **cumulative_fraction**
  - **ndv**
- Buckets may have different sizes
  - Popular values in their own buckets
  - “Each value should be in one bucket” (and not two adjacent buckets)(?)
  - https://bugs.mysql.com/bug.php?id=104789
- There are “holes” between buckets

MySQL's equi-height histogram

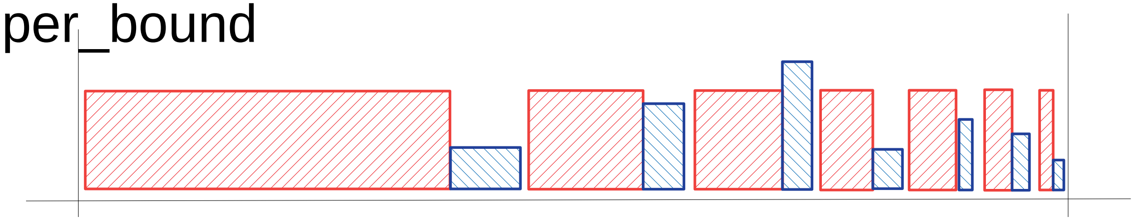
```
{
  "buckets": [
    ...
    [
      "BUGGY",
      "CADDY",
      0.14083891639965046,
      20
    ],
    [
      "CALAIS",
      "CAMARO",
      0.1423483303762942 7,
      2
    ],
    [
      "CAMRY",
      "CAMRY",
      0.1812091200381325 5,
      1
    ],
```

- Bucket is a JSON array with
  - **min**, **max**
  - **cumulative_fraction**
  - **ndv**
- Buckets may have different sizes
  - Popular values in their own buckets
  - "Each value should be in one bucket" (and not two adjacent buckets)(?)
  - https://bugs.mysql.com/bug.php?id=104789
- There are "holes" between buckets

CockroachDB’s equi-height histogram

upper_bound	range_rows	distinct_range_rows	equal_rows
...			
'CAMRY'	306	4.2680661910276	25189
'CAPTIVA'	1760	24.548354627732	5971
'CARRY'	3368	46.9766239587613	306
'CERATO'	1760	24.548354627732	4517
'CHEROKEE'	2067	28.8303686825296	1607
'CIVIC'	459	6.40209928654141	7962
...			

- Just the **upper_bound**
- **range_rows** is number of rows between prev and this bound, exclusive.
  - Not exactly equi-height, either.
  - “A value goes into one bucket”
- **equal_rows** is how many rows were equal to the upper_bound
  - Every bucket has has a “singleton” co-bucket
    - reason for this?



TiDB’s equi-height histogram

```
+-----------+-------+---------+--------------------+----------------------+------+
| Bucket_id | Count | Repeats | Lower_Bound        | Upper_Bound          | Ndv  |
+-----------+-------+---------+--------------------+----------------------+------+
...
|        60 | 22893 |     226 | CABSTAR            | CALIFORNIA           |    0 |
|        61 | 23212 |     226 | CAMROAD            | CAPRI                |    0 |
|        62 | 23714 |     271 | CARAVAN (IMPORT)   | CELERIO              |    0 |
|        63 | 24079 |     136 | CELICA (IMPORT)    | CF SERIES            |    0 |
|        64 | 24489 |     181 | CHARADE CENTRO     | CHASER               |    0 |
|        65 | 24808 |      45 | CHEVELLE           | CITY COUPE           | [illegible]
```

```
SHOW STATS TOP_N;
+--------------------+-------+
| Value              | Count |
+--------------------+-------+
| CAMRY              | 24507 |
| CAPTIVA            |  5019 |
...
```

- **TOP_N** values are stored separately
- **Lower_bound** and **Upper_bound** (holes)
- **Count** is cumulative number of values (Not exactly equi-height, again)
- **Repeats** is the occurence number of the Upper_bound
  - Again, every other bucket is a singleton
- **ndv** is only present with newer collection algorithm

PostgreSQL’s equi-height histogram

```
most_common_vals          | {HILUX,COMMODORE,LANDCRUISER,... }
most_common_freqs         | {0.052233335,0.0415,0.0379,... }
histogram_bounds          | {100,125I,208,"300 SERIES 616",307,318I,
320I,323,323I,335CI,407,"4 RUNNER",530I,86,9-Mar,A200,A4,A5,
ALMERA,AVALON,B180,B3000,BEETLE,BRERA,"C180 SERIES",...
```

- **most_common_vals** (100 of them) are stored separately
- The histogram just stores the bucket bounds
  - A real equi-height histogram.

On the number of buckets

- PostgreSQL: default_statistics_target=100
  - 100 MCVs + 100 buckets
- MySQL: the default number of buckets is 100
- CockroachDB: 200
  - for non-indexed cols, just 2. This means they only get min/max?
- TiDB – varied, 50..250?
- MariaDB: histogram_size=254
  - SINGLE_PREC_HB: 254, DOUBLE_PREC_HB: 127 buckets.

Histogram benchmark?

- Can't have it – apples to apples comparison is hard
- Sampling vs full-scan collection
- Histogram size
  - Do MCV/TopN count as buckets?
  - Bucket with two bounds vs bucket with one
  - ...
- Adequate precision is enough

Histograms take-aways

- MariaDB 10.7 is getting new histogram_type=**JSON_HB**
- Provides histograms similar to other databases
  - Height-balanced*, common values are included
  - Make them the default in the next release?

- Things that are still missing:
  1. “Genuine sampling”. Collection should not be a full table scan.
  2. Automatic re-collection.

Selectivity for multiple conditions

Combining multiple conditions

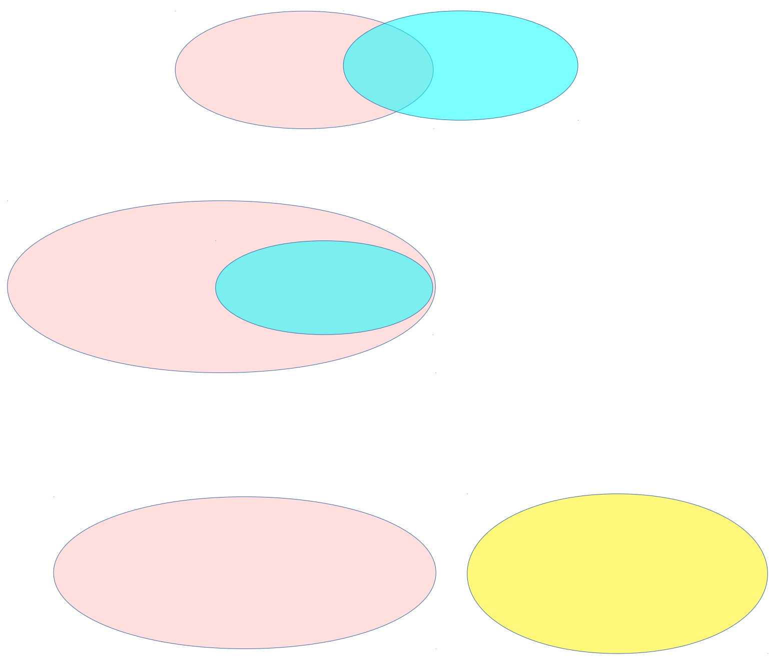
sel1 sel2

'swimsuit'

```
select ...
from order_items
where shipdate='2021-12-15' AND item_name='christmas light'
```

Combined selectivity

- if independent: sel1*sel2
- If dependent:
  - “Worst” case: MIN(sel1, sel2)
  - “Best” case: 0.0



Solution #1: use a certain assumption

`shipdate >= '2021-12-15'` AND `shipdate <= '2021-12-24'` AND `item_name='doll'`

- Combine conditions on the same column
- Textbook: assume conditions are independent

  *sel=sel1*sel2*

- Conservative: use the most selective

  sel= MIN(*sel1*, *sel2*)

Solution #1: use a certain assumption

`shipdate >= '2021-12-15'` AND `shipdate <= '2021-12-24'` AND `item_name='doll'`

- "Exponential back-off" (SQL Server 2014)
  - order conditions by selectivity, most selective first: s_1 s_2 s_3 s_4 ...
  - Then,

$$sel = s_1 \cdot s_2^{\frac{1}{2}} \cdot s_3^{\frac{1}{4}} \cdot s_4^{\frac{1}{8}}$$

  - Assume s_1 ,"half" of s_2, "quarter" of s_3, ...

Solution #2:

Multi-Column statistics

Multi-column stats

- Multi-column histogram would take a lot of space
  - Typically not used
    - TiDB seems to support them (but see TiDB issue 22589)?
- PostgreSQL
  - 10: Functional dependency
  - 12 (Oct,2019): Multivariate MCV lists
- MySQL, MariaDB: **records_in_range**.

MariaDB, MySQL: records-in-range

```
INDEX idx(col1, col2, col3, ...)
WHERE col1=... AND col2<=... AND ...col3...
```

The optimizer

- Builds a list of ranges over {col1, col2, col3 ...}
  - (this is complex topic)
  - In MariaDB, check the optimizer trace (not in MySQL: Bug #95824)
- Uses the index as a large histogram
  - The call to query the index:
    records_in_range({c1,c2,c3} <= {col1,col2,col3} <= {...}**)**

records_in_range in action

- Using vehicle registration database again

```
alter table vehicle_reg add index (Make, Model);
```

- Try a frequently occurring combination:

```
explain select * from vehicle_reg where Make='FORD' and Model='FOCUS';
+------+-------------+-------------+------+---------------+------+---------+-------------+------+-------------
| id   | select_type | table       | type | possible_keys | key  | key_len | ref         | rows | Extra
+------+-------------+-------------+------+---------------+------+---------+-------------+------+-------------
|    1 | SIMPLE      | vehicle_reg | ref  | Make          | Make | 390     | const,const | 8638 | Using index
+------+-------------+-------------+------+---------------+------+---------+-------------+------+-------------
```

- Try a combination that doesn't exist:

```
explain select * from vehicle_reg where Make='MITSUBISHI' and Model='FOCUS';
+------+-------------+-------------+------+---------------+------+---------+-------------+------+-------------
| id   | select_type | table       | type | possible_keys | key  | key_len | ref         | rows | Extra
+------+-------------+-------------+------+---------------+------+---------+-------------+------+-------------
|    1 | SIMPLE      | vehicle_reg | ref  | Make          | Make | 390     | const,const | 1    | Using index
+------+-------------+-------------+------+---------------+------+---------+-------------+------+-------------
```

- Good estimates!

PostgreSQL, for comparison

- INDEX(Make, Model), no multi-variate statistics

```
test=# explain select * from vehicle_reg where Make='FORD' and Model='FOCUS';
                                    QUERY PLAN
------------------------------------------------------------------------------------------
 Bitmap Heap Scan on vehicle_reg  (cost=8.35..1354.94 rows=383 width=125)
   Recheck Cond: (((make)::text = 'FORD'::text) AND ((model)::text = 'FOCUS'::text))
   ->  Bitmap Index Scan on make_model  (cost=0.00..8.25 rows=383 width=0)
         Index Cond: (((make)::text = 'FORD'::text) AND ((model)::text = 'FOCUS'::text))
```

- 383 rows is "sel1 * sel2"

```
test=# explain select * from vehicle_reg where Make='MITSUBISHI' and Model='FOCUS';
                                    QUERY PLAN
------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on vehicle_reg  (cost=7.98..1237.73 rows=347 width=125)
   Recheck Cond: (((make)::text = 'MITSUBISHI'::text) AND ((model)::text = 'FOCUS'::text))
   ->  Bitmap Index Scan on make_model  (cost=0.00..7.90 rows=347 width=0)
         Index Cond: (((make)::text = 'MITSUBISHI'::text) AND ((model)::text = 'FOCUS'::text))
```

- Doesn't see the difference!
  - 347/383 is a count(Mitsubishi)/count(Ford)

Multivariate stats

```
CREATE STATISTICS make_model_stat (mcv) ON Make, Model FROM vehicle_reg;
analyze vehicle_reg;

test=# explain select * from vehicle_reg where Make='FORD' and Model='FOCUS';
                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Bitmap Heap Scan on vehicle_reg  (cost=69.86..10465.53 rows=4823 width=124)
   Recheck Cond: (((make)::text = 'FORD'::text) AND ((model)::text = 'FOCUS'::text))
   ->  Bitmap Index Scan on make_model  (cost=0.00..68.66 rows=4823 width=0)
         Index Cond: (((make)::text = 'FORD'::text) AND ((model)::text = 'FOCUS'::text))
```

- **rows=4823** is a much better estimate. **Ford-Focus** is in MCV list

```
test=# explain select * from vehicle_reg where Make='MITSUBISHI' and Model='FOCUS';
                                        QUERY PLAN
---------------------------------------------------------------------------------------------
 Bitmap Heap Scan on vehicle_reg  (cost=8.38..1364.94 rows=386 width=124)
   Recheck Cond: (((make)::text = 'MITSUBISHI'::text) AND ((model)::text = 'FOCUS'::text))
   ->  Bitmap Index Scan on make_model  (cost=0.00..8.29 rows=386 width=0)
         Index Cond: (((make)::text = 'MITSUBISHI'::text) AND ((model)::text = 'FOCUS'::text))
```

- rows=386, the same as before. No help if the combination is not in MCV list.

Multiple conditions selectivity takeaways

- Basic: combine per-column statistics
  - Assume independence/overlap/etc
  - Very imprecise (no information)

- Advanced: multi-column statistics
  - Much better
  - Much harder than single-column stats
  - MariaDB/MySQL has a non-conventional form: **records_in_range**

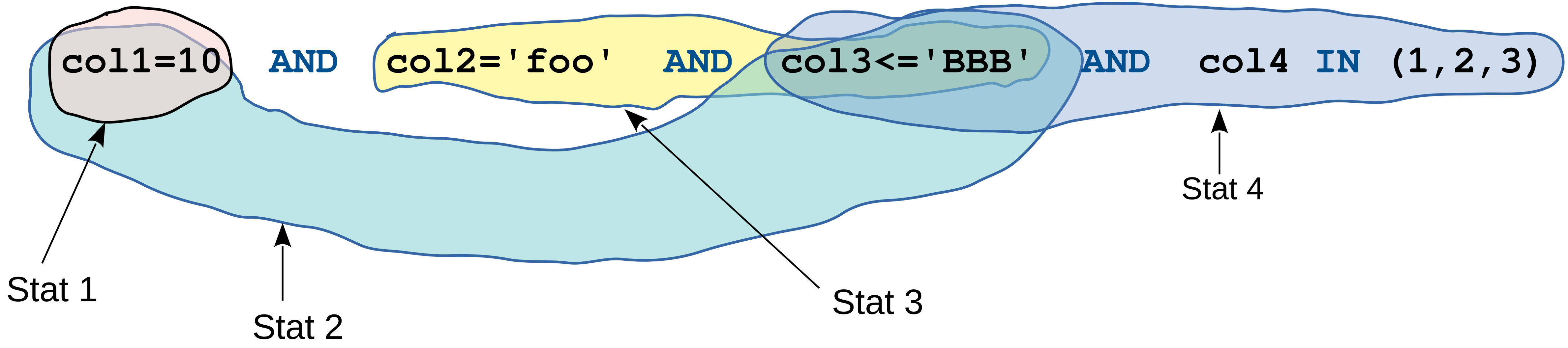
Combining

multi-column selectivities

Combining multi-column statistics

```
col1=10  AND  col2='foo'  AND  col3<='BBB'  AND  col4 IN (1,2,3)
```

Combining multi-column statistics



- Multi-column statistics can “overlap”
- How to combine them?
  - Don’t want to assume independence...

The right way – research papers

- VLDB‘2005: V. Markl et. al, **"Consistently Estimating the Selectivity of Conjuncts of Predicates"**
  - Introduces “Maximum Entropy” principle
  - Makes use of all available information
- EBDT’2020, D Havenstein et. al. **“Fast Entropy Maximization for Selectivity Estimation of Conjunctive Predicates on CPUs and GPUs”**
  - New, faster entropy maximization algorithm
- Hard!
  - Not used in production(?)

What is used in production?

- Prefer one statistic to other
- Some heuristic rule
  - “Statistics that uses more columns goes first”
  - “Minimize the number of independence assumptions we have to make”
  - “Pick the most selective stats first”
  - ...

In MariaDB

- Reconciling overlapping estimates since 2006!
  - e.g. sql_select.cc, grep for ReuseRangeEstimateForRef
- The issue got worse in 10.4
  - Optimizer tries to account for selectivity more agressively
  - => More “collisions”
- MDEV-23707 and linked tasks: MDEV-21813, MDEV-25830, ...
  - Need to fix these
  - Working on this
- Workaround for now: set optimizer_use_condition_selectivity=1

Not covered in this talk:

Join Condition Selectivity

Takeaways

- Computing condition selectivity is
  - important
  - hard
- MariaDB 10.7: JSON_HB histograms
  - more precise for common values, VARCHARs.
- Work is underway to improve selectivity computations in the optimizer.

Thanks for your attention!