Adaptive query optimization in PostgreSQL

Oleg Ivanov

Postgres Professional

postgrespro.ru

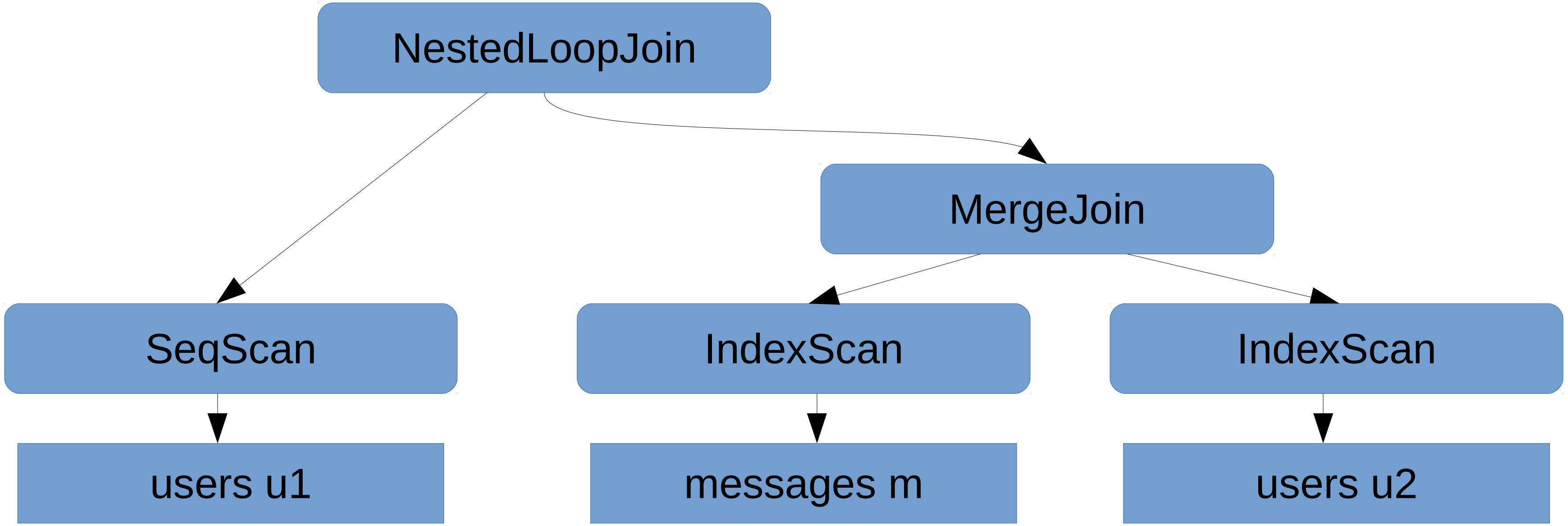
Outline

- Problem statement
  - Query optimization
  - Correlated clauses issue
- Machine learning
  - Gradient K Nearest Neighbours method
- Adaptive query optimization
  - Theory
  - Implementation
- Experimental evaluation

Query optimization

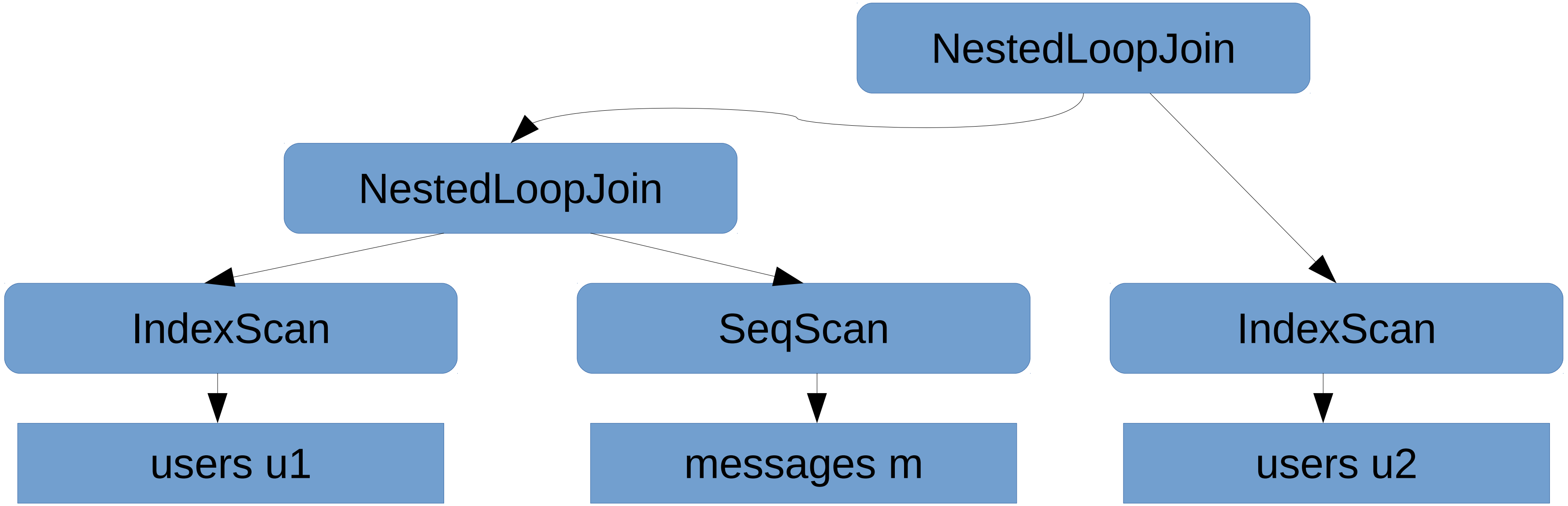
Query plan

```
SELECT *
FROM users AS u1, messages AS m, users AS u2
WHERE u1.id = m.sender_id AND m.receiver_id = u2.id;
```



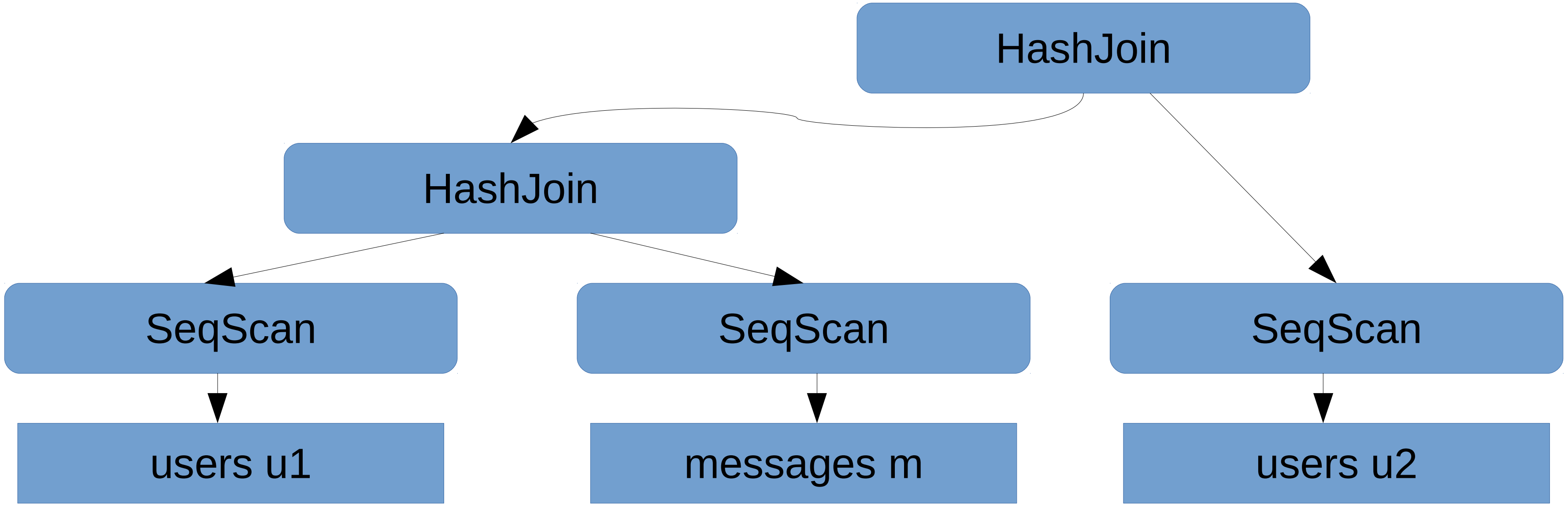
Query plan

```
SELECT *
FROM users AS u1, messages AS m, users AS u2
WHERE u1.id = m.sender_id AND m.receiver_id = u2.id;
```



Query plan

```
SELECT *
FROM users AS u1, messages AS m, users AS u2
WHERE u1.id = m.sender_id AND m.receiver_id = u2.id;
```



Query plan

```
EXPLAIN SELECT *
FROM users AS u1, messages AS m, users AS u2
WHERE u1.id = m.sender_id AND m.receiver_id = u2.id;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Hash Join  (cost=540.00..439429.44 rows=10003825 width=27)
   Hash Cond: (m.receiver_id = u2.id)
   ->  Hash Join  (cost=270.00..301606.84 rows=10003825 width=23)
         Hash Cond: (m.sender_id = u1.id)
         ->  Seq Scan on messages m  (cost=0.00..163784.25 rows=10003825 width=19)
         ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
               ->  Seq Scan on users u1  (cost=0.00..145.00 rows=10000 width=4)
   ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
         ->  Seq Scan on users u2  (cost=0.00..145.00 rows=10000 width=4)
(9 rows)
```

Query plan

Plan execution cost

```
EXPLAIN SELECT *
FROM users AS u1, messages AS m, users AS u2
WHERE u1.id = m.sender_id AND m.receiver_id = u2.id;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Hash Join  (cost=540.00..439429.44 rows=10003825 width=27)
   Hash Cond: (m.receiver_id = u2.id)
   ->  Hash Join  (cost=270.00..301606.84 rows=10003825 width=23)
         Hash Cond: (m.sender_id = u1.id)
         ->  Seq Scan on messages m  (cost=0.00..163784.25 rows=10003825 width=19)
         ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
               ->  Seq Scan on users u1  (cost=0.00..145.00 rows=10000 width=4)
   ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
         ->  Seq Scan on users u2  (cost=0.00..145.00 rows=10000 width=4)
(9 rows)
```

Plan node execution cost

Plan node cardinality

PROFESSIONAL Postgres

Cost-based query optimization

System R

1. A function which determines the plan's cost
2. Minimizing the function value over all possible plans for the query

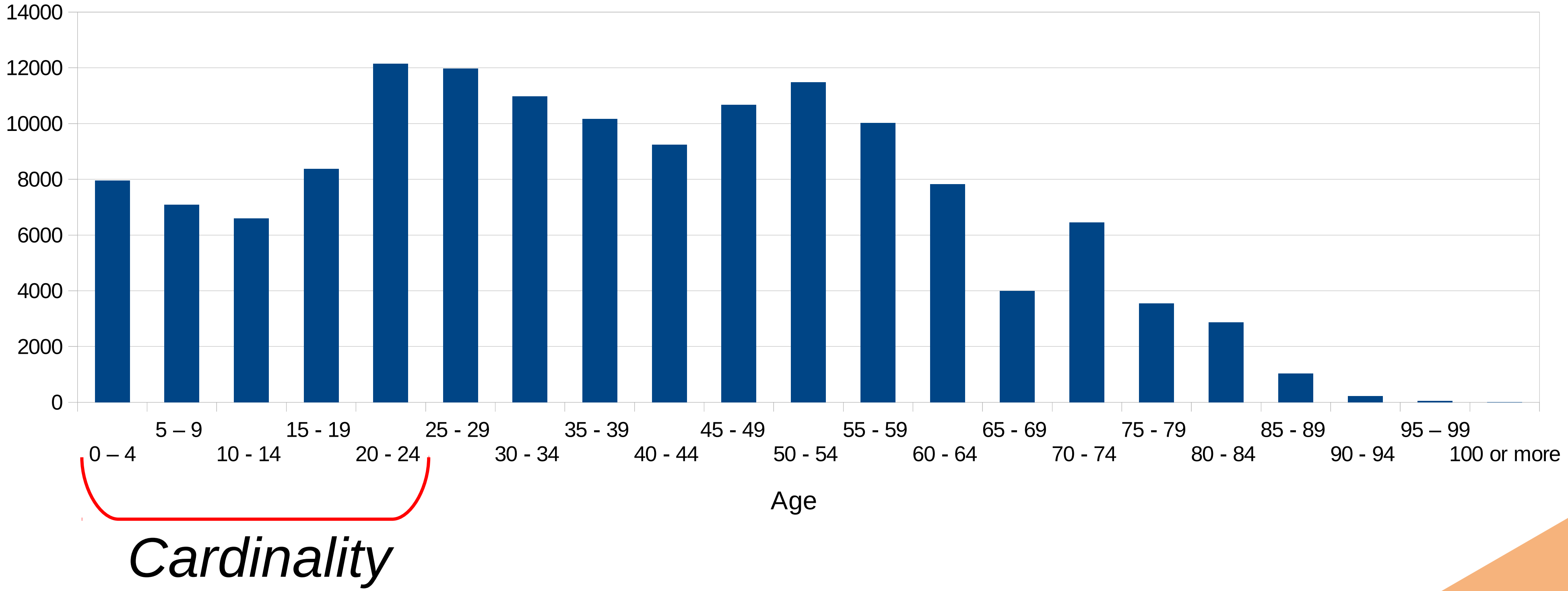
PostgreSQL cost model

$$Cost = n_s c_s + n_r c_r + n_t c_t + n_i c_i + n_o c_o$$

c_s	seq_page_cost	1.0
c_r	random_page_cost	4.0
c_t	cpu_Tuple_cost	0.01
c_i	cpu_Index_tuple_cost	0.005
c_o	cpu_Operator_cost	0.0025

Cardinality estimation

```
SELECT * FROM users
WHERE age < 25;
```



Dynamic programming over subsets

- System R
- Time complexity: 3^n
- Memory consumption: 2^n
- Always finds the cheapest plan

Genetic algorithm

- PostgreSQL
- Common and flexible method
- Can be stopped on every iteration
- No guarantees

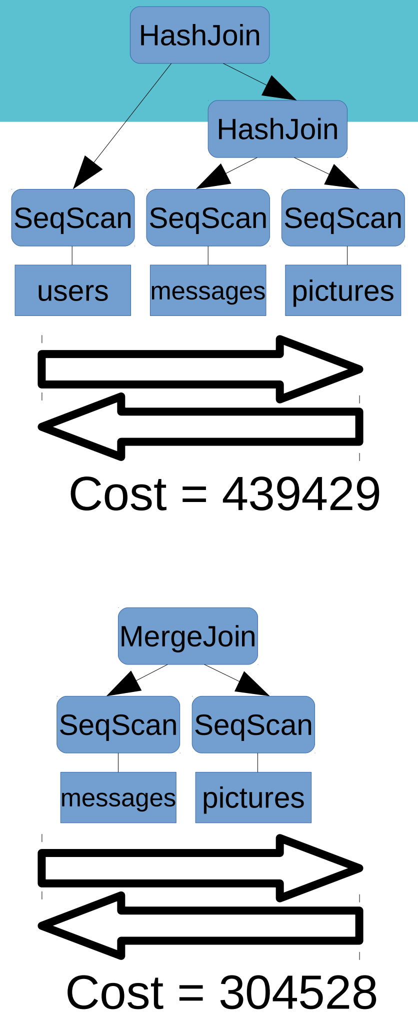
PostgreSQL query optimization

Optimization method

Dynamic programming

or

Genetic algorithm

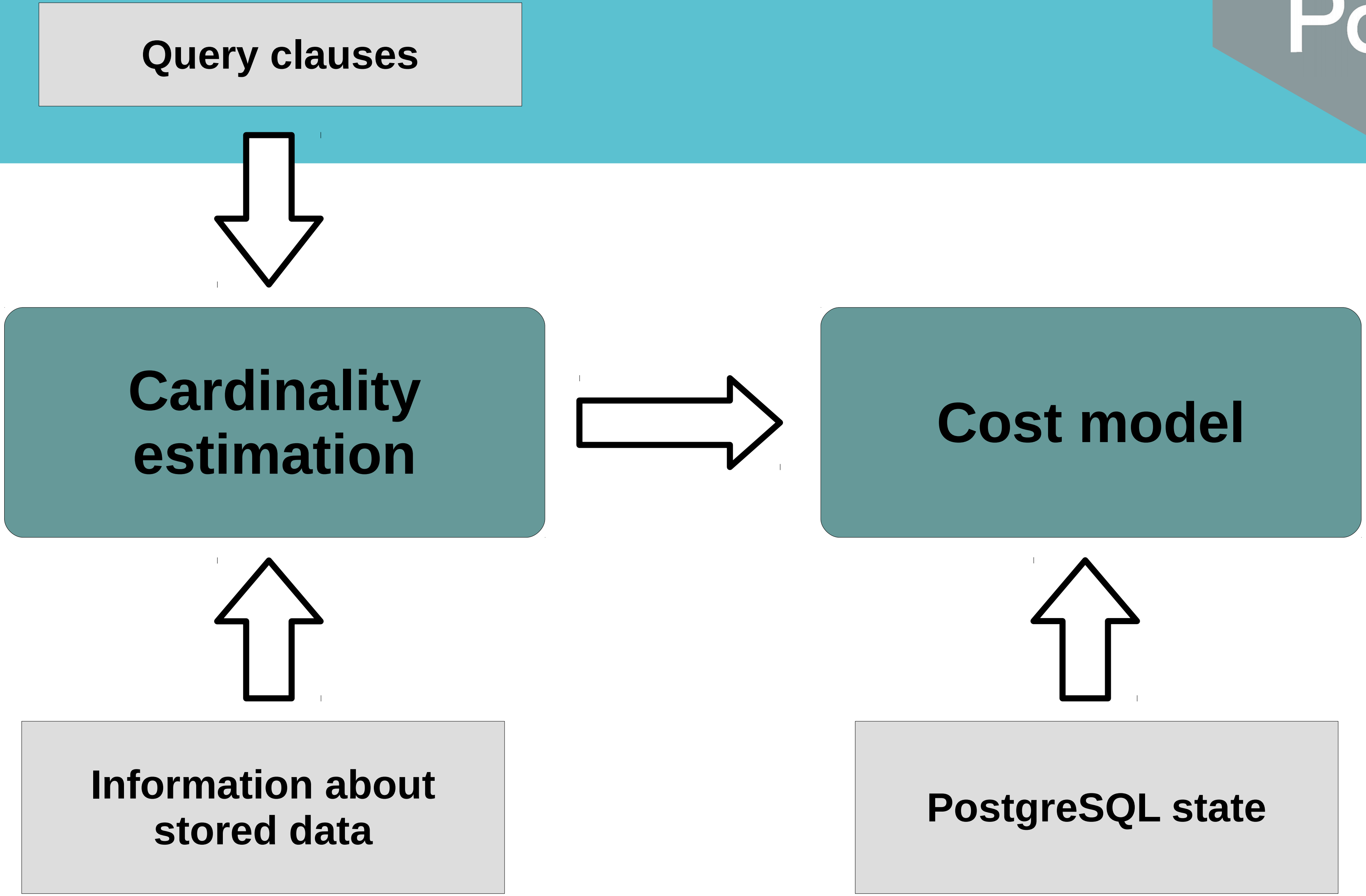


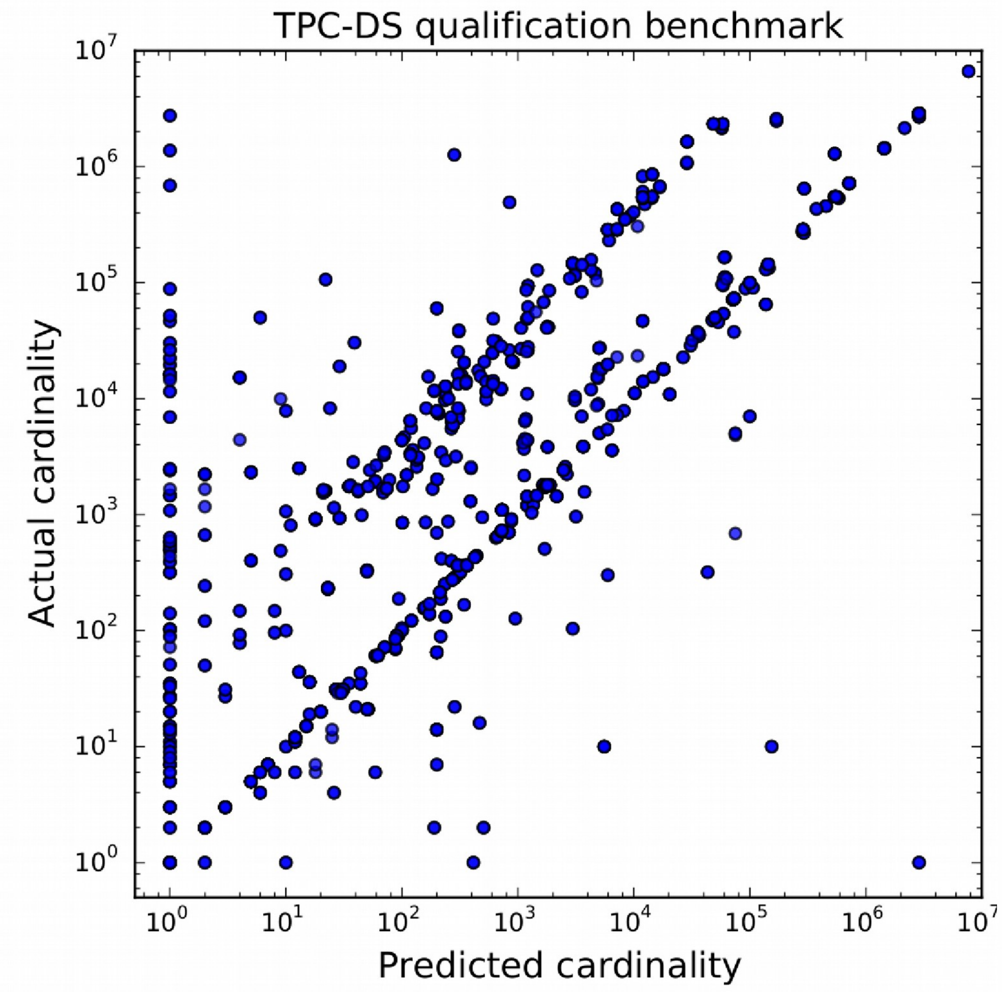
Plan's cost estimation

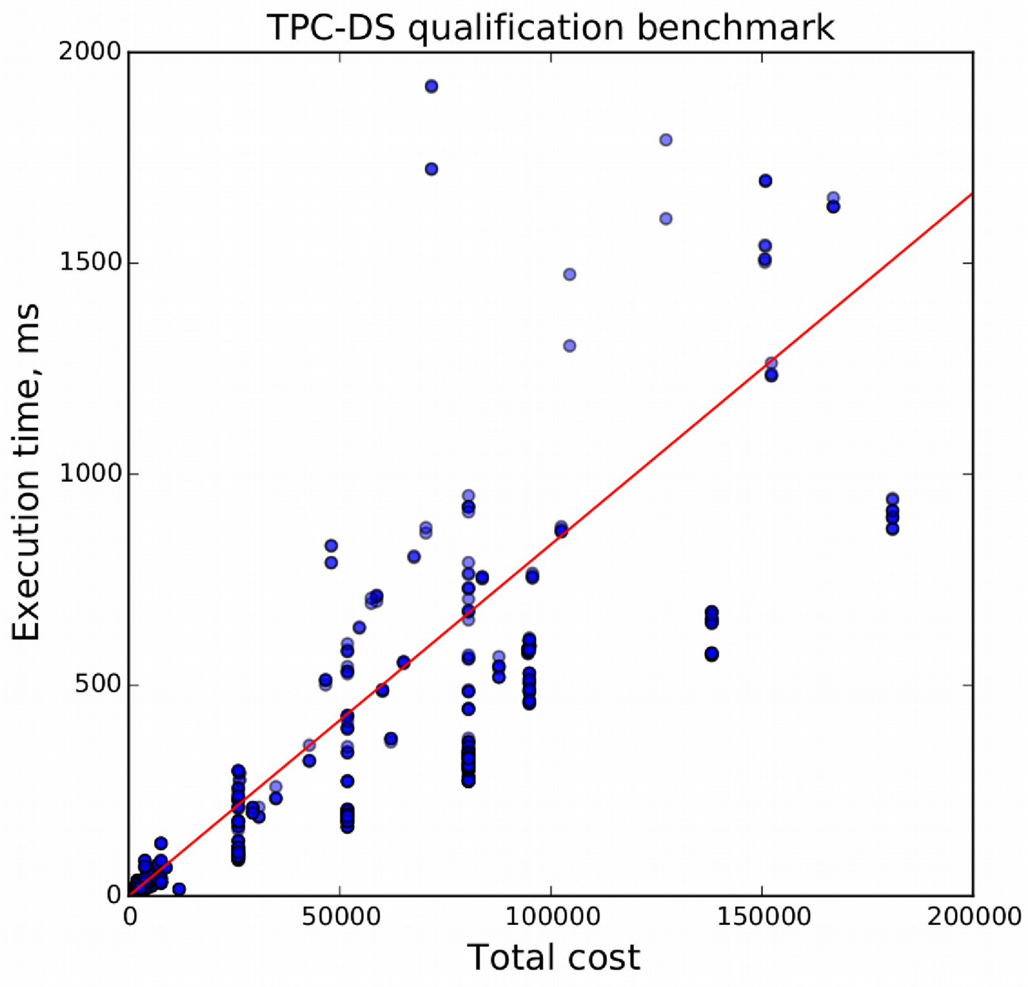
Cardinality estimation

Cost model

Correlated clauses issue


Query clauses
Cardinality estimation
Cost model
Information about stored data
PostgreSQL state

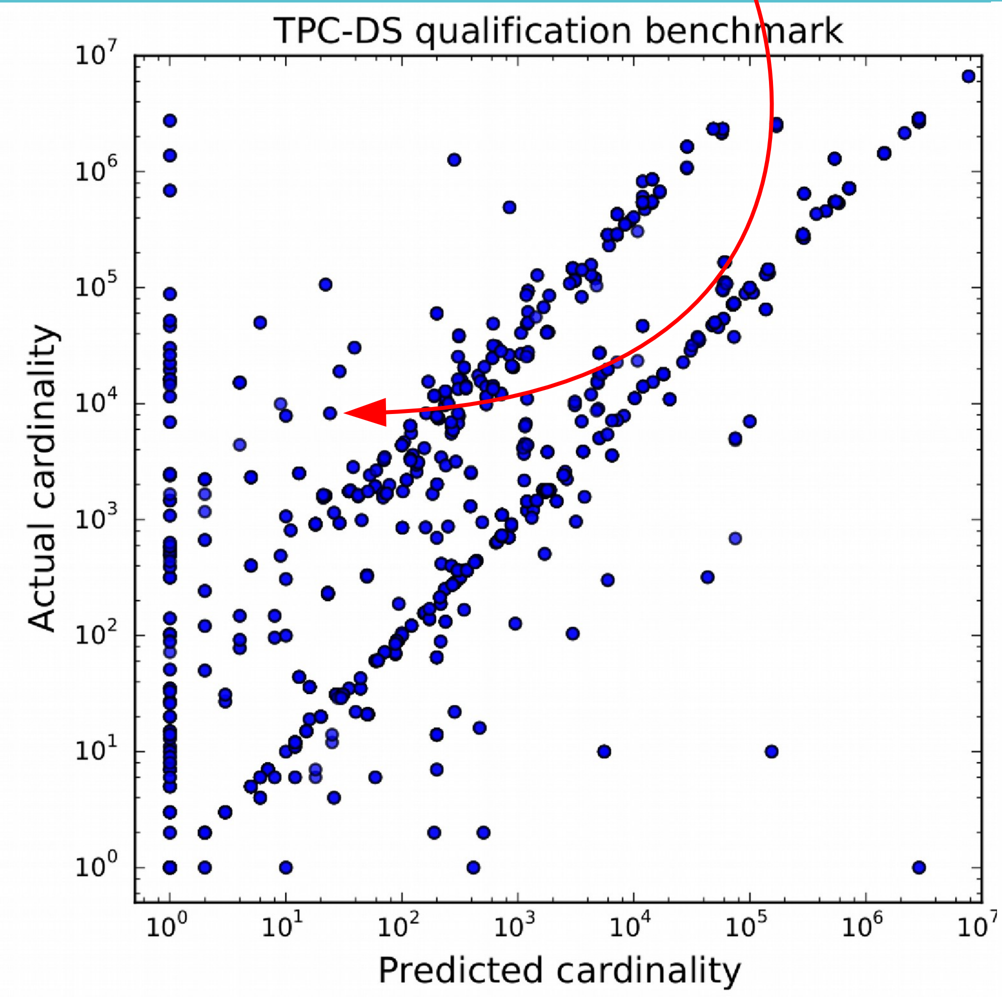


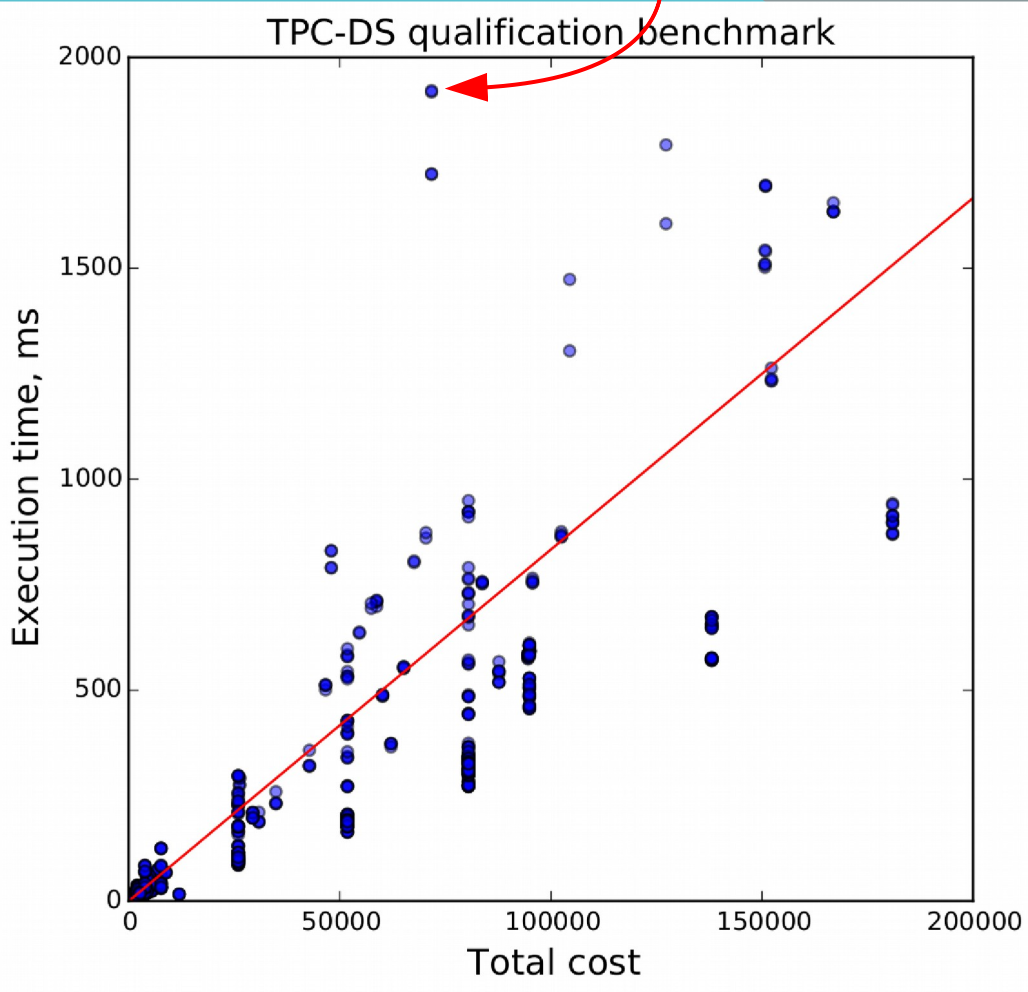


Dataset:
The TPC Benchmark™DS (TPC-DS)
http://www.tpc.org/tpcds/

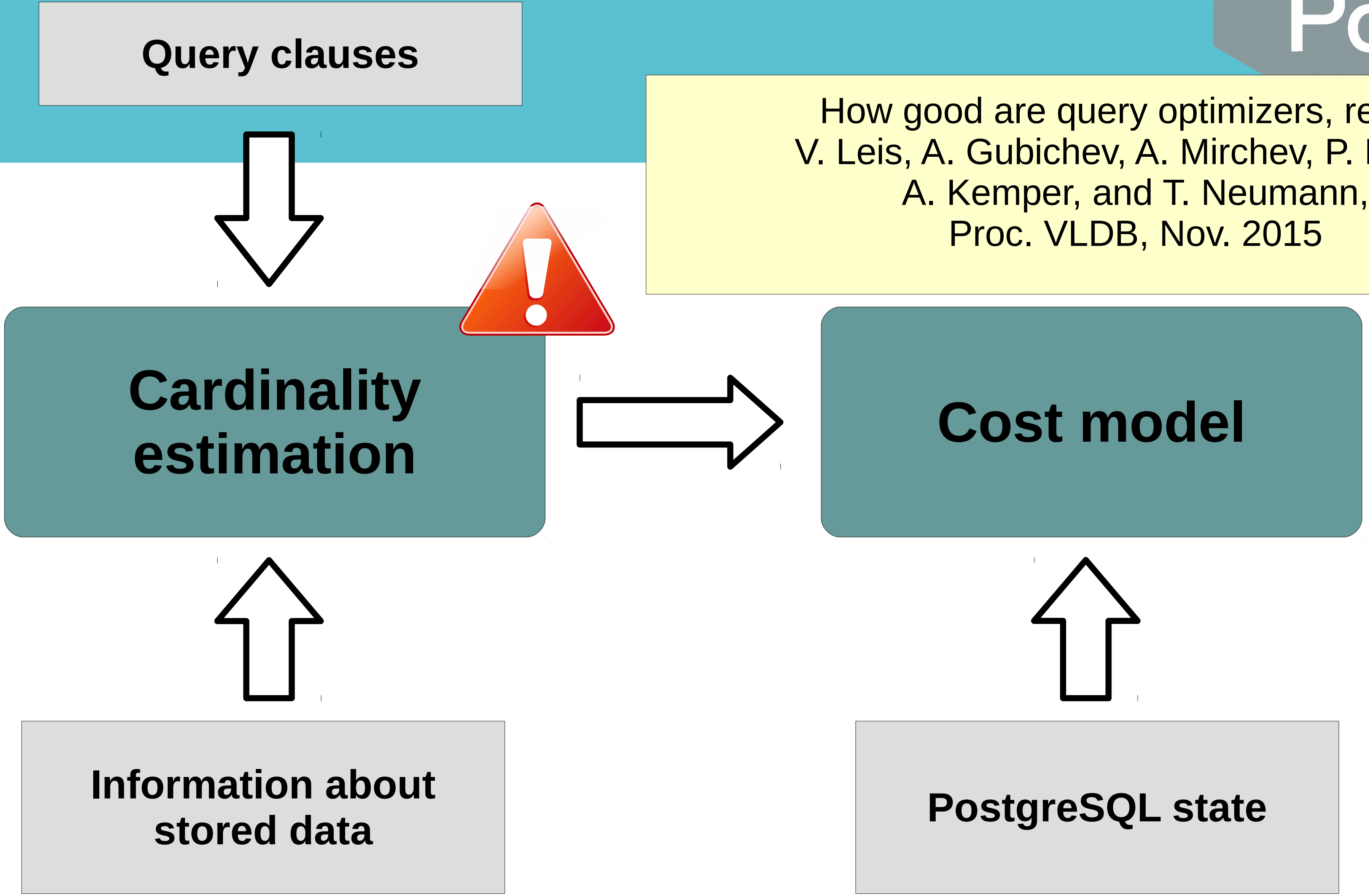
Error: 300 times

Error: 4 times



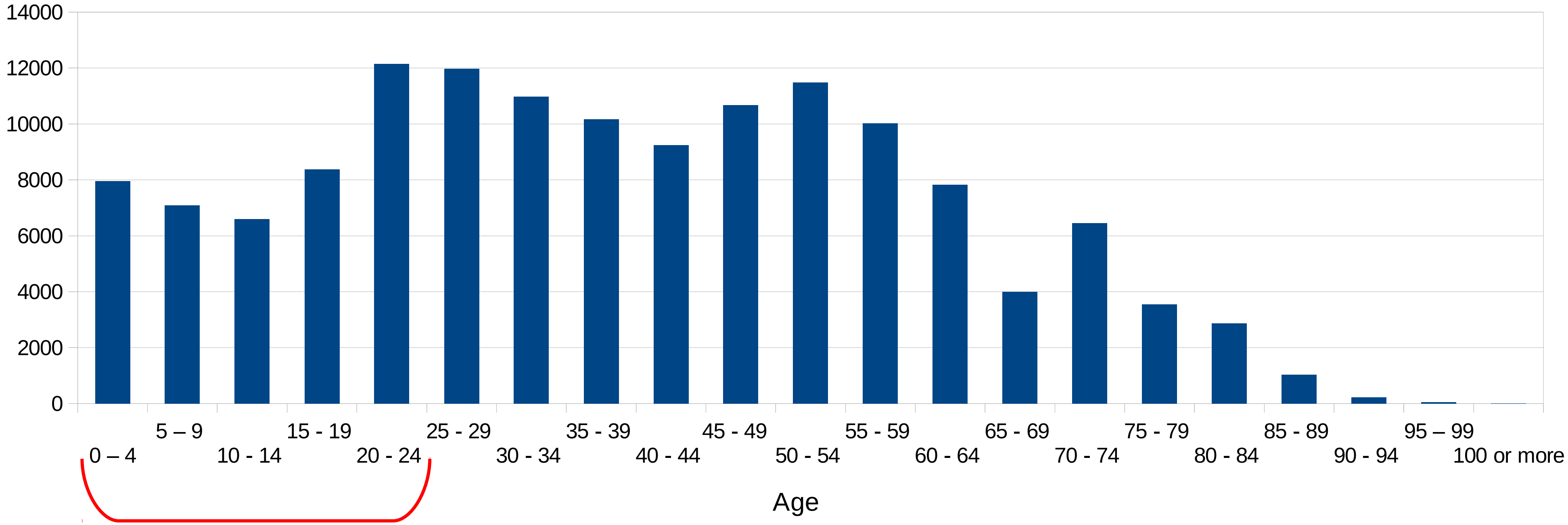


Dataset:
The TPC Benchmark™DS (TPC-DS)
http://www.tpc.org/tpcds/



How good are query optimizers, really?
V. Leis, A. Gubichev, A. Mirchev, P. Boncz,
A. Kemper, and T. Neumann,
Proc. VLDB, Nov. 2015

```sql
SELECT * FROM users
WHERE age < 25;
```



$$Selectivity \simeq 0.3$$

$$Cardinality = N_{tuples} \cdot Selectivity$$

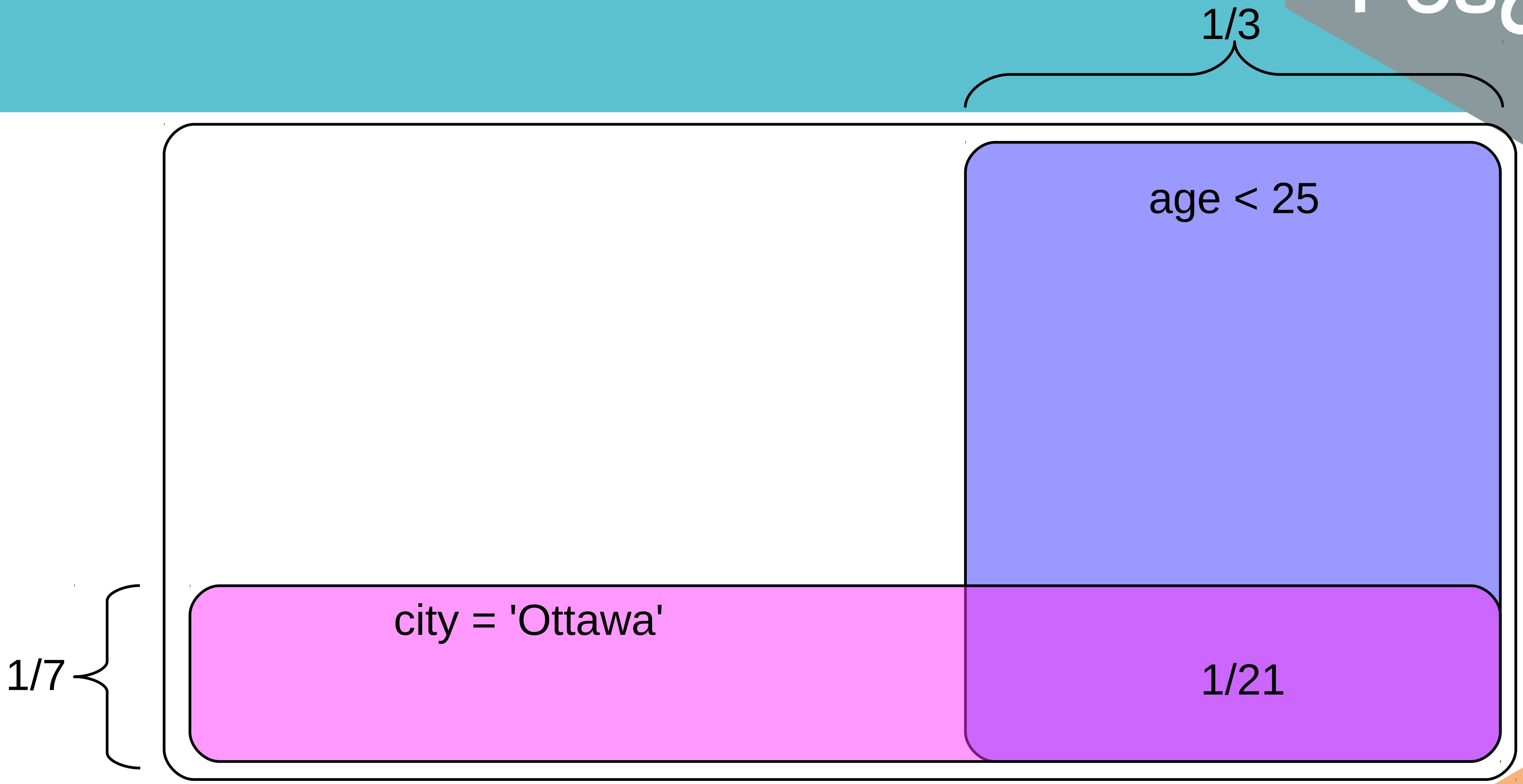
```
SELECT * FROM users
WHERE age < 25 AND city = 'Ottawa';
```

Only selectivities of individual clauses
(i.e. *marginal* selectivities)
are known

$$Selectivity_{age} = \frac{1}{3}$$

$$Selectivity_{city} = \frac{1}{7}$$

$$Selectivity_{age,city} = ?$$


1/3
age < 25
city = 'Ottawa'
1/7
1/21

```
SELECT * FROM users
WHERE age < 25 AND city = 'Ottawa';
```

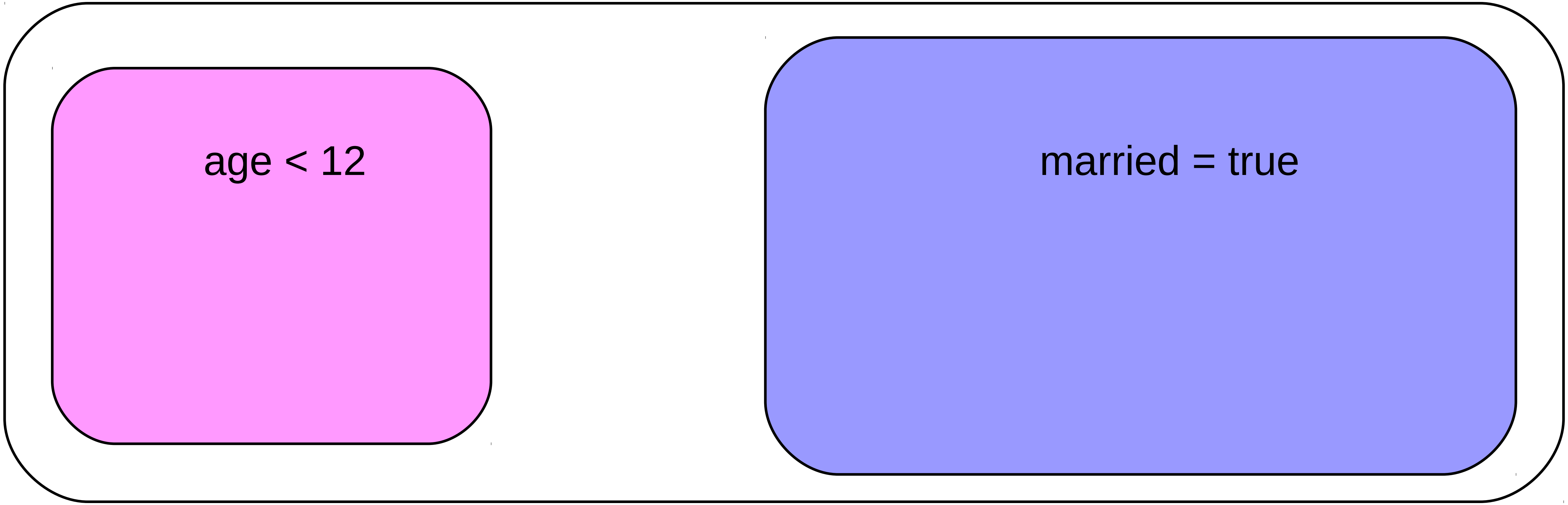
Only selectivities of individual clauses are known

The clauses are considered to be independent:

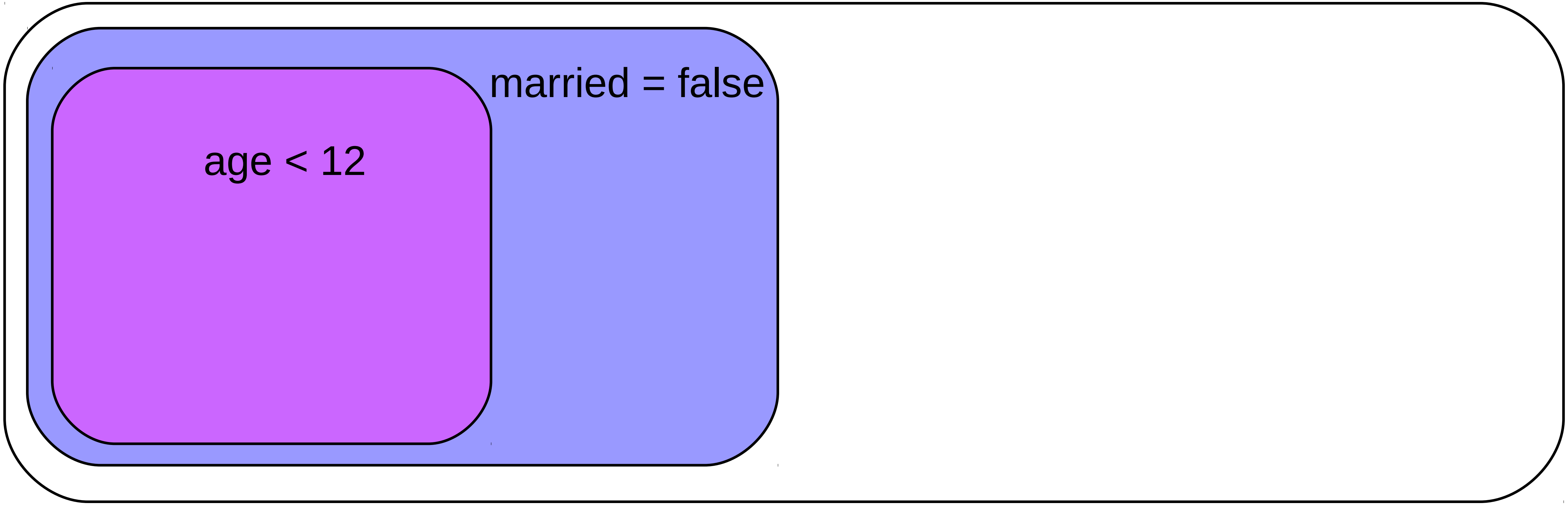
$$Selectivity_{age,city} = Selectivity_{age} \cdot Selectivity_{city}$$

With the only following exception $Selectivity_{25<age \text{ AND } age<57} = Selectivity_{25<age<57}$

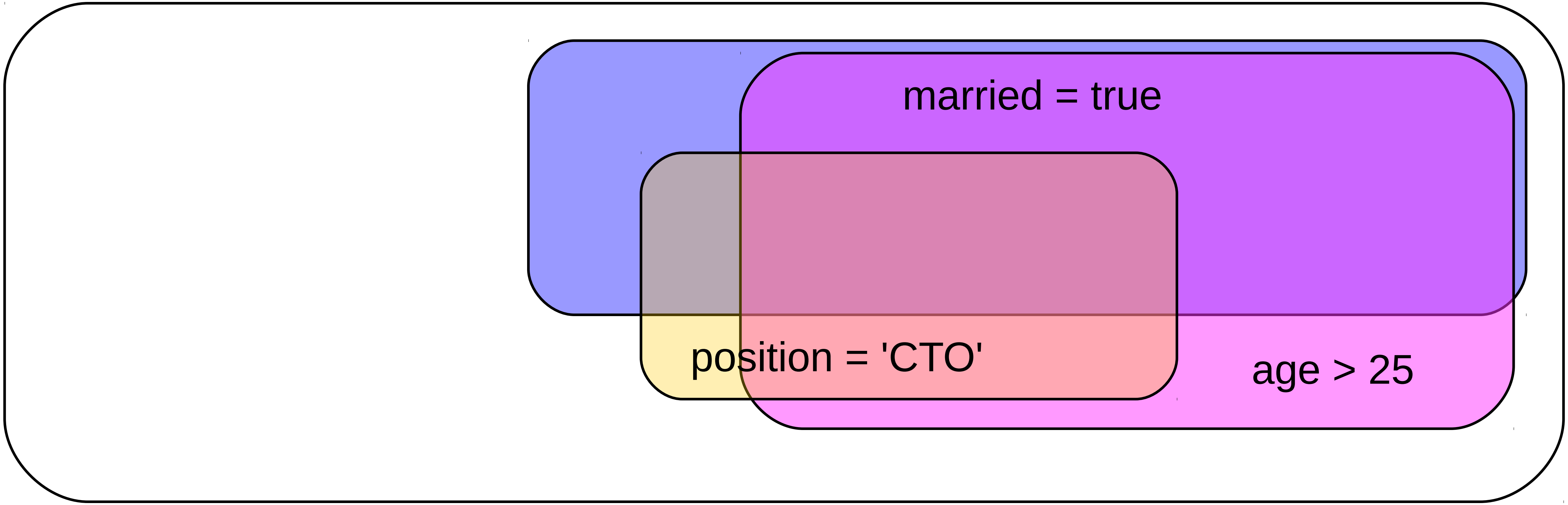
```sql
SELECT * FROM users
WHERE age < 12 AND married = true;
```



```sql
SELECT * FROM users
WHERE age < 12 AND married = false;
```



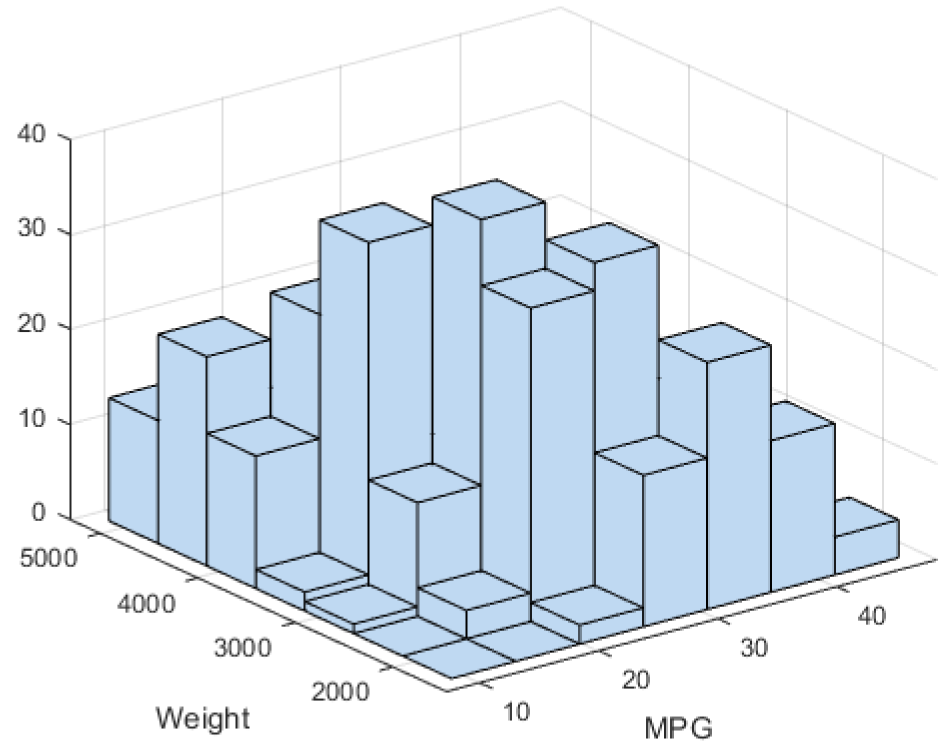
```
SELECT * FROM users
WHERE age > 25 AND married = true
AND position = 'CTO';
```



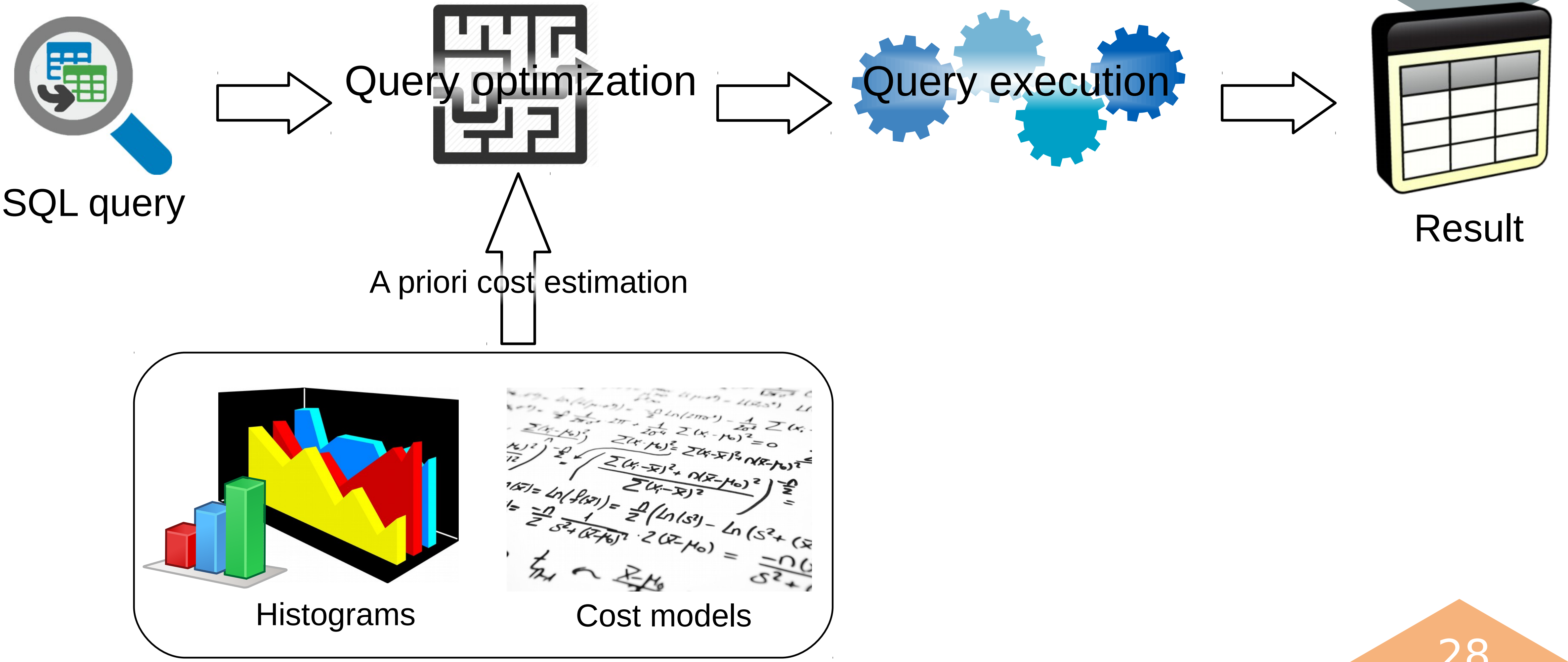
Multidimensional histograms

Pros:
- Solve the problem
- Have theoretical guarantees

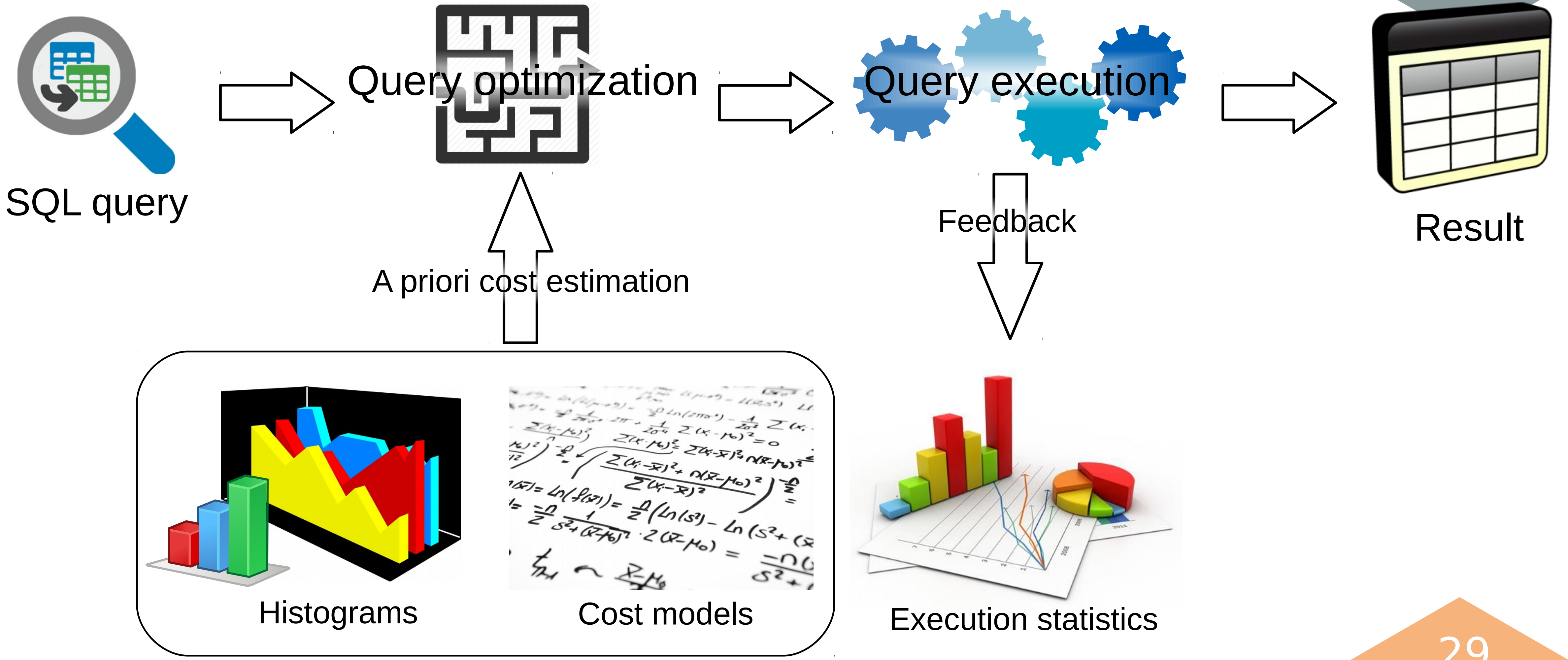
Contras:
- Dimensionality curse
- Require memory
- Require time for building or updating
- Not clear which of all possible column subsets are needed
- Correlation tests are slow



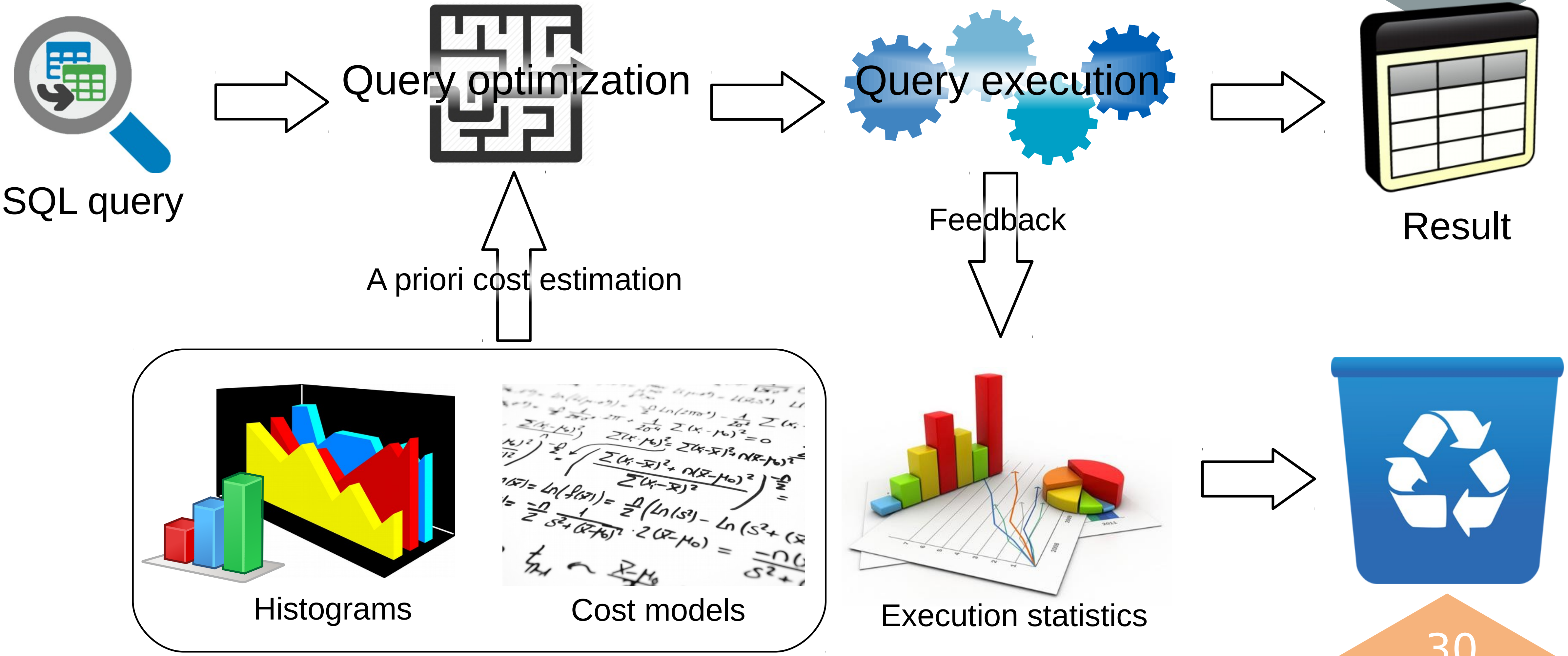
Adaptive query optimization: idea



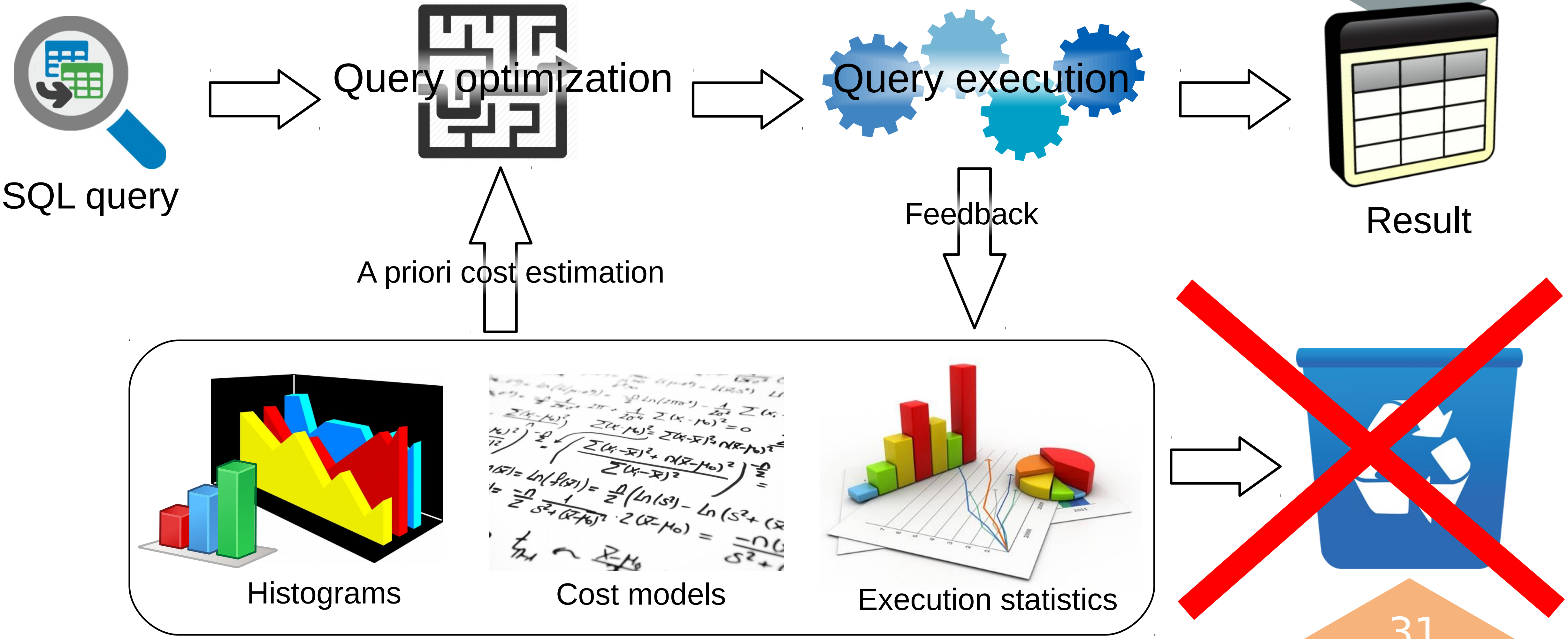
Adaptive query optimization: idea



Adaptive query optimization: idea

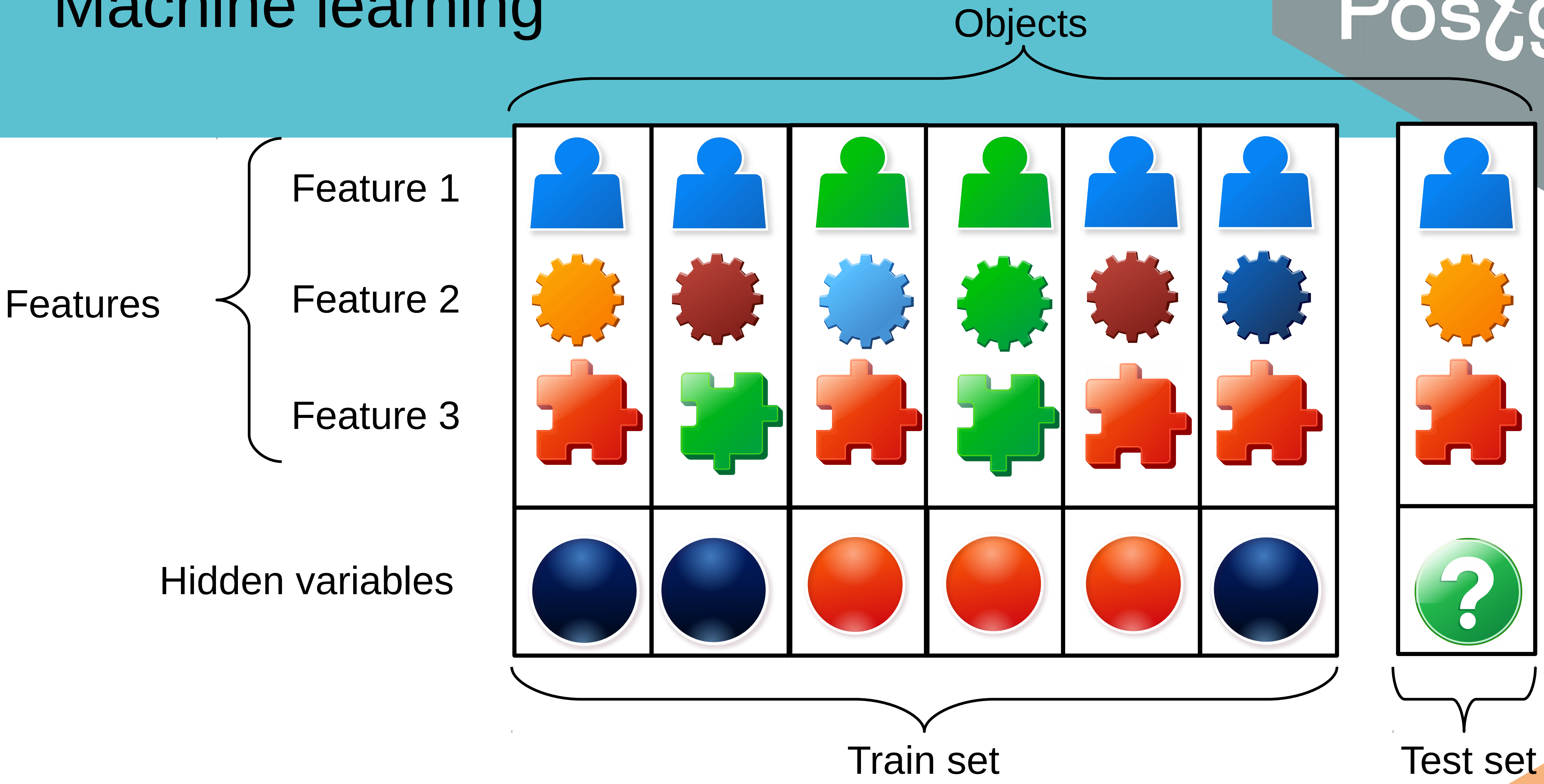


Adaptive query optimization: idea



Machine learning

Machine learning



Learning procedure

For our problem the learning workflow is iterative:

- On the planning stage the model has to predict hidden variables for a number of objects
- After the execution stage some of these objects are appended to the train set and the model can learn on them

Learning procedure

Query 1

Train set:
Empty

Planning stage:
Object 1 - ?
Object 2 - ?
Object 3 - ?

After-execution stage:
Object 1 – Variable 1
Object 3 – Variable 3

Learning procedure

Query 2

Train set:

- Object 1 – Variable 1
- Object 3 – Variable 3

Planning stage:

- Object 4 - ?
- Object 1 - ?
- Object 5 - ?

After-execution stage:

- Object 4 – Variable 4

Learning procedure

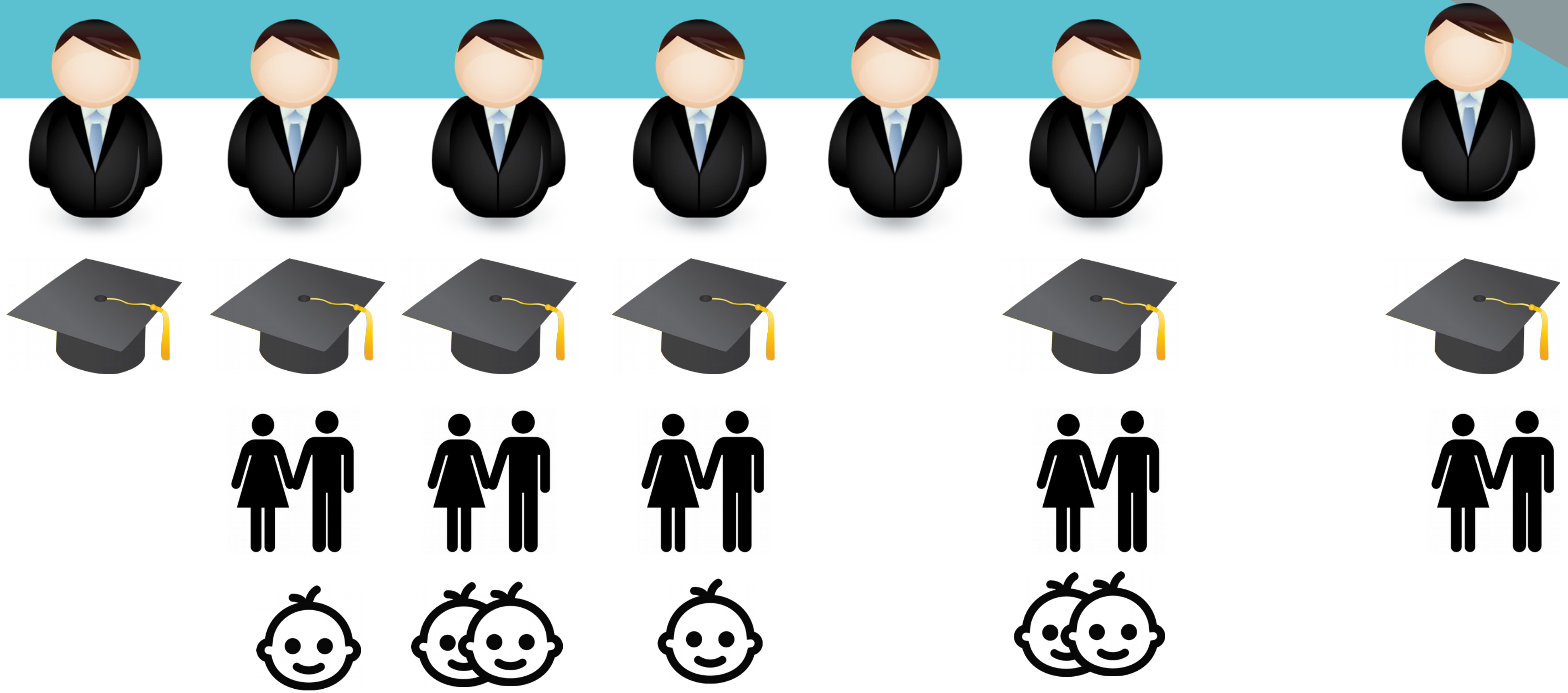
Query 3

Train set:
- Object 1 – Variable 1
- Object 3 – Variable 3
- Object 4 – Variable 4

Planning stage:
...

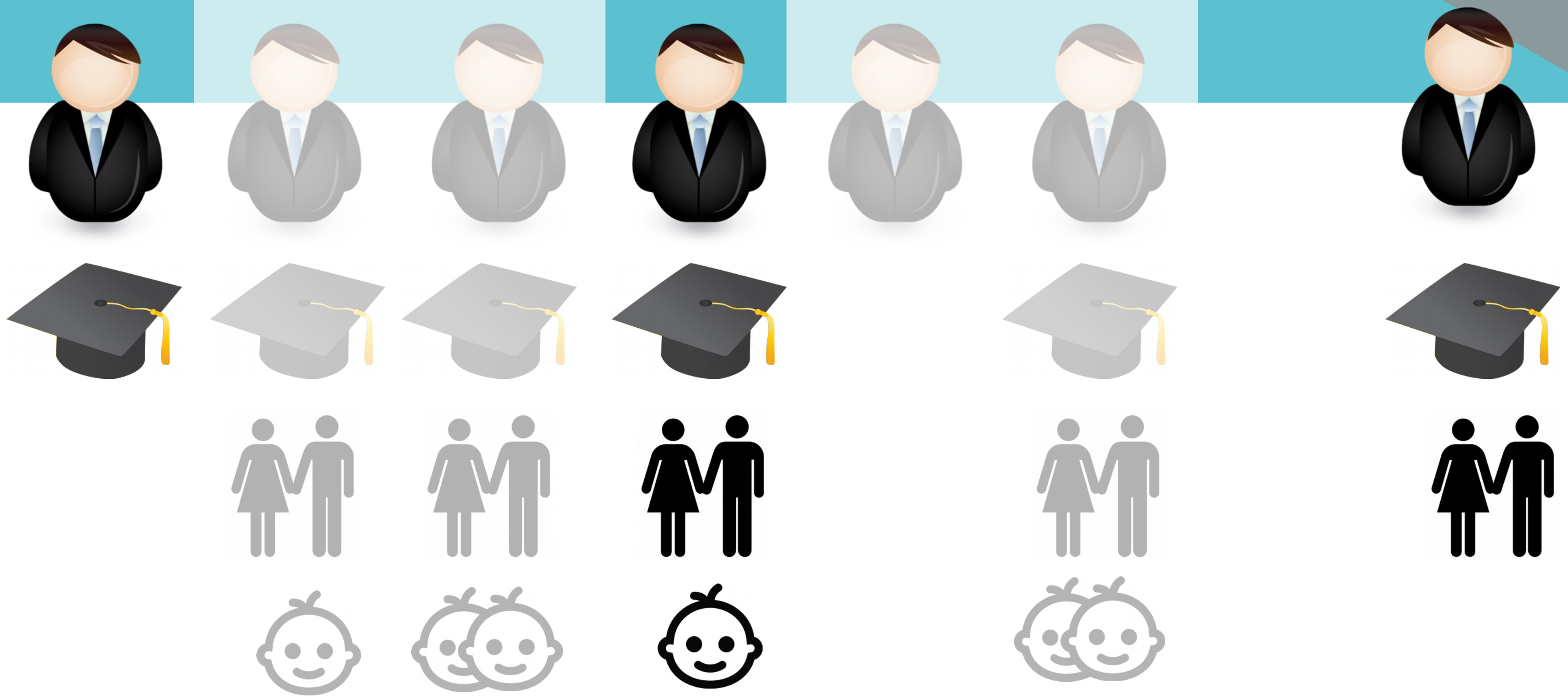
After-execution stage:
...

K Nearest Neighbours method



Age	25	47	55	32	22	45	28
Salary	50	120	100	80	30	90	?

K Nearest Neighbours method



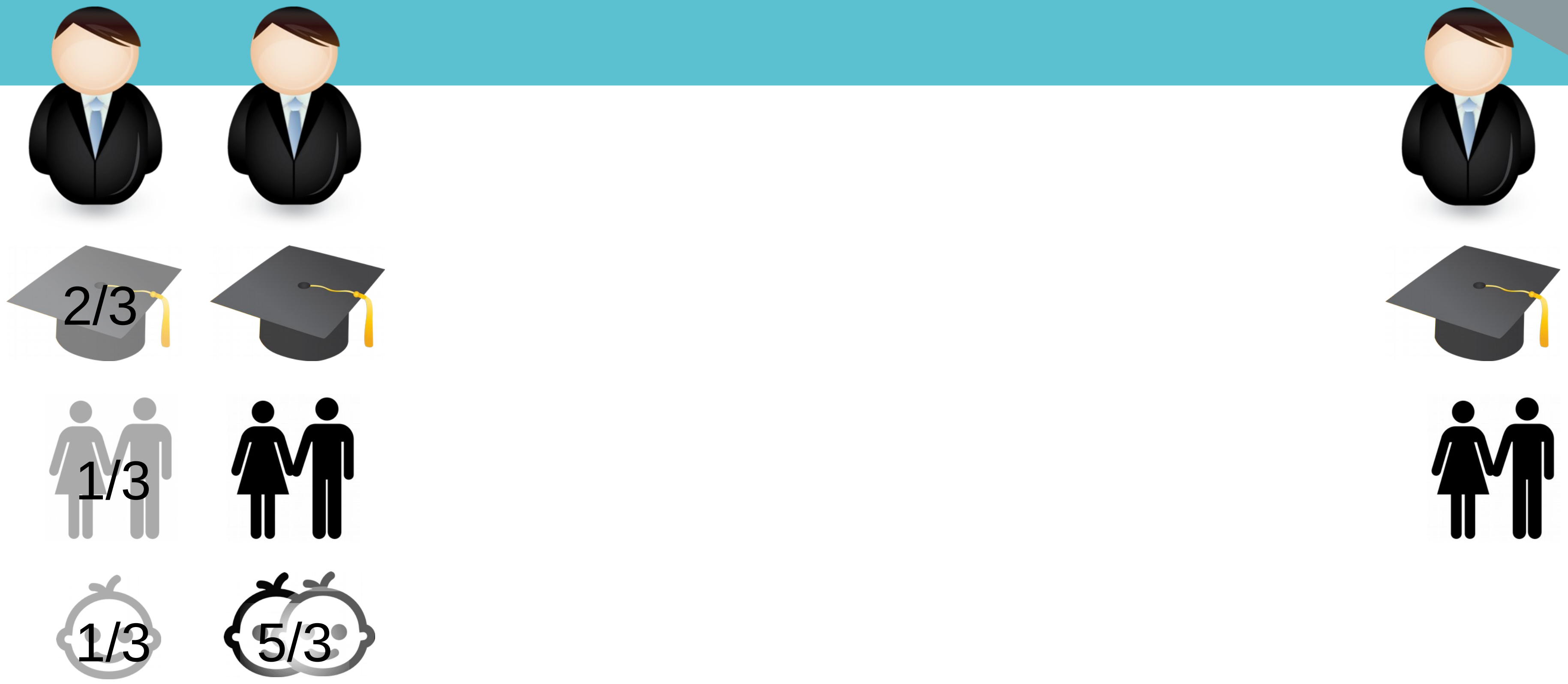
Age	25	47	55	32	22	45	28
Salary	50	120	100	80	30	90	?

Gradient approach to kNN

Goal: not to store the whole train set

Idea: to use a fixed number of virtual objects
that provide the best possible prediction quality

Gradient approach to kNN



Age	27	47	28
Salary	53	103	?

Gradient approach to kNN

Math for learning:

Loss function

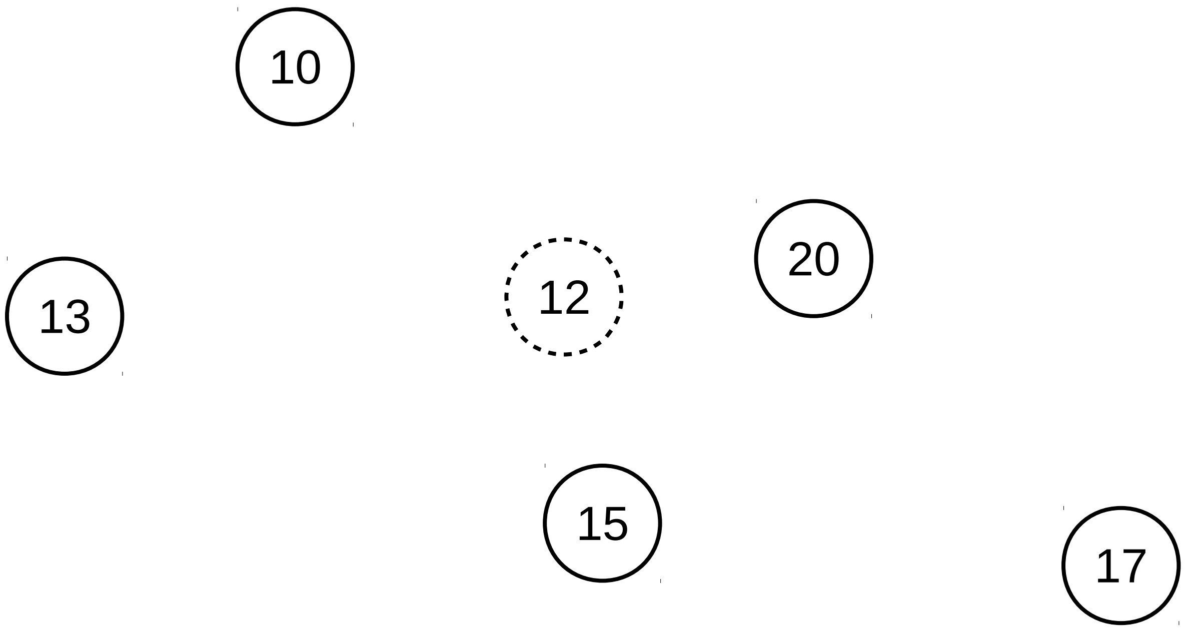
\+

Stochastic gradient descent to optimize it

\+

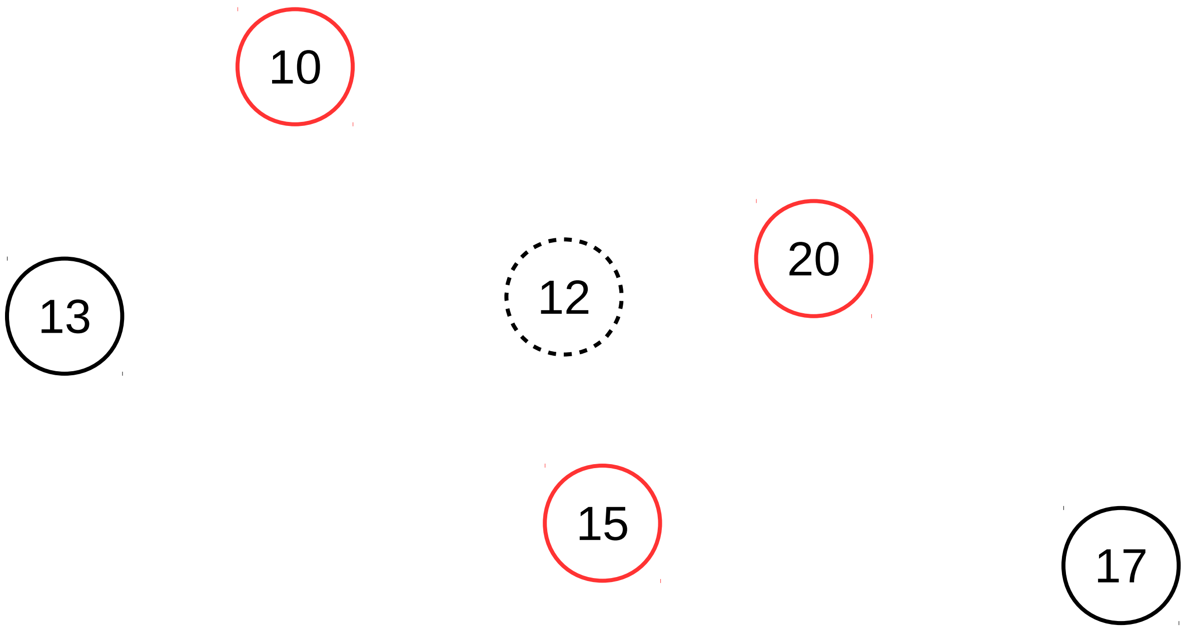
K Nearest Neighbours method

Gradient approach to kNN



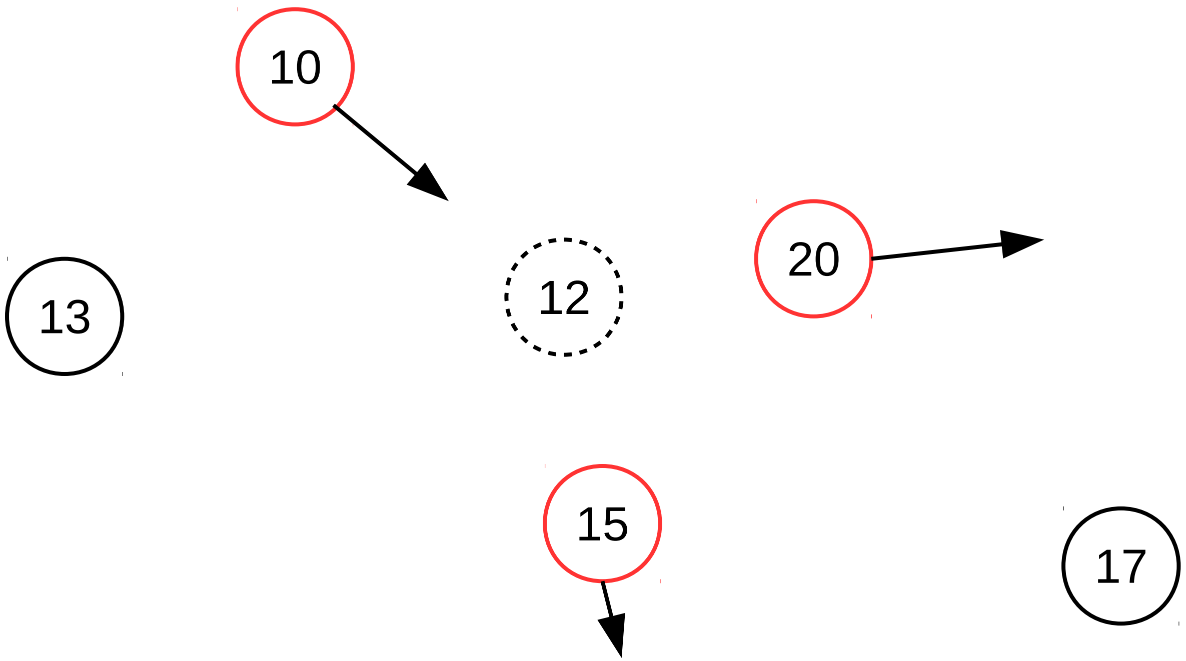
Gradient approach to kNN

K = 3

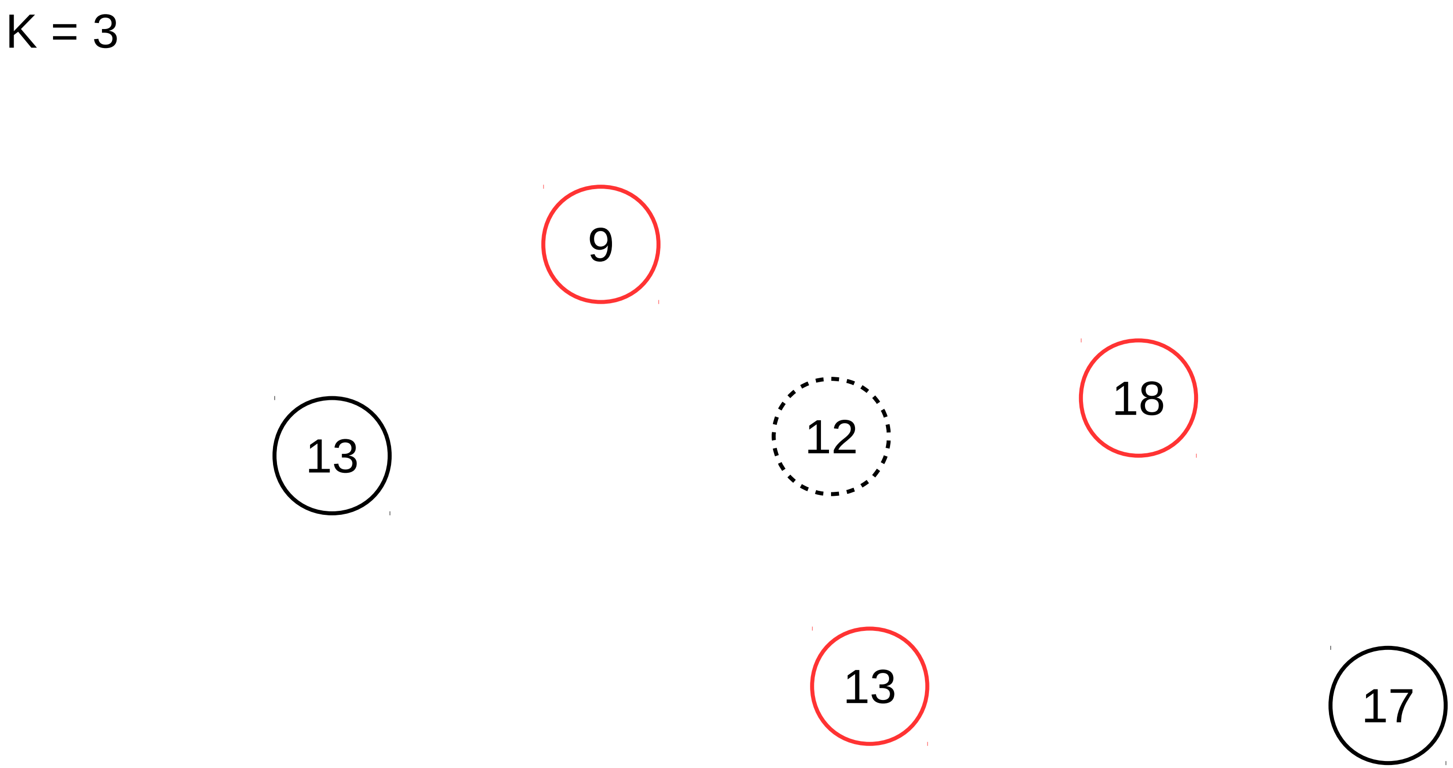


Gradient approach to kNN

K = 3



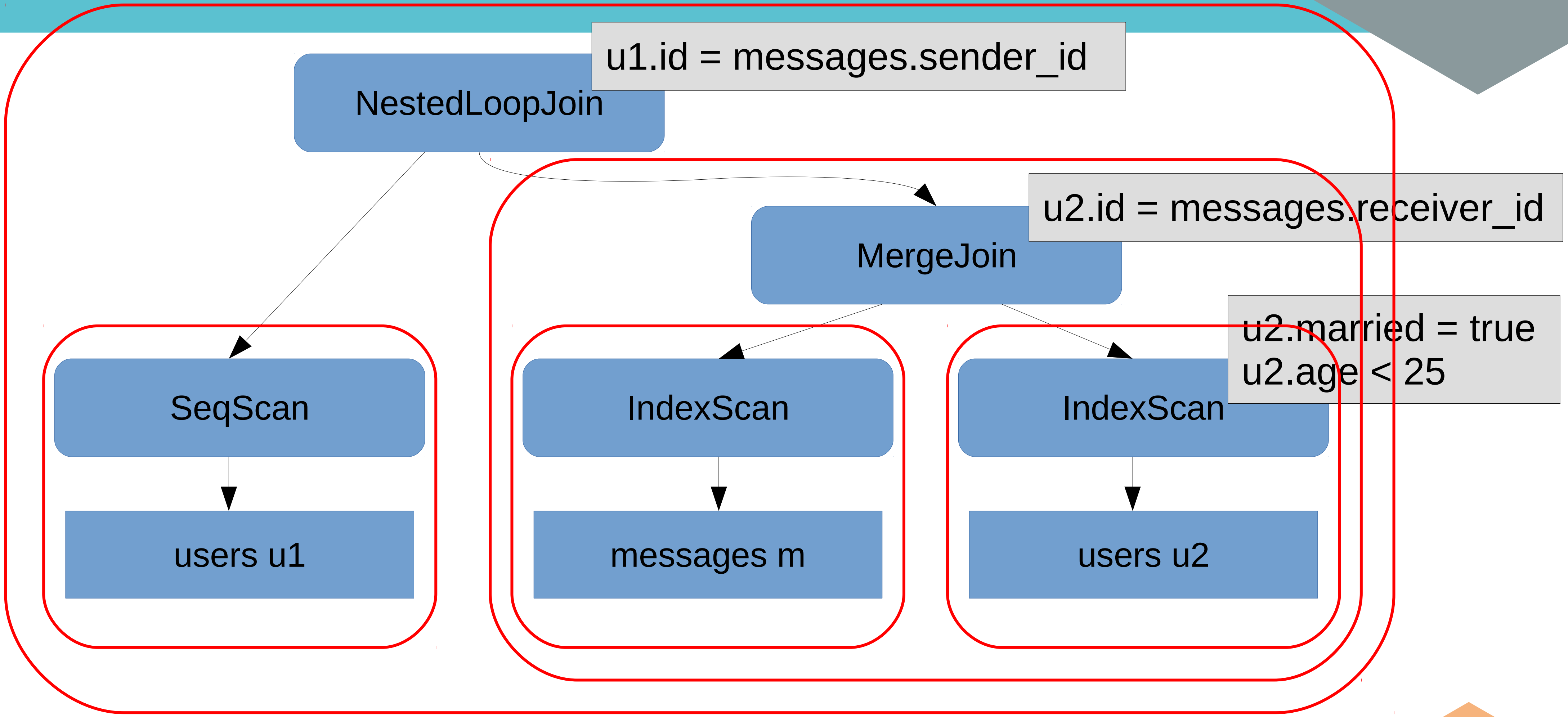
Gradient approach to kNN



Adaptive query optimization

Theory

The object is a node with its subtree



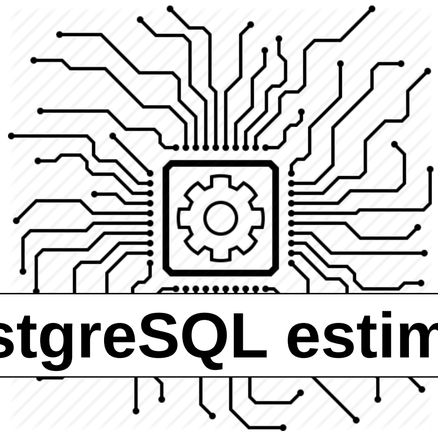
Histograms

users.id = messages.receiver_id
AND
users.married = true
AND
users.age < 25

Information about the data

Clauses list

Node cardinality is 105 tuples!



PostgreSQL estimator

Clause selectivities

- 0.0001
- 0.73
- 0.23

Relations to join

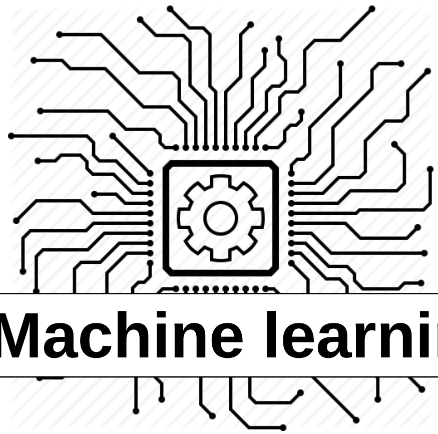
- users
- messages

users.id = messages.receiver_id
AND
users.married = **const**
AND
users.age < **const**

Information about the data

Clauses list

Node cardinality is 1017 tuples!



Machine learning

*equal clauses must be handled specially: transform into clause with an equalence class as an argument

Machine learning problem statement

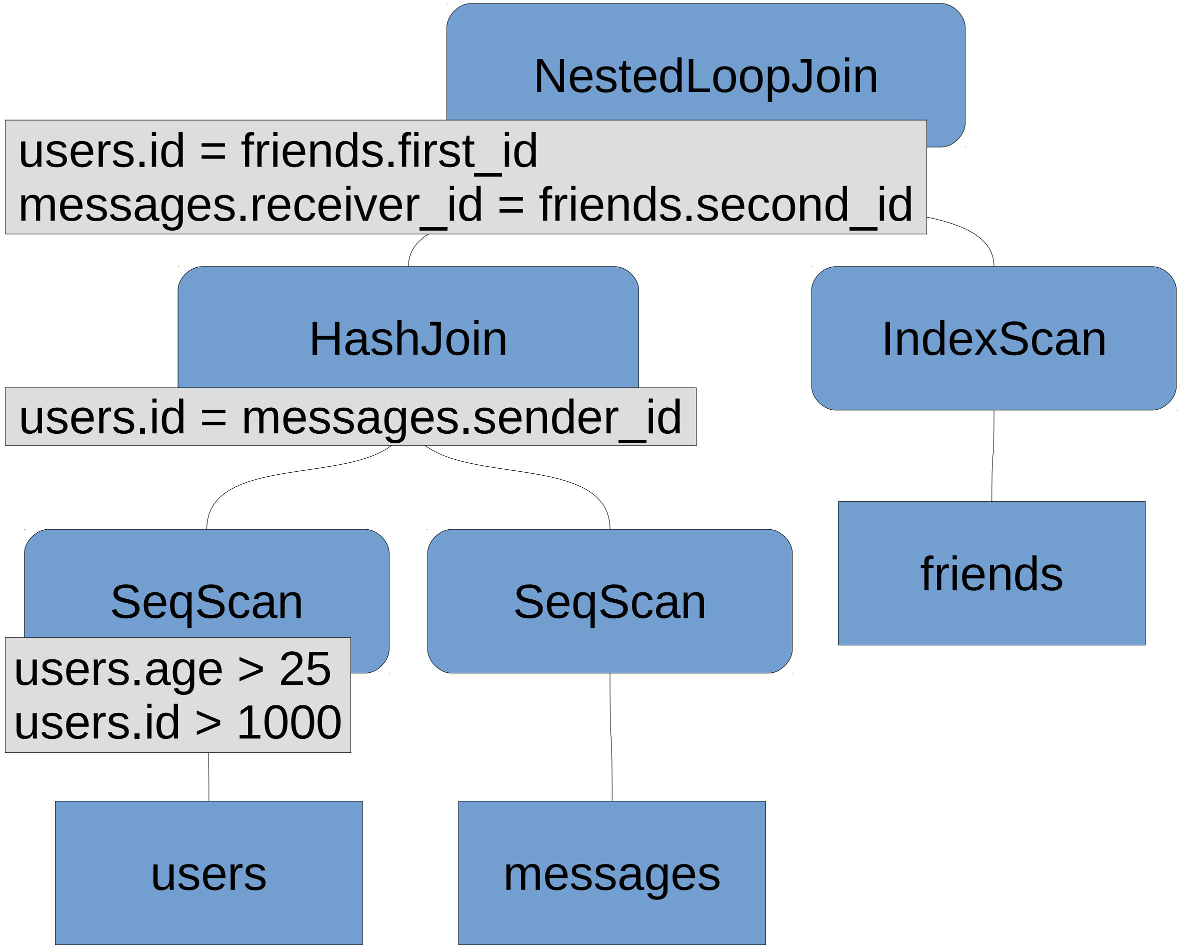
Base relations
- users
- messages

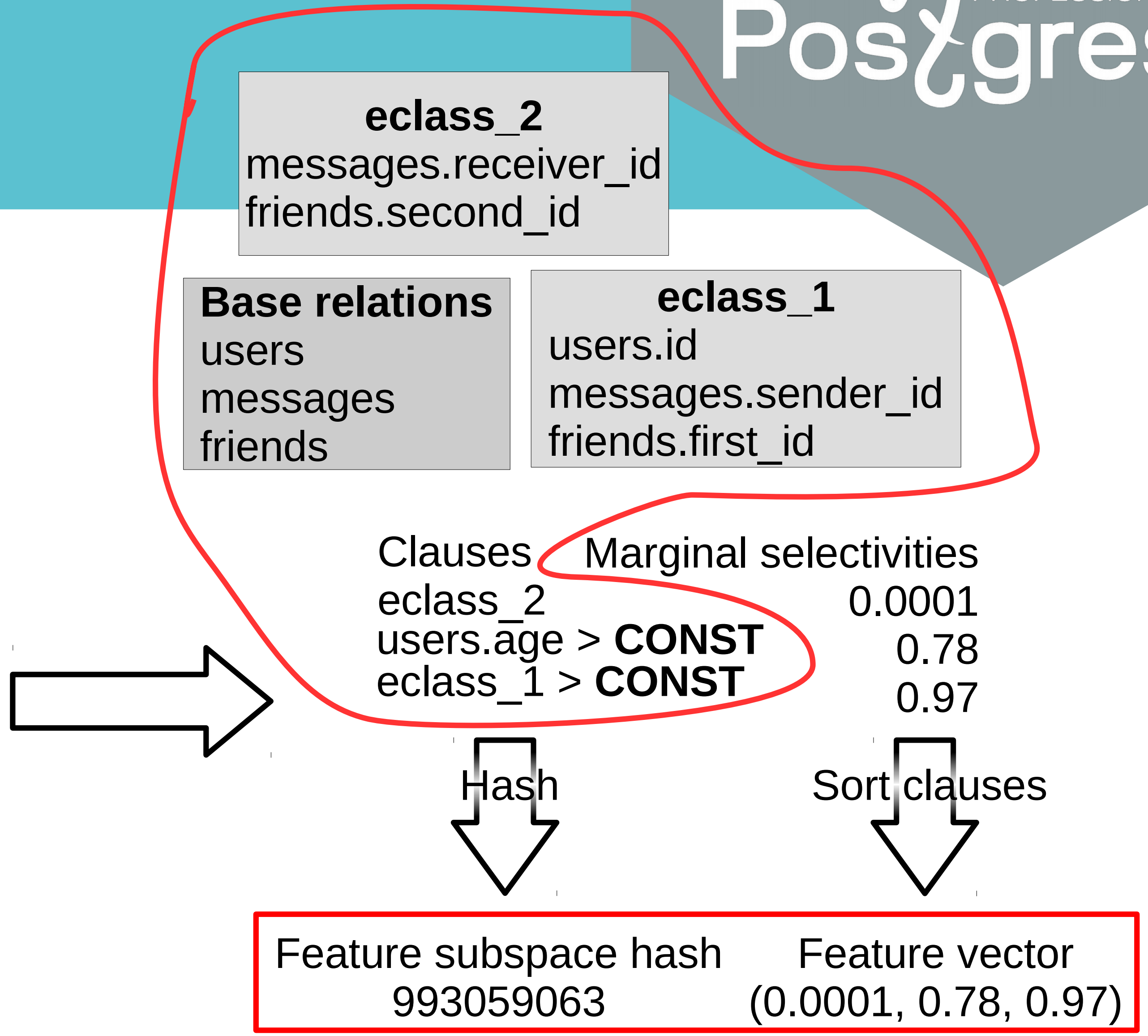
Object is a plan node

Features (clause types)	users.id = messages.receiver_id	0.0001
	users.married = **const**	0.73
	users.age < **const**	0.23
Hidden variable	Node cardinality	?

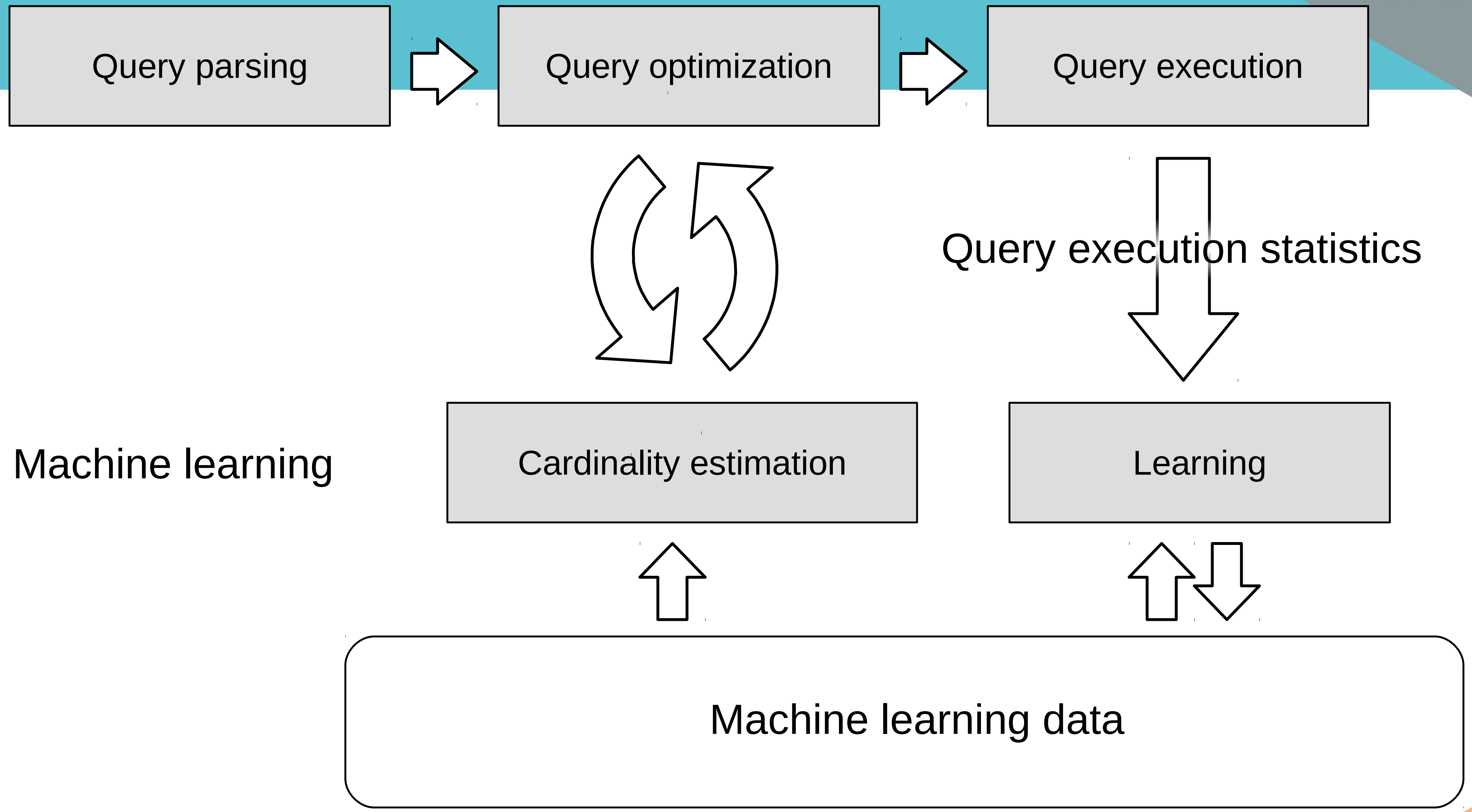
Each set of base relations and clause types induce its own machine learning problem (*feature subspace hash*)!

```sql
SELECT * FROM users, messages, friends
WHERE users.age > 25 AND users.id > 1000
AND users.id = messages.sender_id
AND users.id = friends.first_id
AND messages.receiver_id = friends.second_id;
```





Workflow


Query parsing
Query optimization
Query execution
Query execution statistics
Machine learning
Cardinality estimation
Learning
Machine learning data

Theoretical properties

- Will it converge?

  Yes, in the finite number of steps

- How fast will it converge?

  Don't know (in practice in a few steps)

- What guarantees on obtained plans or regressor do we have?

  Predictions are correct for all executed paths

  With perfect cost model obtained plans are not worse

Adaptive query optimization

Implementation

Source code

Current code for vanilla PostgreSQL (extension + patch):
https://github.com/tigvarts/aqo

Open Source, PostgreSQL license

Without extension equivalent to standard PostgreSQL optimizer

Needs to be in the shared preload libraries

Hooks

Planning stage (prediction):

set_baserel_size_estimates
get_parameterized_baserel_size
set_joinrel_size_estimates
get_parameterized_joinrel_size

After-execution stage:

ExecutorEnd – learning
ExplainOnePlan – visualization

Other:

planner_hook – prepare to the planning stage
ExecutorStart – setting the flags for statistics collection
copy_generic_path_info – transmit Path information to Plan node

Control

For some queries we don't need AQO.

So we need a mechanism to disable AQO learning or usage for some queries.

Query types

Query type is the set of queries, which differ only in their constants.

Query type:
SELECT * **FROM** users **WHERE** age > **const AND** city = **const**;

Queries:
SELECT * **FROM** users **WHERE** age > 18 **AND** city = 'Ottawa';
SELECT * **FROM** users **WHERE** age > 65 **AND** city = 'New York';
…

AQO tables

aqo_queries
- Query_hash
- Learn AQO
- Use AQO
- Feature_space

Settings

aqo_query_texts
- Query_hash
- Query_text

For user

aqo_data
- Feature_space
- Feature_subspace
- Features
- Target

Machine learning

Control

The users don't want to configure AQO query settings manually.

So we need a mechanism to determine automatically whether the query needs AQO.

It is called auto tuning.

AQO tables

aqo_queries
- Query_hash
- Learn AQO
- Use AQO
- Feature_space
- Auto_tuning

Settings

aqo_query_texts
- Query_hash
- Query_text

For user

aqo_data
- Feature_space
- Feature_subspace
- Features
- Target

Machine learning

aqo_query_stat
- Query_hash
- Planning time
- Execution time
- Cardinality error
- Number of executions

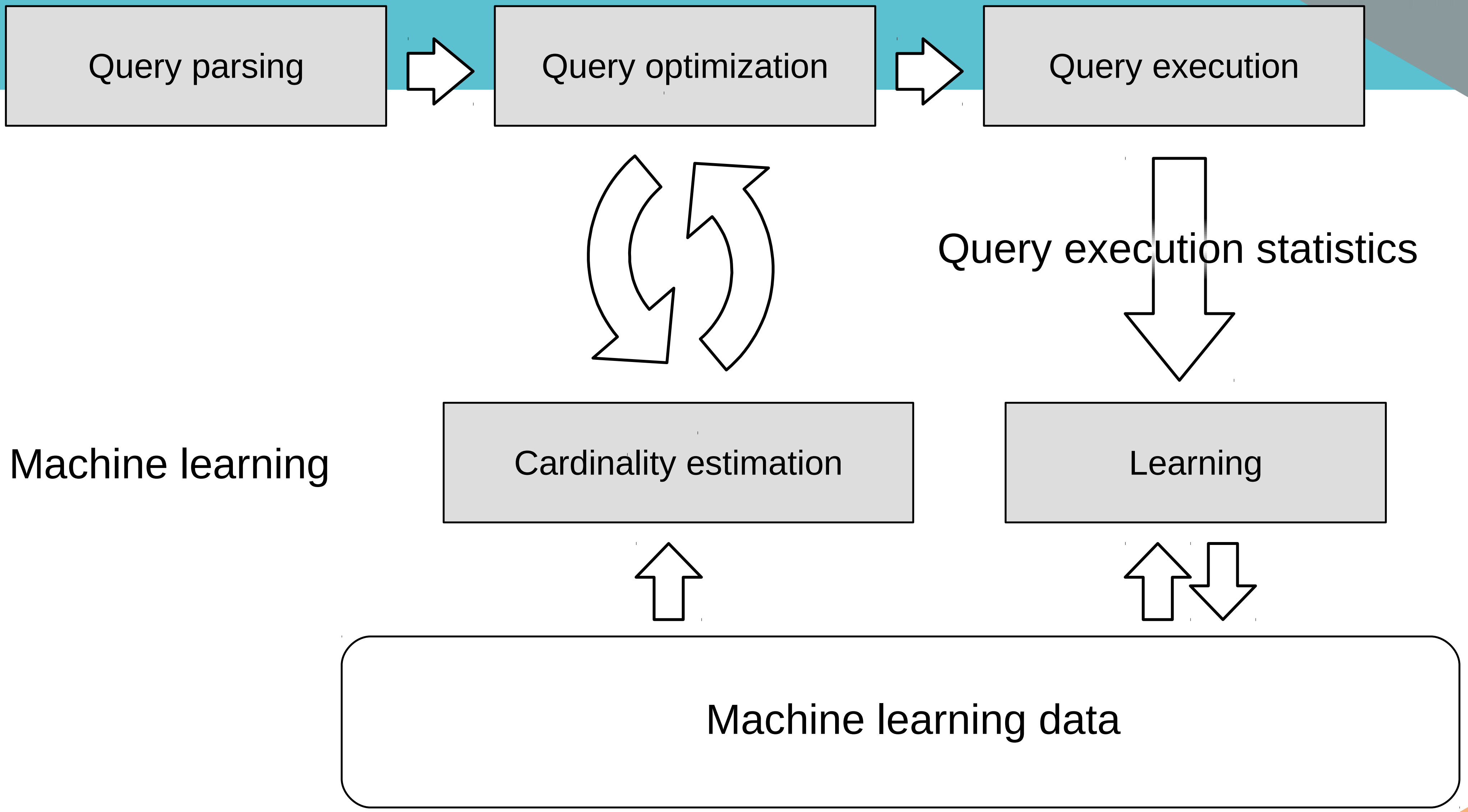
For auto tuning

Control

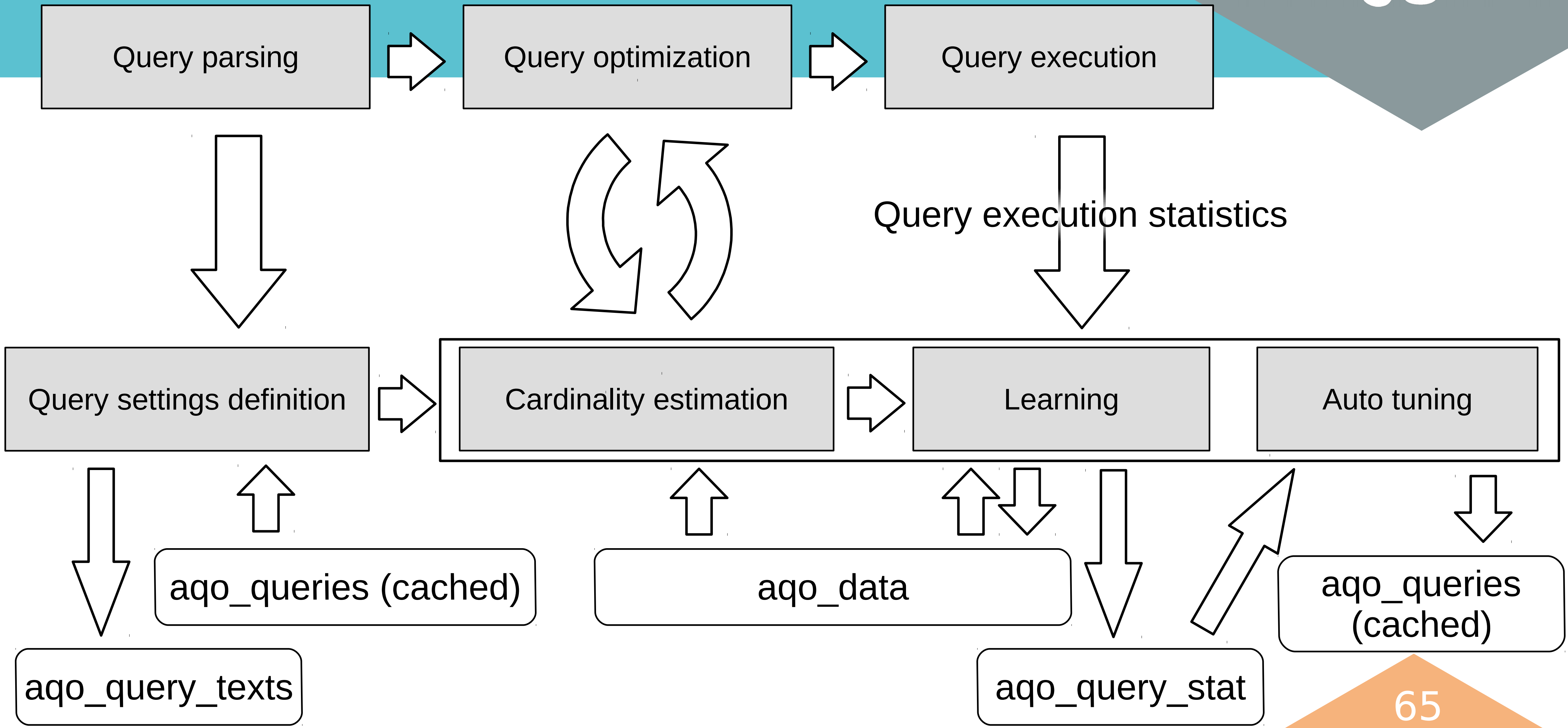
GUC: aqo.mode

- *Disabled*: disabled for all query types
- *Forced*: enabled for all query types
- *Controlled*: use manual settings for known query types, ignore others
- *Intelligent*: use manual settings for known query types,
  tries to tune others automatically

Workflow


Query parsing
Query optimization
Query execution
Query execution statistics
Machine learning
Cardinality estimation
Learning
Machine learning data

Fair workflow

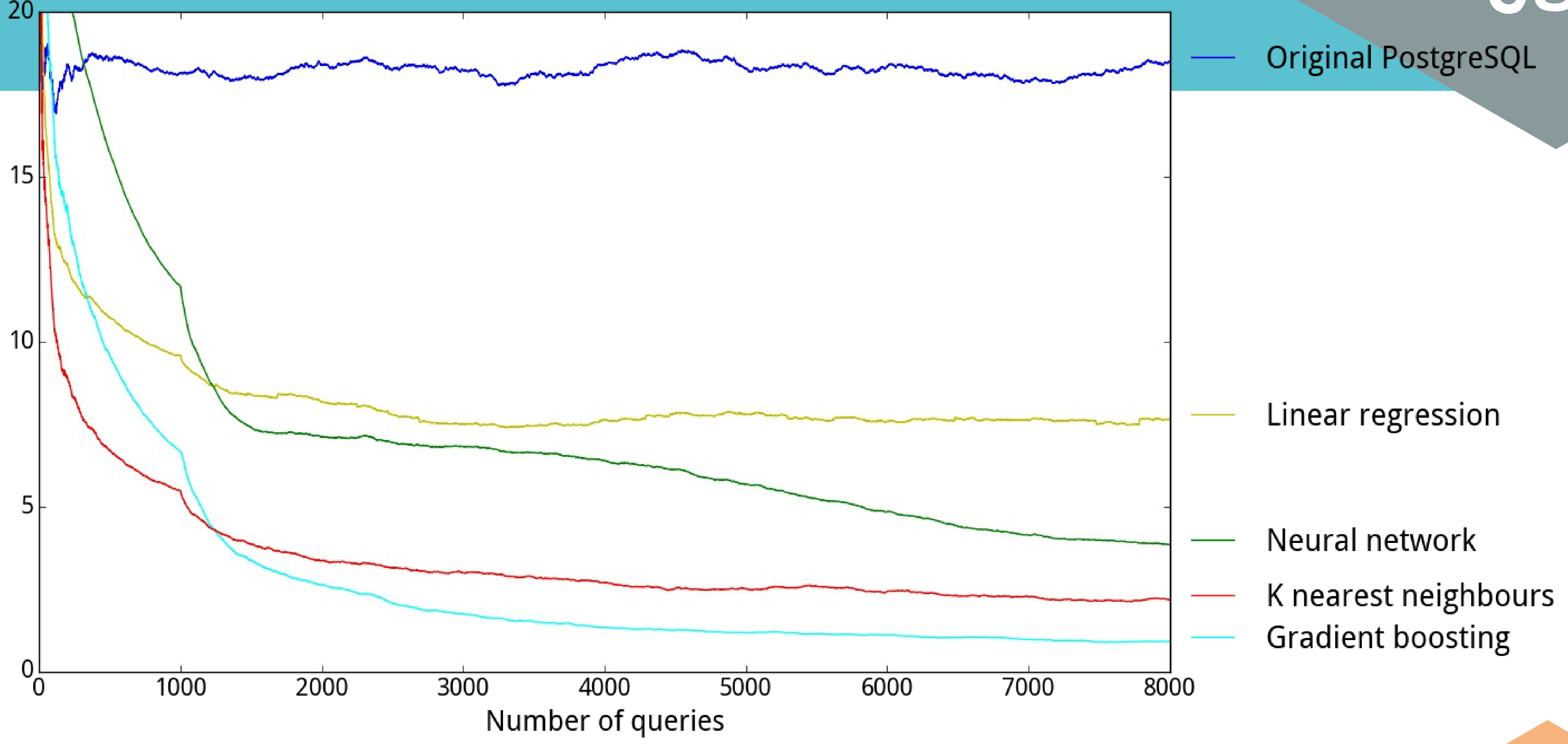


Experimental evaluation

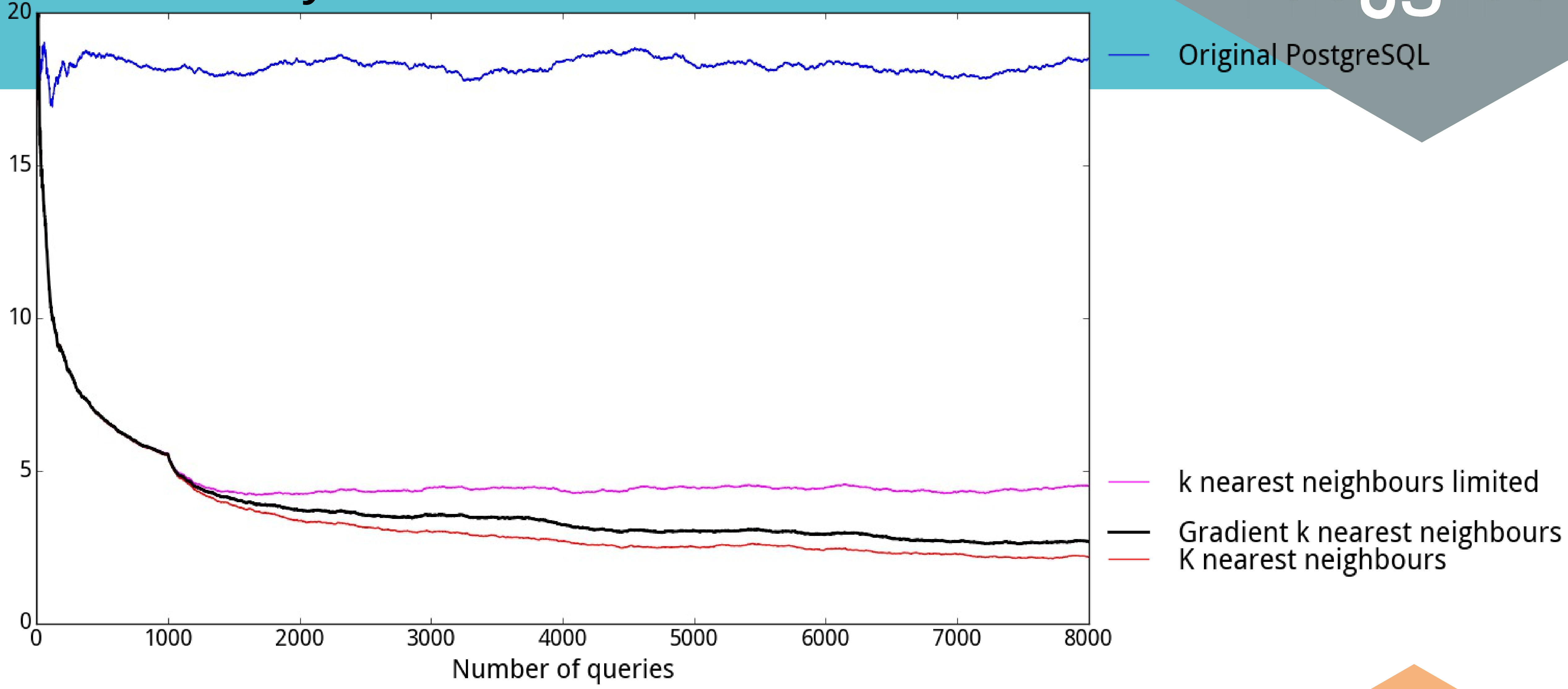
Strongly Correlated Columns (StrongCor)

Cardinality estimation using neural networks
H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte,
CASCON’15, 2015, IBM Corp.

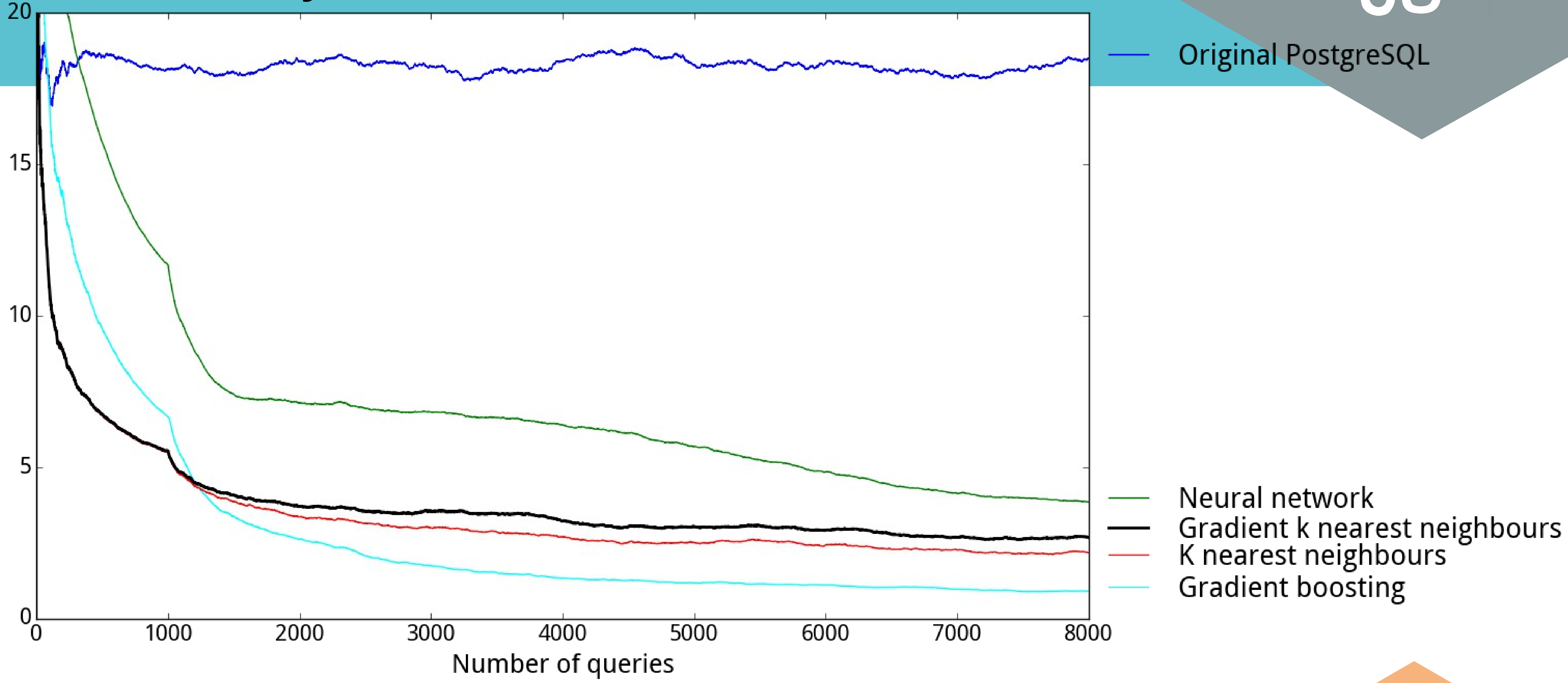
Cardinality estimation error



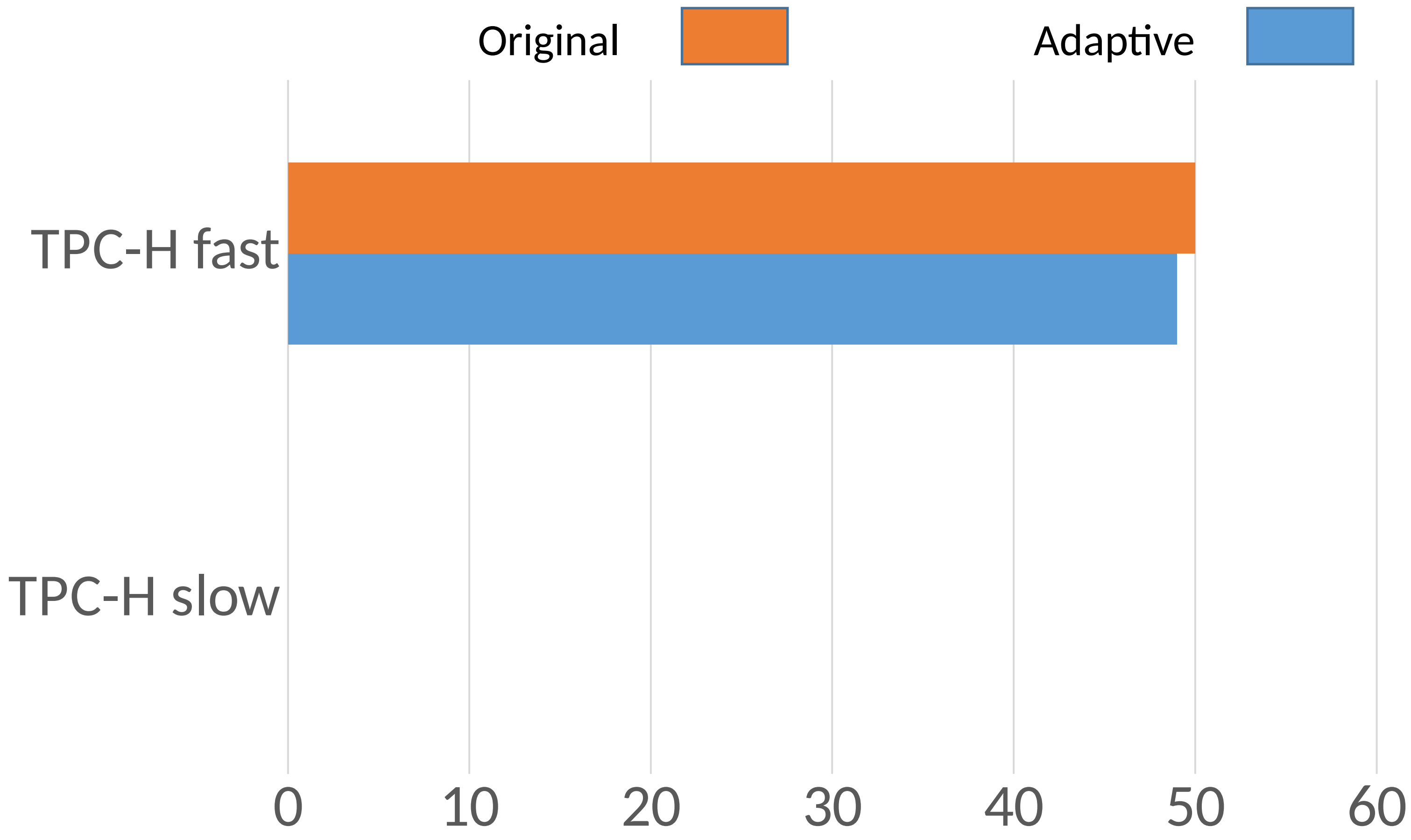
Cardinality estimation error



Cardinality estimation error

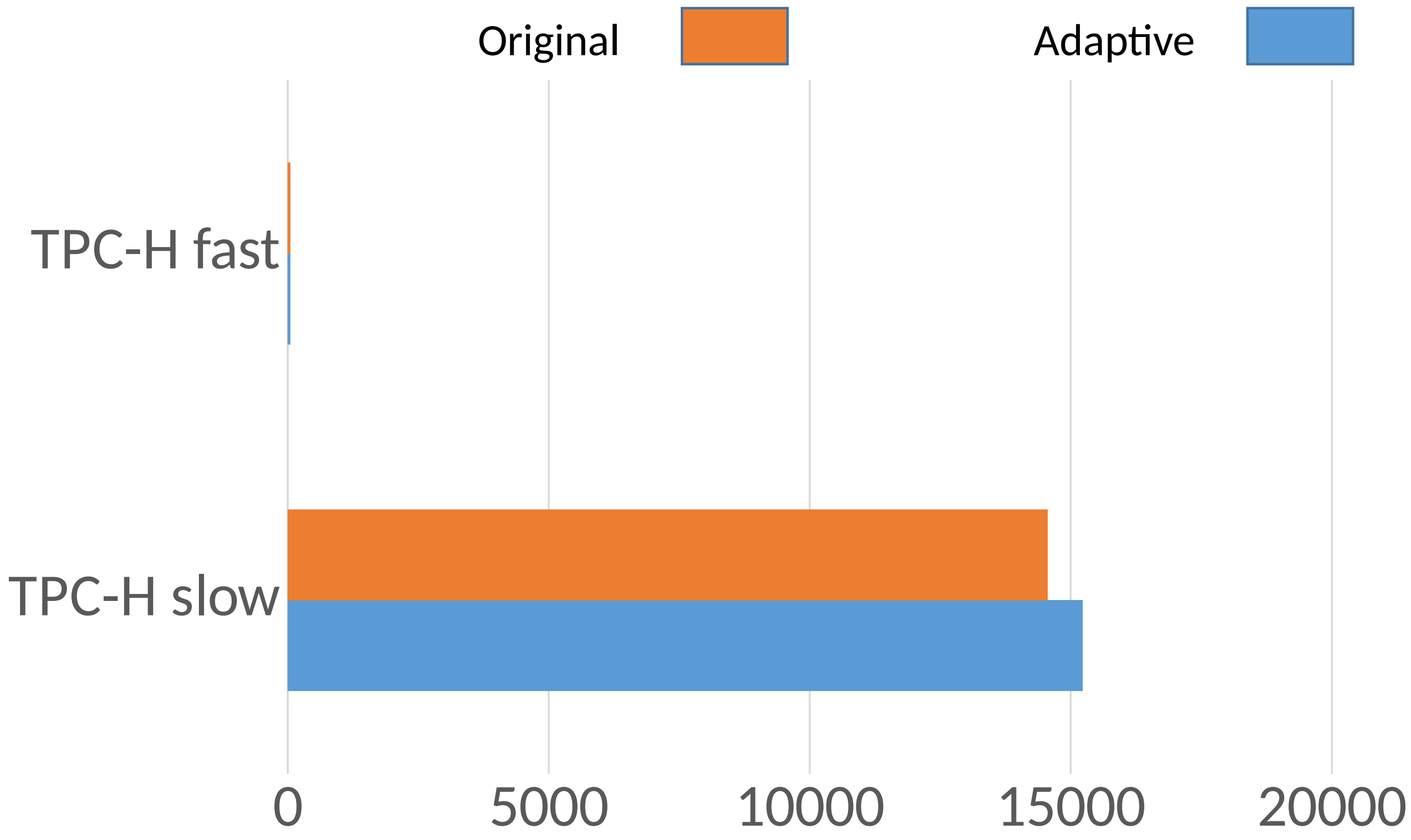


Performance improvement



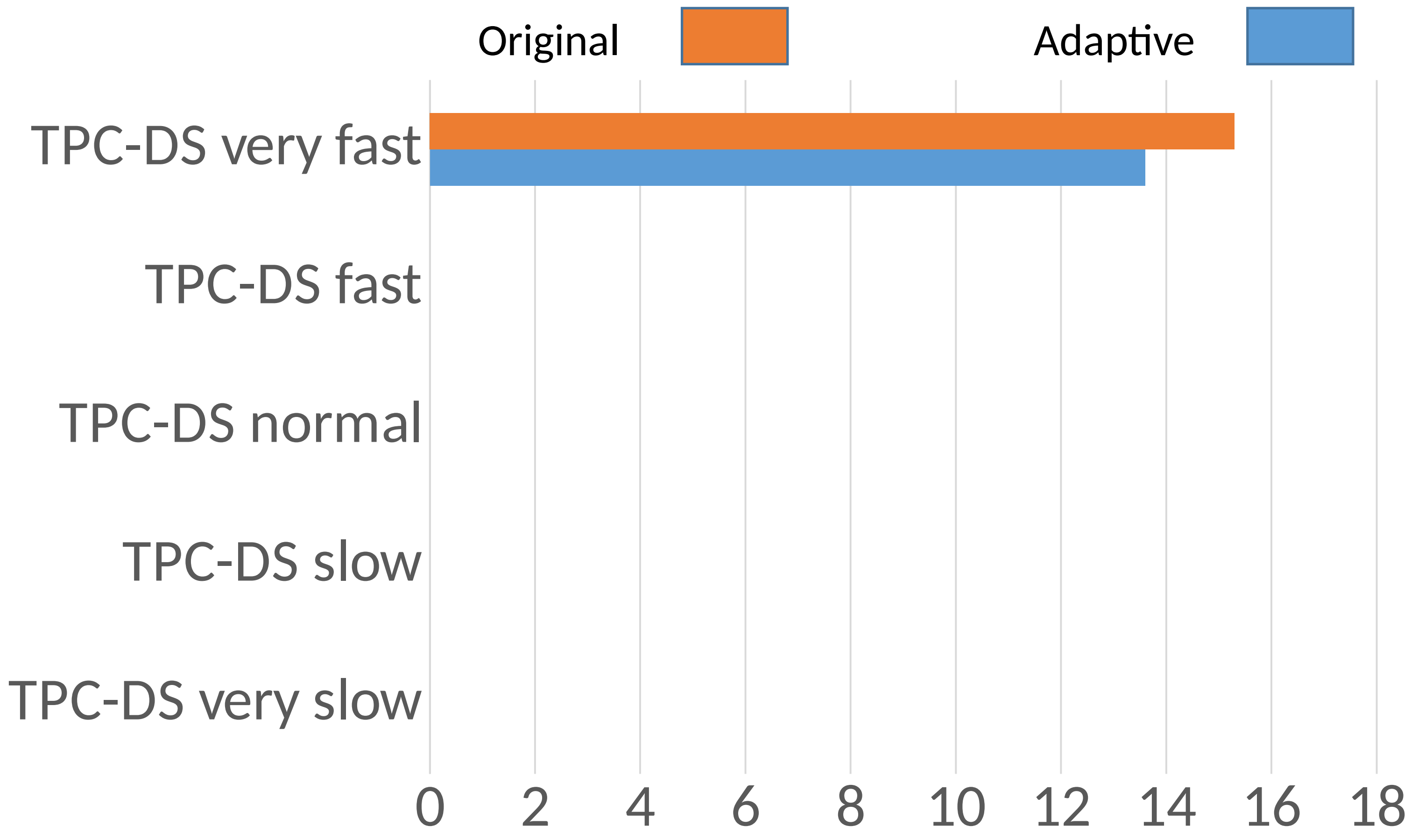
+1.3%

Performance improvement



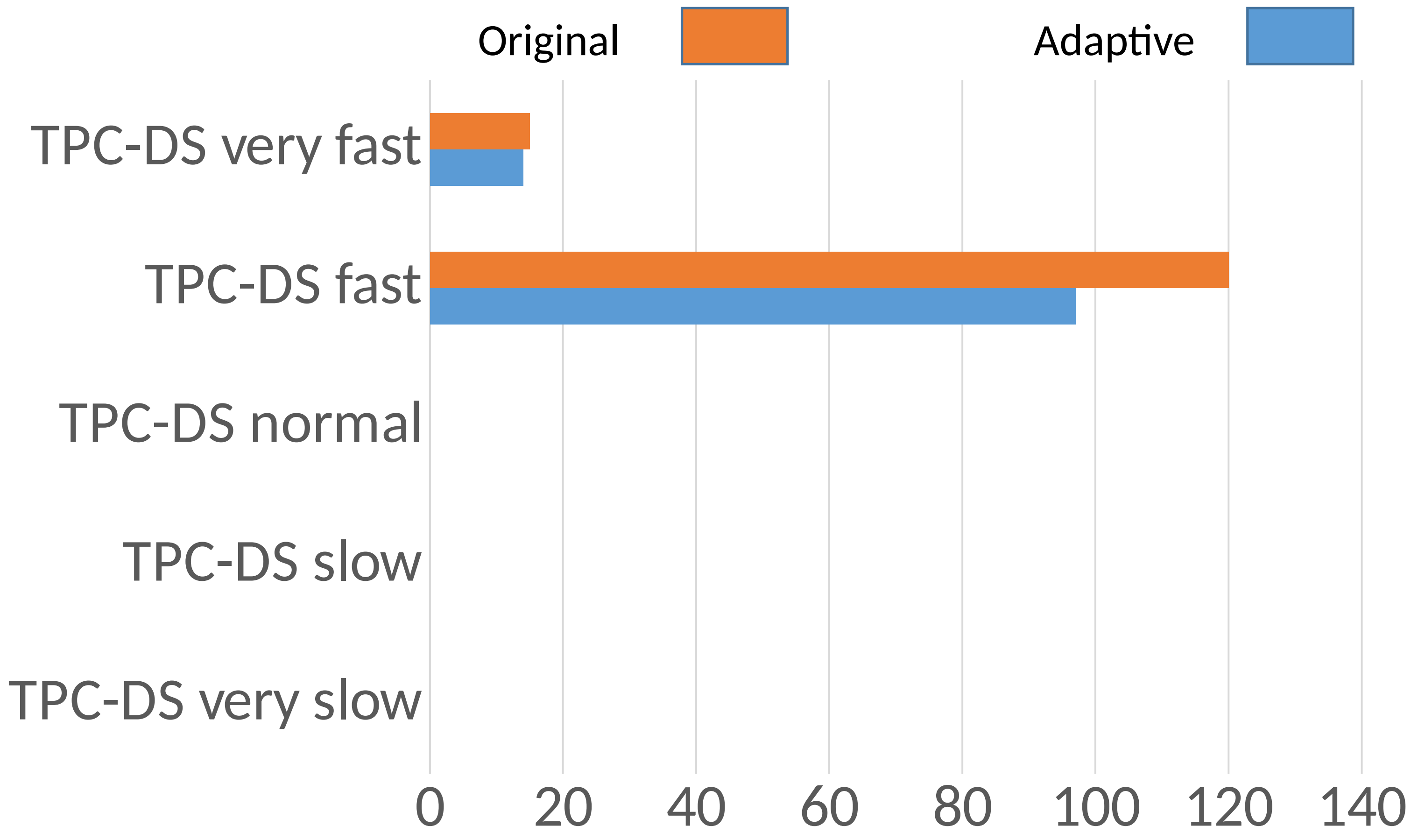
-4.4%

Performance improvement



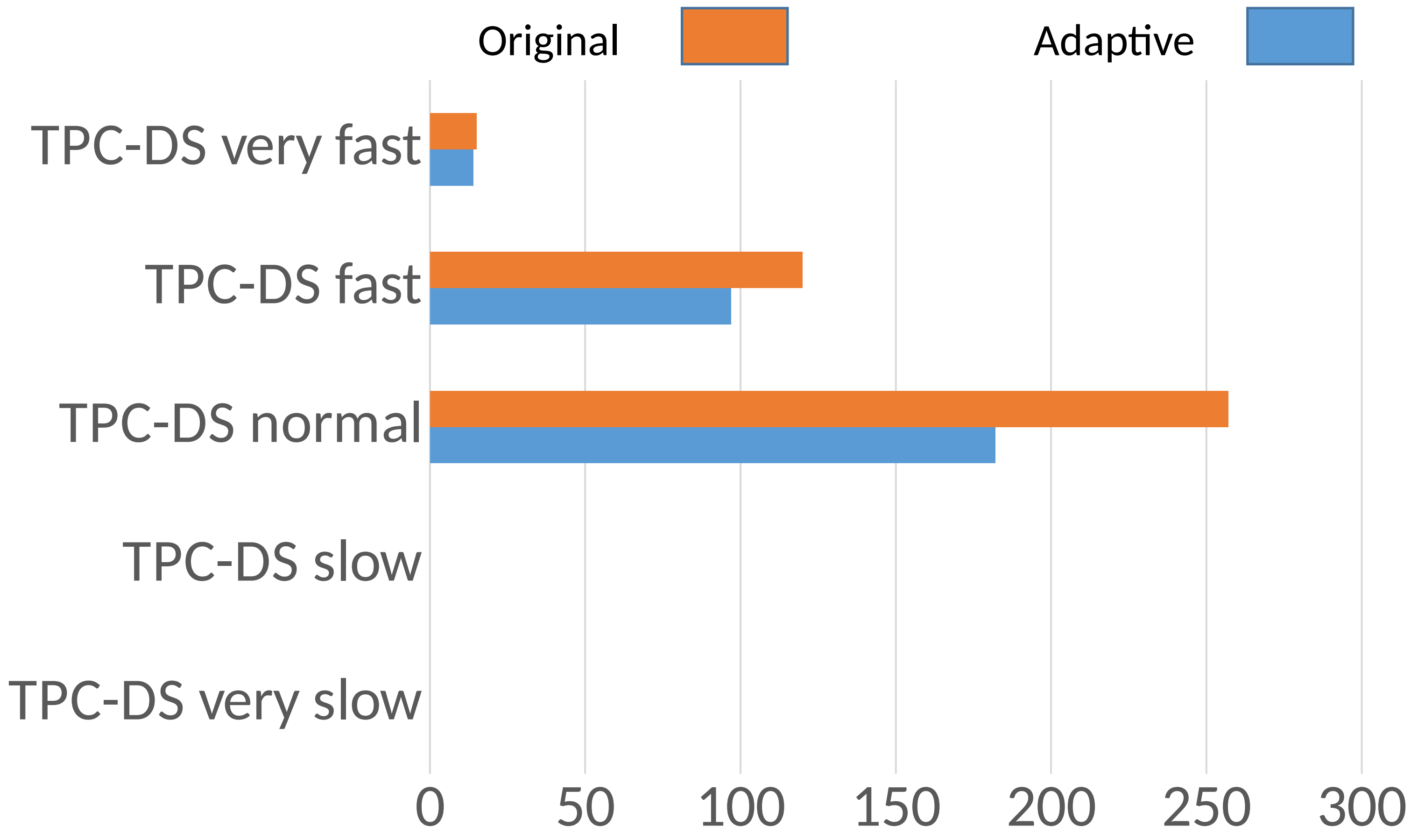
+12%

Performance improvement



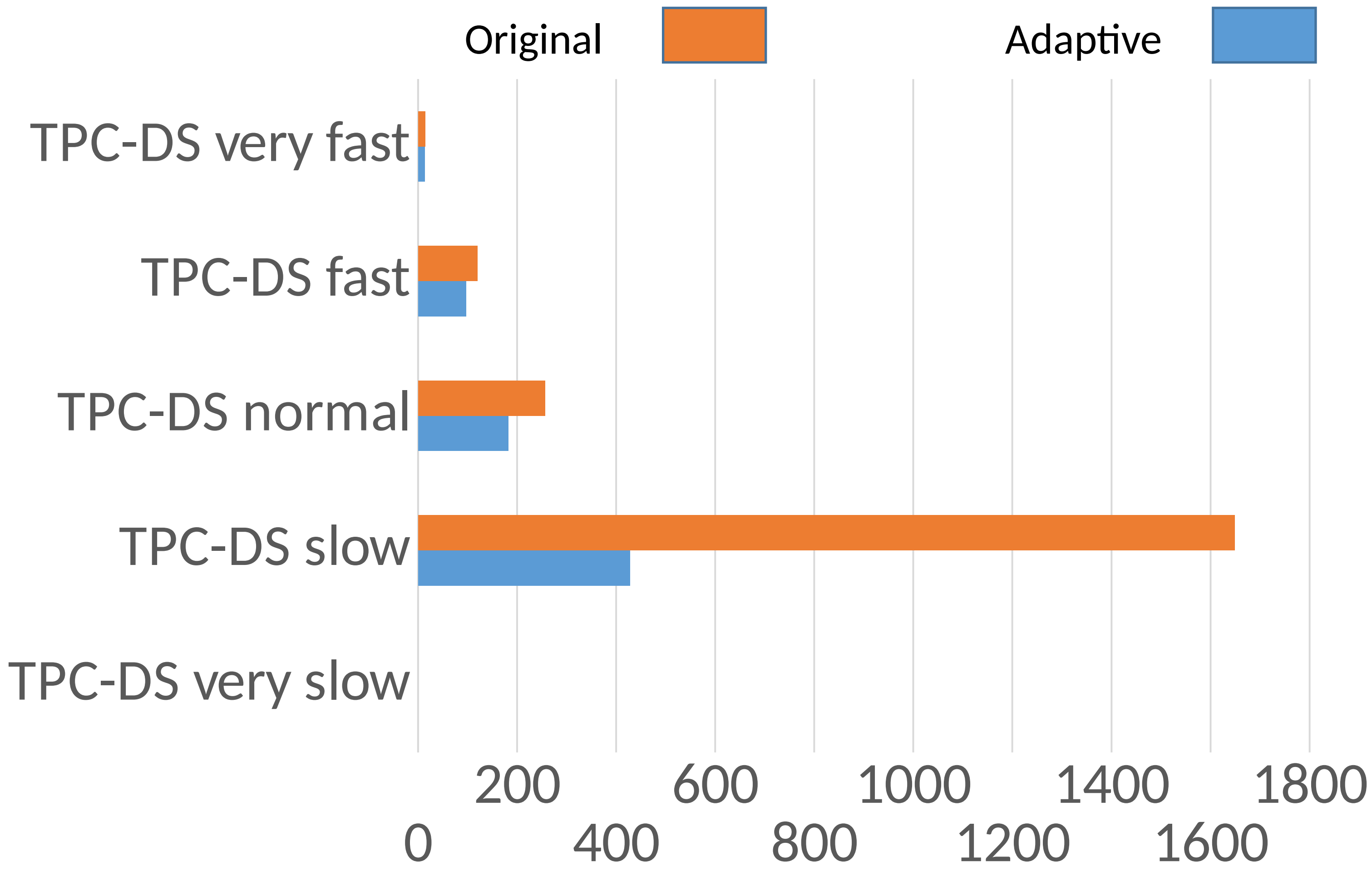
+24%

Performance improvement



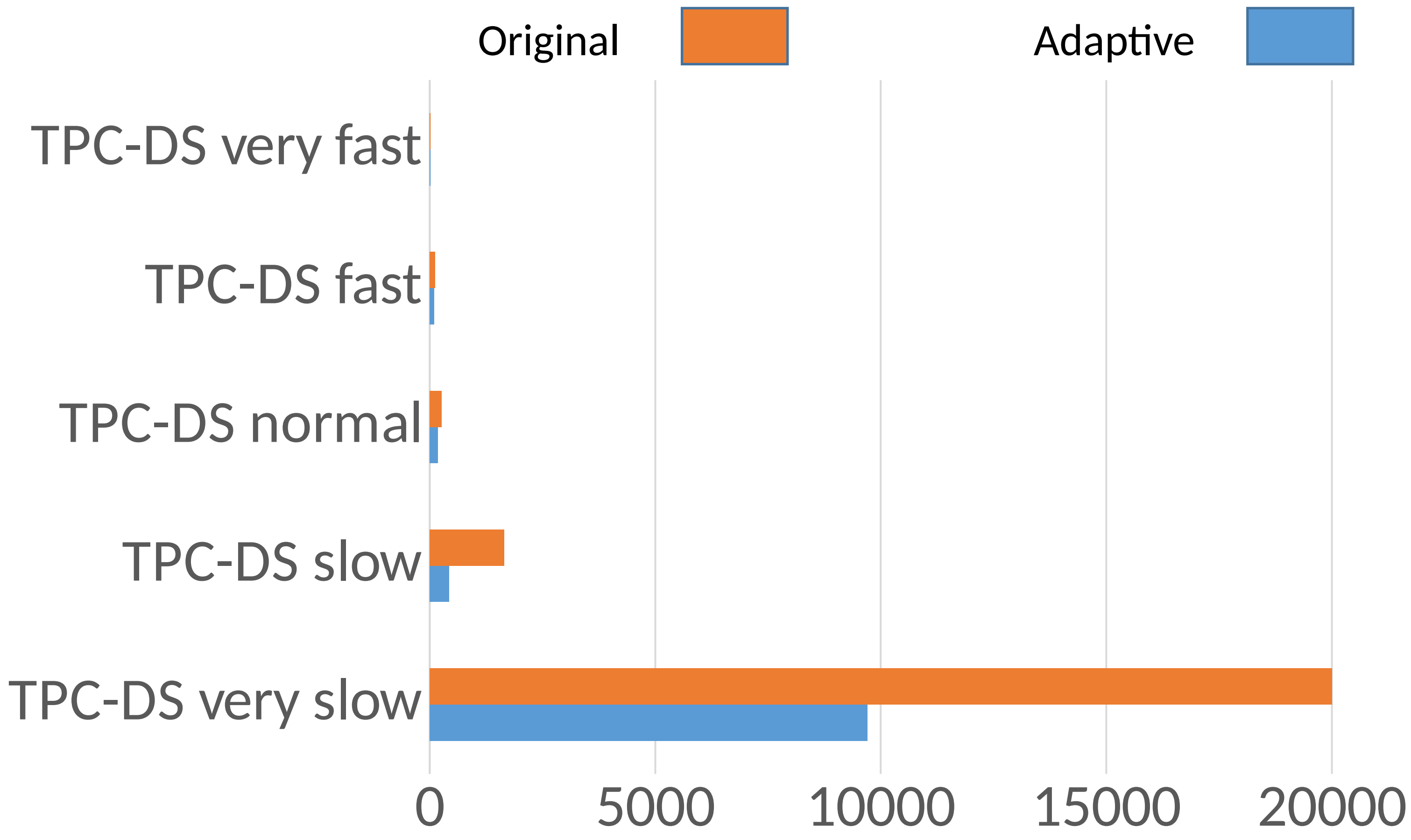
+41%

Performance improvement



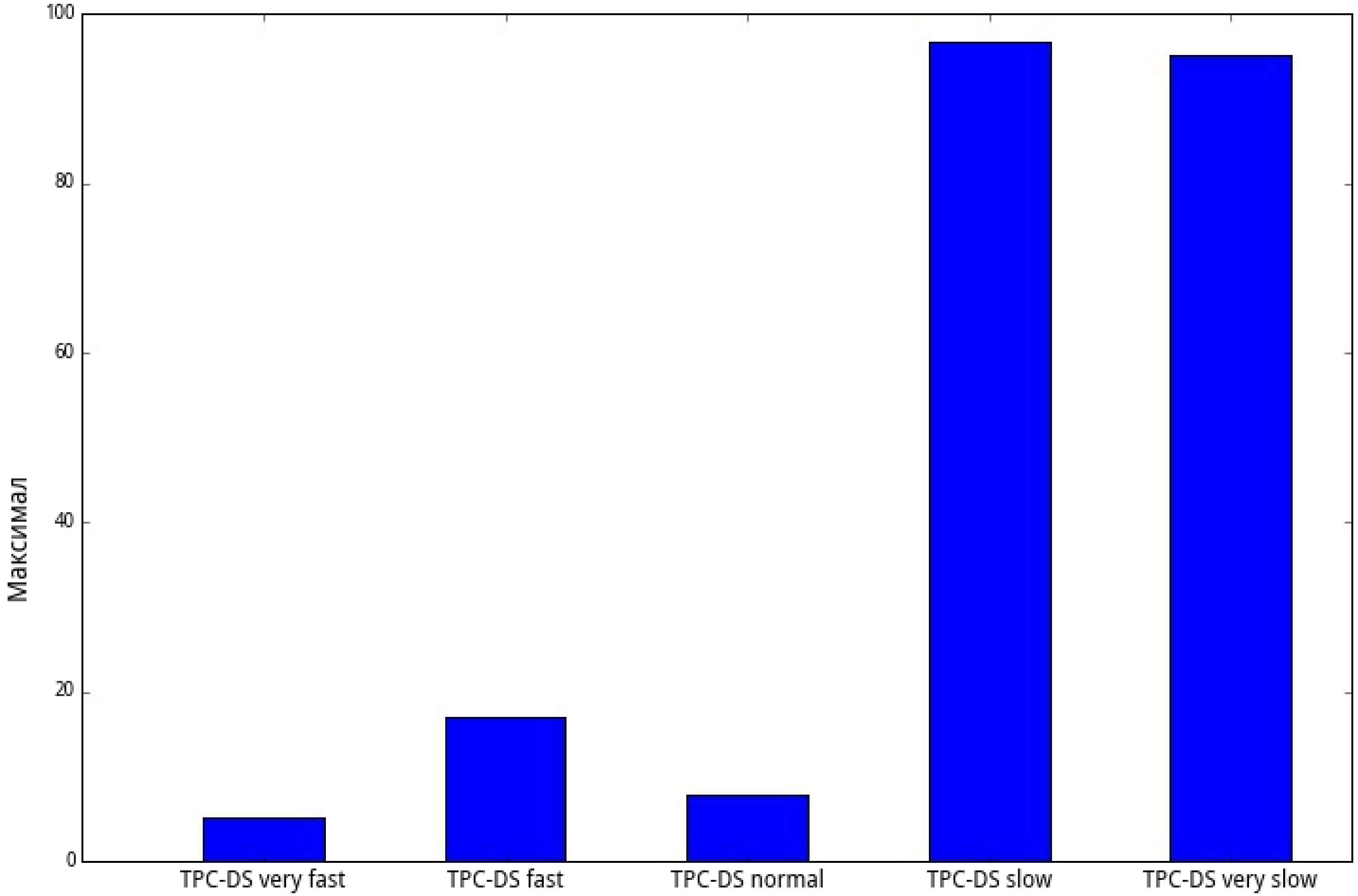
+285%

Performance improvement

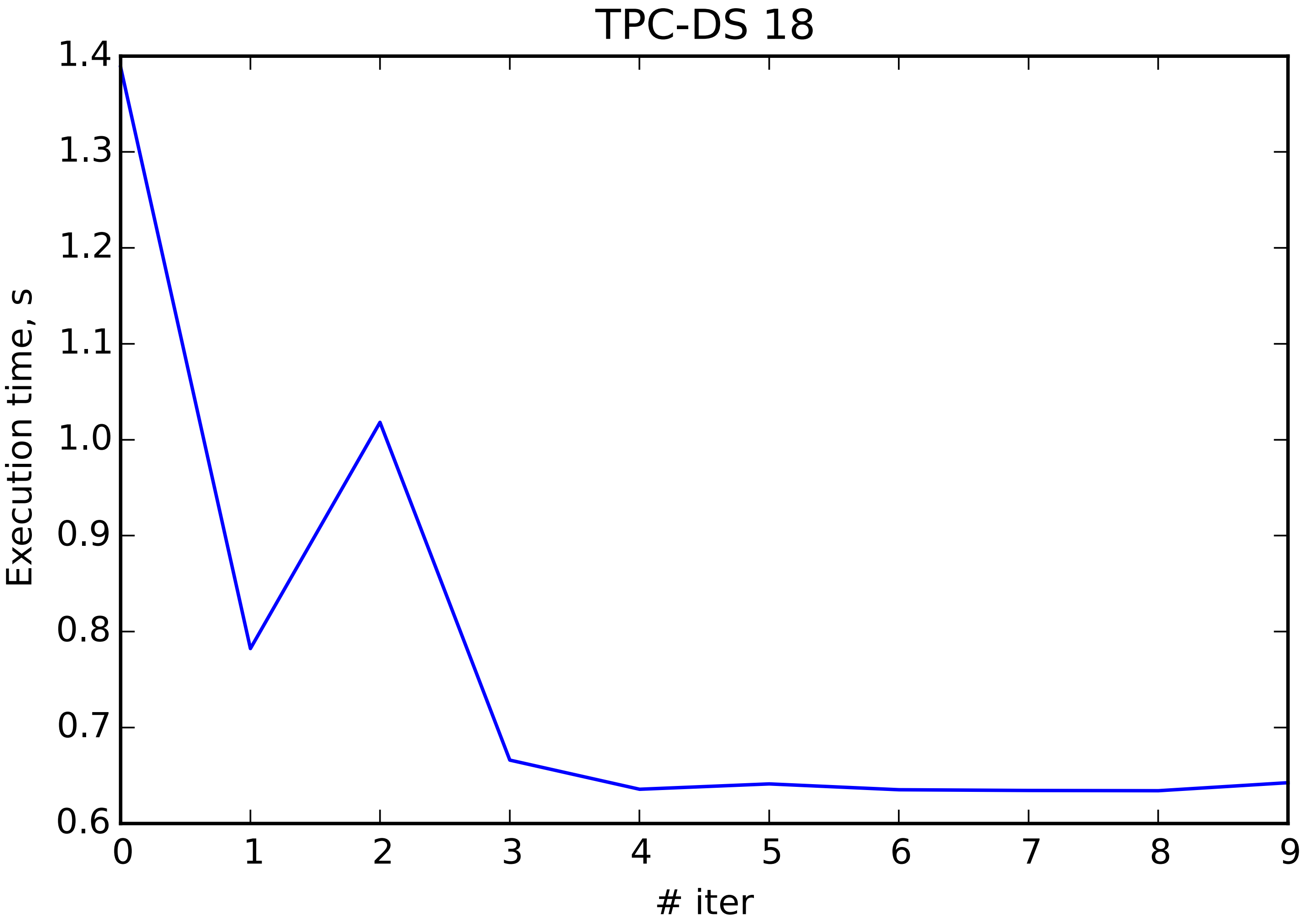


+115%

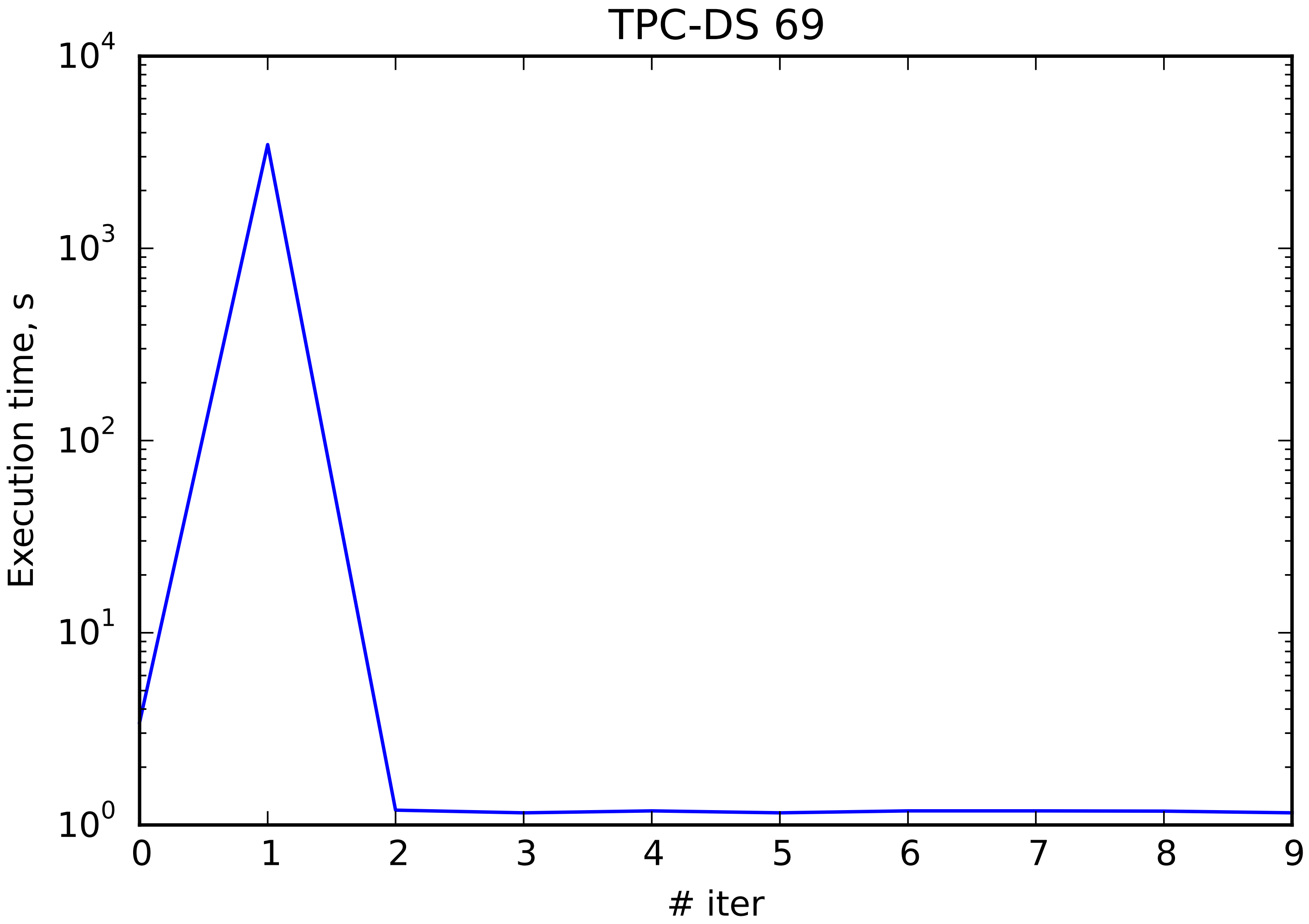
Maximum acceleration



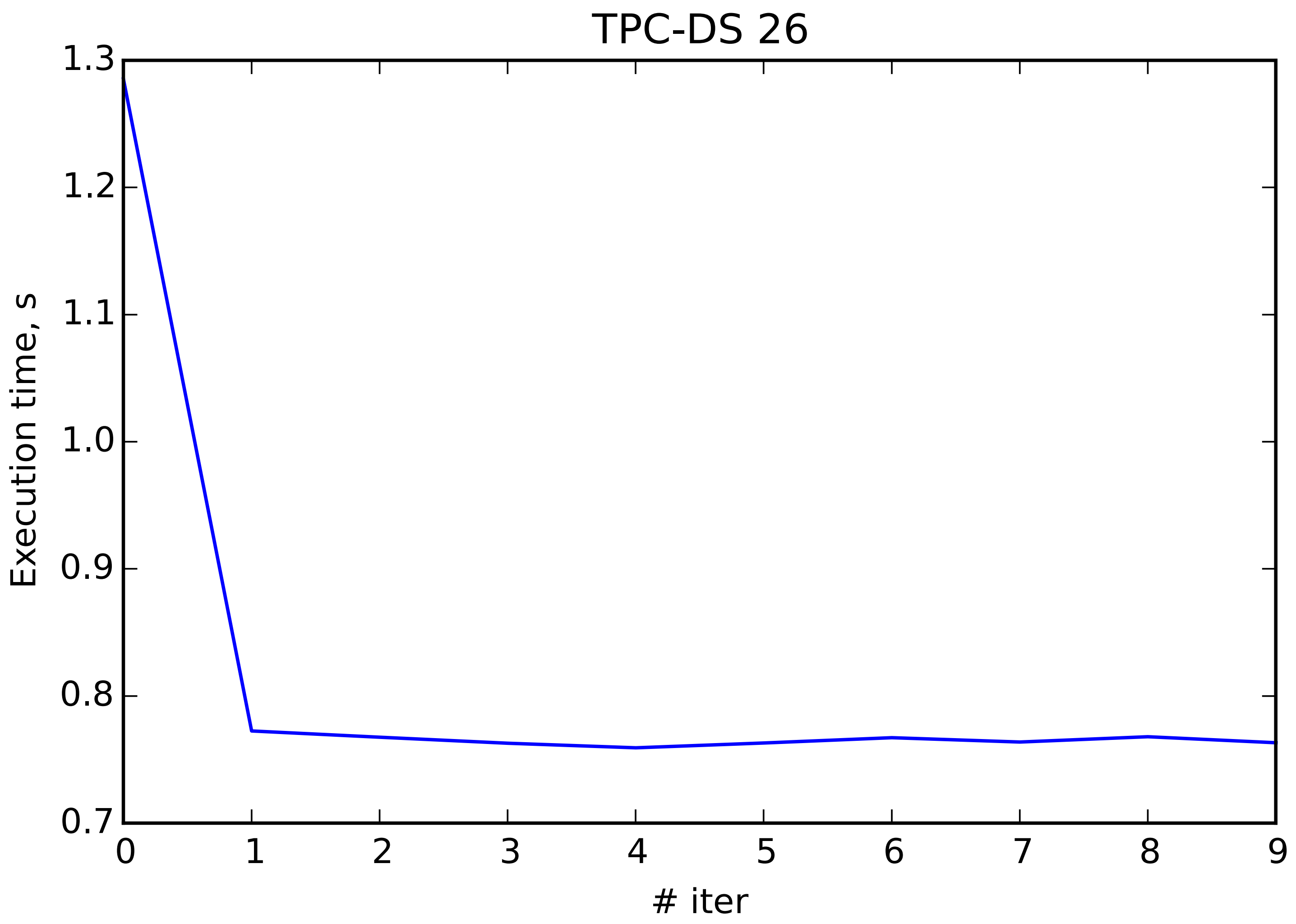
Learning progress



Learning progress



Learning progress



Example: complicated query

How good are query optimizers, really?
V. Leis, A. Gubichev, A. Mirchev, P. Boncz,
A. Kemper, and T. Neumann,
Proc. VLDB, Nov. 2015

Join Order Benchmark

SELECT MIN(k.keyword) **AS** movie_keyword, MIN(n.name) **AS** actor_name, MIN(t.title) **AS** hero_movie
FROM cast_info **AS** ci, keyword **AS** k, movie_keyword **AS** mk, name **AS** n, title **AS** t
WHERE k.keyword in ('superhero', 'sequel', 'second-part', 'marvel-comics', 'based-on-comic', 'tv-special', 'fight', 'violence') **AND** n.name **LIKE** '%Downey%Robert%' **AND** t.production_year > 2000 **AND** k.id = mk.keyword_id **AND** t.id = mk.movie_id **AND** t.id = ci.movie_id **AND** ci.movie_id = mk.movie_id **AND** n.id = ci.person_id;

Bad cardinality estimates

```
                                                  QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=15028.15..15028.16 rows=1 width=96) (actual time=9059.369..9059.369 rows=1 loops=1)
   ->  Nested Loop  (cost=8.21..15028.14 rows=1 width=48) (actual time=218.507..9058.827 rows=88 loops=1)
         ->  Nested Loop  (cost=7.78..12650.75 rows=5082 width=37) (actual time=0.700..3343.348 rows=785477 loops=1)
               Join Filter: (t.id = ci.movie_id)
               ->  Nested Loop  (cost=7.21..12370.11 rows=148 width=41) (actual time=0.682..423.503 rows=14165 loops=1)
                     ->  Nested Loop  (cost=6.78..12235.17 rows=270 width=20) (actual time=0.661..159.090 rows=35548 loops=1)
                           ->  Seq Scan on keyword k  (cost=0.00..3632.40 rows=8 width=20) (actual time=0.125..28.679 rows=8 loops=1)
                                 Filter: (keyword = ANY ('{superhero,sequel,second-part,marvel-comics,based-on-comic,tv-special,fight,violence}'::text[]))
                                 Rows Removed by Filter: 134162
                           ->  Bitmap Heap Scan on movie_keyword mk  (cost=6.78..1072.32 rows=303 width=8) (actual time=0.949..15.307 rows=4444 loops=8)
                                 Recheck Cond: (keyword_id = k.id)
                                 Heap Blocks: exact=23488
                                 ->  Bitmap Index Scan on keyword_id_movie_keyword  (cost=0.00..6.71 rows=303 width=0) (actual time=0.563..0.563 rows=4444 loops=8)
                                       Index Cond: (keyword_id = k.id)
                     ->  Index Scan using title_pkey on title t  (cost=0.43..0.49 rows=1 width=21) (actual time=0.007..0.007 rows=0 loops=35548)
                           Index Cond: (id = mk.movie_id)
                           Filter: (production_year > 2000)
                           Rows Removed by Filter: 1
               ->  Index Scan using movie_id_cast_info on cast_info ci  (cost=0.56..1.47 rows=34 width=8) (actual time=0.010..0.190 rows=55 loops=14165)
                     Index Cond: (movie_id = mk.movie_id)
         ->  Index Scan using name_pkey on name n  (cost=0.43..0.46 rows=1 width=19) (actual time=0.007..0.007 rows=0 loops=785477)
               Index Cond: (id = ci.person_id)
               Filter: (name ~~ '%Downey%Robert%'::text)
               Rows Removed by Filter: 1
 Planning time: 36.697 ms
 Execution time: 9059.593 ms
(26 rows)
```

Using previous statistics to refine estimates

```
                                        QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=88572.37..88572.38 rows=1 width=96) (actual time=16070.666..16070.666 rows=1 loops=1)
   ->  Nested Loop  (cost=8.21..88571.71 rows=88 width=48) (actual time=367.618..16070.206 rows=88 loops=1)
         Join Filter: (mk.movie_id = t.id)
         ->  Nested Loop  (cost=7.78..88571.24 rows=1 width=39) (actual time=367.595..16068.698 rows=112 loops=1)
               ->  Nested Loop  (cost=7.35..84247.78 rows=9242 width=28) (actual time=0.659..5438.176 rows=1564305 loops=1)
                     ->  Nested Loop  (cost=6.78..12235.17 rows=35548 width=20) (actual time=0.642..154.833 rows=35548 loops=1)
                           ->  Seq Scan on keyword k  (cost=0.00..3632.40 rows=8 width=20) (actual time=0.121..28.424 rows=8 loops=1)
                                 Filter: (keyword = ANY ('{superhero,sequel,second-part,marvel-comics,based-on-comic,tv-special,fight,violence}'::text[]))
                                 Rows Removed by Filter: 134162
                           ->  Bitmap Heap Scan on movie_keyword mk  (cost=6.78..1072.32 rows=303 width=8) (actual time=0.925..14.836 rows=4444 loops=8)
                                 Recheck Cond: (keyword_id = k.id)
                                 Heap Blocks: exact=23488
                                 ->  Bitmap Index Scan on keyword_id_movie_keyword  (cost=0.00..6.71 rows=303 width=0) (actual time=0.543..0.543 rows=4444 loops=8)
                                       Index Cond: (keyword_id = k.id)
                     ->  Index Scan using movie_id_cast_info on cast_info ci  (cost=0.56..1.47 rows=55 width=8) (actual time=0.008..0.139 rows=44 loops=35548)
                           Index Cond: (movie_id = mk.movie_id)
               ->  Index Scan using name_pkey on name n  (cost=0.43..0.46 rows=1 width=19) (actual time=0.007..0.007 rows=0 loops=1564305)
                     Index Cond: (id = ci.person_id)
                     Filter: (name ~~ '%Downey%Robert%'::text)
                     Rows Removed by Filter: 1
         ->  Index Scan using title_pkey on title t  (cost=0.43..0.45 rows=1 width=21) (actual time=0.012..0.012 rows=1 loops=112)
               Index Cond: (id = ci.movie_id)
               Filter: (production_year > 2000)
               Rows Removed by Filter: 0
 Planning time: 11.166 ms
 Execution time: 16070.850 ms
(26 rows)
```

```
                                                    QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=169291.60..169291.61 rows=1 width=96) (actual time=3851.890..3851.890 rows=1 loops=1)
   ->  Hash Join  (cost=107718.88..169290.94 rows=88 width=48) (actual time=845.627..3851.419 rows=88 loops=1)
         Hash Cond: (ci.person_id = n.id)
         ->  Nested Loop  (cost=7.78..58633.52 rows=785477 width=37) (actual time=0.700..3011.292 rows=785477 loops=1)
               Join Filter: (t.id = ci.movie_id)
               ->  Nested Loop  (cost=7.21..30001.33 rows=14165 width=41) (actual time=0.682..415.470 rows=14165 loops=1)
                     ->  Nested Loop  (cost=6.78..12235.17 rows=35548 width=20) (actual time=0.663..154.262 rows=35548 loops=1)
                           ->  Seq Scan on keyword k  (cost=0.00..3632.40 rows=8 width=20) (actual time=0.126..28.971 rows=8 loops=1)
                                 Filter: (keyword = ANY ('{superhero,sequel,second-part,marvel-comics,based-on-comic,tv-special,fight,violence}'::text[]))
                                 Rows Removed by Filter: 134162
                           ->  Bitmap Heap Scan on movie_keyword mk  (cost=6.78..1072.32 rows=303 width=8) (actual time=0.980..14.743 rows=4444 loops=8)
                                 Recheck Cond: (keyword_id = k.id)
                                 Heap Blocks: exact=23488
                                 ->  Bitmap Index Scan on keyword_id_movie_keyword  (cost=0.00..6.71 rows=303 width=0) (actual time=0.579..0.579 rows=4444 loops=8)
                                       Index Cond: (keyword_id = k.id)
                     ->  Index Scan using title_pkey on title t  (cost=0.43..0.49 rows=1 width=21) (actual time=0.007..0.007 rows=0 loops=35548)
                           Index Cond: (id = mk.movie_id)
                           Filter: (production_year > 2000)
                           Rows Removed by Filter: 1
               ->  Index Scan using movie_id_cast_info on cast_info ci  (cost=0.56..1.47 rows=44 width=8) (actual time=0.009..0.170 rows=55 loops=14165)
                     Index Cond: (movie_id = mk.movie_id)
         ->  Hash  (cost=107705.93..107705.93 rows=414 width=19) (actual time=756.785..756.785 rows=2 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on name n  (cost=0.00..107705.93 rows=414 width=19) (actual time=77.074..756.771 rows=2 loops=1)
                     Filter: (name ~~ '%Downey%Robert%'::text)
                     Rows Removed by Filter: 4167489
 Planning time: 13.462 ms
 Execution time: 3852.110 ms
(28 rows)
```

```
                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=109869.39..109869.40 rows=1 width=96) (actual time=783.543..783.543 rows=1 loops=1)
   ->  Nested Loop  (cost=1.85..109868.73 rows=88 width=48) (actual time=93.291..783.412 rows=88 loops=1)
         ->  Nested Loop  (cost=1.43..109849.92 rows=41 width=36) (actual time=92.005..767.675 rows=5202 loops=1)
               ->  Nested Loop  (cost=0.99..109833.44 rows=9 width=40) (actual time=91.982..763.536 rows=306 loops=1)
                     ->  Nested Loop  (cost=0.56..109825.54 rows=17 width=19) (actual time=91.907..758.112 rows=486 loops=1)
                           ->  Seq Scan on name n  (cost=0.00..107705.93 rows=2 width=19) (actual time=73.336..732.998 rows=2 loops=1)
                                 Filter: (name ~~ '%Downey%Robert%'::text)
                                 Rows Removed by Filter: 4167489
                           ->  Index Scan using person_id_cast_info on cast_info ci  (cost=0.56..1054.82 rows=499 width=8) (actual time=12.418..12.503 rows=243 loops=2)
                                 Index Cond: (person_id = n.id)
                     ->  Index Scan using title_pkey on title t  (cost=0.43..0.45 rows=1 width=21) (actual time=0.011..0.011 rows=1 loops=486)
                           Index Cond: (id = ci.movie_id)
                           Filter: (production_year > 2000)
                           Rows Removed by Filter: 0
               ->  Index Scan using movie_id_movie_keyword on movie_keyword mk  (cost=0.43..1.36 rows=47 width=8) (actual time=0.007..0.010 rows=17 loops=306)
                     Index Cond: (movie_id = t.id)
         ->  Index Scan using keyword_pkey on keyword k  (cost=0.42..0.45 rows=1 width=20) (actual time=0.003..0.003 rows=0 loops=5202)
               Index Cond: (id = mk.keyword_id)
               Filter: (keyword = ANY ('{superhero,sequel,second-part,marvel-comics,based-on-comic,tv-special,fight,violence}'::text[]))
               Rows Removed by Filter: 1
 Planning time: 13.981 ms
 Execution time: 783.723 ms
(22 rows)
```

```
                                                      QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=110747.71..110747.72 rows=1 width=96) (actual time=770.231..770.232 rows=1 loops=1)
   ->  Nested Loop  (cost=1.85..110747.05 rows=88 width=48) (actual time=78.828..770.093 rows=88 loops=1)
         Join Filter: (mk.movie_id = t.id)
         ->  Nested Loop  (cost=1.42..110694.70 rows=112 width=39) (actual time=75.208..769.518 rows=112 loops=1)
               ->  Nested Loop  (cost=1.00..110660.75 rows=74 width=27) (actual time=74.639..743.329 rows=10066 loops=1)
                     ->  Nested Loop  (cost=0.56..109820.42 rows=486 width=19) (actual time=74.589..736.659 rows=486 loops=1)
                           ->  Seq Scan on name n  (cost=0.00..107705.93 rows=2 width=19) (actual time=74.543..736.376 rows=2 loops=1)
                                 Filter: (name ~~ '%Downey%Robert%'::text)
                                 Rows Removed by Filter: 4167489
                           ->  Index Scan using person_id_cast_info on cast_info ci  (cost=0.56..1054.82 rows=243 width=8) (actual time=0.027..0.094 rows=243 loops=2)
                                 Index Cond: (person_id = n.id)
                     ->  Index Scan using movie_id_movie_keyword on movie_keyword mk  (cost=0.43..1.26 rows=47 width=8) (actual time=0.006..0.010 rows=21 loops=486)
                           Index Cond: (movie_id = ci.movie_id)
               ->  Index Scan using keyword_pkey on keyword k  (cost=0.42..0.45 rows=1 width=20) (actual time=0.002..0.002 rows=0 loops=10066)
                     Index Cond: (id = mk.keyword_id)
                     Filter: (keyword = ANY ('{superhero,sequel,second-part,marvel-comics,based-on-comic,tv-special,fight,violence}'::text[]))
                     Rows Removed by Filter: 1
         ->  Index Scan using title_pkey on title t  (cost=0.43..0.45 rows=1 width=21) (actual time=0.005..0.005 rows=1 loops=112)
               Index Cond: (id = ci.movie_id)
               Filter: (production_year > 2000)
               Rows Removed by Filter: 0
 Planning time: 14.306 ms
 Execution time: 770.452 ms
(23 rows)
```

Convergence to the plan with good estimates

```
                                                   QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=112898.09..112898.10 rows=1 width=96) (actual time=754.243..754.243 rows=1 loops=1)
   ->  Nested Loop  (cost=1.85..112897.43 rows=88 width=48) (actual time=75.655..754.117 rows=88 loops=1)
         ->  Nested Loop  (cost=1.43..110510.89 rows=5202 width=36) (actual time=74.710..739.330 rows=5202 loops=1)
               Join Filter: (t.id = mk.movie_id)
               ->  Nested Loop  (cost=0.99..110046.39 rows=306 width=40) (actual time=74.694..735.745 rows=306 loops=1)
                     ->  Nested Loop  (cost=0.56..109820.42 rows=486 width=19) (actual time=74.648..732.892 rows=486 loops=1)
                           ->  Seq Scan on name n  (cost=0.00..107705.93 rows=2 width=19) (actual time=74.602..732.599 rows=2 loops=1)
                                 Filter: (name ~~ '%Downey%Robert%'::text)
                                 Rows Removed by Filter: 4167489
                           ->  Index Scan using person_id_cast_info on cast_info ci  (cost=0.56..1054.82 rows=243 width=8) (actual time=0.027..0.097 rows=243 loops=2)
                                 Index Cond: (person_id = n.id)
                     ->  Index Scan using title_pkey on title t  (cost=0.43..0.45 rows=1 width=21) (actual time=0.005..0.006 rows=1 loops=486)
                           Index Cond: (id = ci.movie_id)
                           Filter: (production_year > 2000)
                           Rows Removed by Filter: 0
               ->  Index Scan using movie_id_movie_keyword on movie_keyword mk  (cost=0.43..1.26 rows=21 width=8) (actual time=0.004..0.008 rows=17 loops=306)
                     Index Cond: (movie_id = ci.movie_id)
         ->  Index Scan using keyword_pkey on keyword k  (cost=0.42..0.45 rows=1 width=20) (actual time=0.003..0.003 rows=0 loops=5202)
               Index Cond: (id = mk.keyword_id)
               Filter: (keyword = ANY ('{superhero,sequel,second-part,marvel-comics,based-on-comic,tv-special,fight,violence}'::text[]))
               Rows Removed by Filter: 1
 Planning time: 15.498 ms
 Execution time: 754.449 ms
(23 rows)
```

Conclusion

Adaptive Query Optimization

Uses stored statistics to refine cardinality estimates

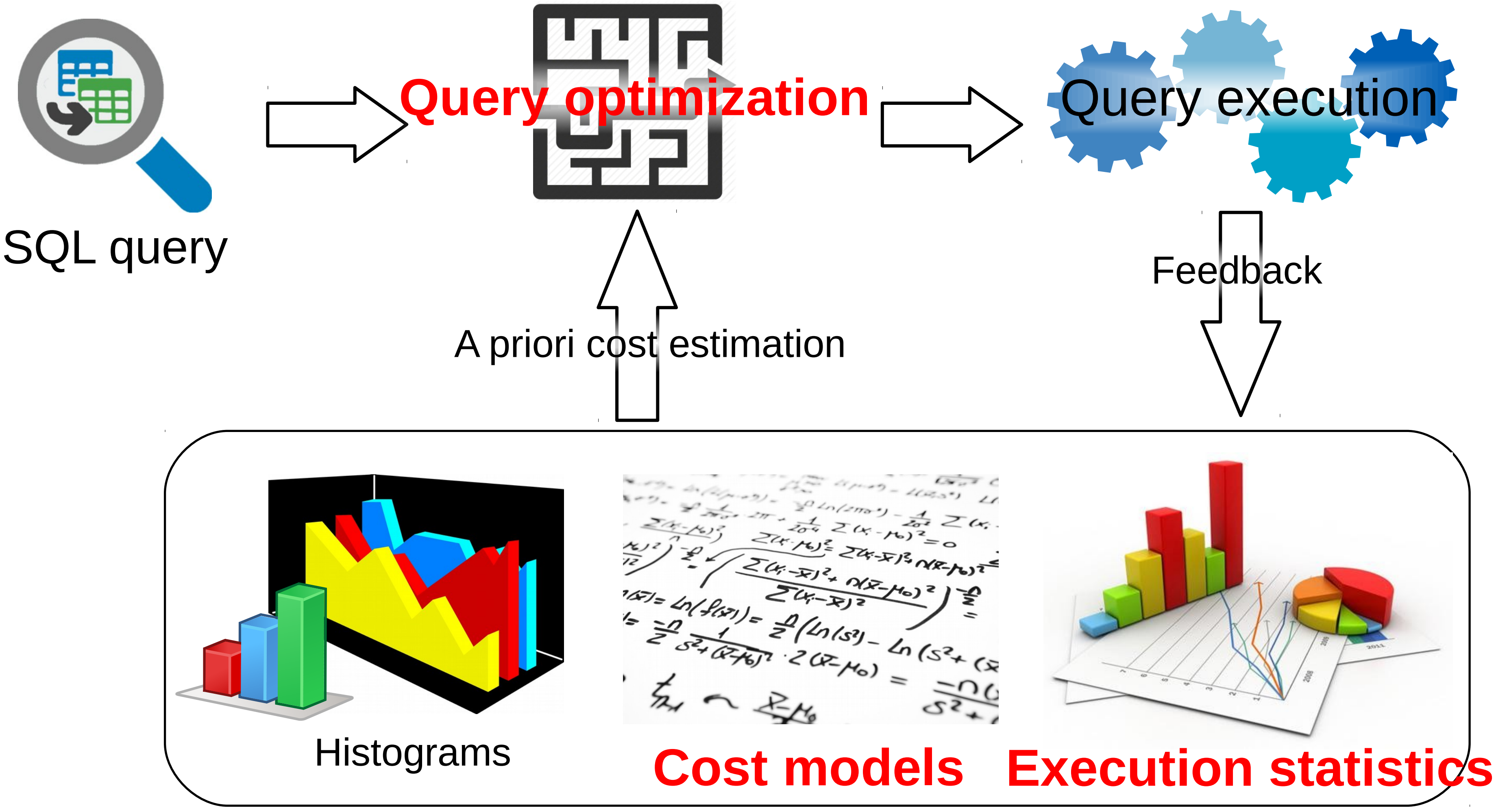
Works for clauses of the same structure (i. e. *age* < some_const)

Works when data distribution doesn't change rapidly

Is suitable for the static workload (i. e. the finite number of query templates or clause structures)

Is useful for complicated queries of the same structure with slow plan caused by bad cardinality estimates (OLAP)

Further work



Questions

Contacts:

- o.ivanov@postgrespro.ru
- +7 (916) 377-55-63

Postgres Professional

http://postgrespro.ru/

+7 495 150 06 91

info@postgrespro.ru

postgrespro.ru