Adaptive Query Optimization
Approaches and Challenges
What is Adaptive Query Optimization?
How does DBMS execute a query?

SQL Query → ... → Result
SQL query

SELECT *
FROM students, departments
WHERE students.GPA > 4
AND departments.name = 'Data Science'
AND students.department_id = departments.id;
How does DBMS execute a query?

SQL Query → Query Optimization → ... → Result

Data Statistics
Query plan

**Hash Join** (cost=25.95..28.42 rows=1 width=59)
  Hash Cond: (students.department_id = departments.id)
  -> **Seq Scan** on students (cost=0.00..2.24 rows=89 width=23)
    Filter: (gpa > '4'::double precision)
  -> **Hash** (cost=25.88..25.88 rows=6 width=36)
    -> **Seq Scan** on departments (cost=0.00..25.88 rows=6 width=36)
      Filter: (name = 'Data Science'::text)
How does DBMS execute a query?

SQL Query → Query Optimization → Query Execution → Result

Data Statistics

Query Execution Statistics
Hash Join (cost=25.95..28.42 rows=1 width=59) (actual time=0.061..0.147 rows=41 loops=1)
  Hash Cond: (students.department_id = departments.id)
  ->  Seq Scan on students (cost=0.00..2.24 rows=89 width=23) (actual time=0.018..0.058 rows=92 loops=1)
      Filter: (gpa > '4':double precision)
      Rows Removed by Filter: 9
  ->  Hash (cost=25.88..25.88 rows=6 width=36) (actual time=0.014..0.014 rows=1 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 9kB
      ->  Seq Scan on departments (cost=0.00..25.88 rows=6 width=36) (actual time=0.007..0.009 rows=1 loops=1)
          Filter: (name = 'Data Science':text)
          Rows Removed by Filter: 1
Adaptive Query Optimization is the red line
How does Query Optimization work?
Cost-based Query Optimization

CostFunction

Plan

Hash Join (cost=25.95..28.42 rows=1 width=59)
  Hash Cond: (students.department_id = departments.id)
  -> Seq Scan on students (cost=0.00..2.24 rows=89 width=23)
    Filter: (gpa > '4'::double precision)
  -> Hash (cost=25.88..25.88 rows=6 width=36)
    -> Seq Scan on departments (cost=0.00..25.88 rows=6 width=36)
      Filter: (name = 'Data Science'::text)

Cost

28.42
BestPlan = \text{ArgMin} \left( \text{CostFunction} \left( \text{Plan} \right) \right)

\text{CostFunction is additive: } \square + \square + \square = \square

\begin{tabular}{|l|}
\hline
\textbf{Hash Join} \ (\text{cost}=25.95..28.42 \text{ rows}=1 \text{ width}=59) \\
Hash \ Cond: \ (\text{students.department_id} = \text{departments.id}) \\
\hline
\text{-> Seq Scan on students} \ (\text{cost}=0.00..2.24 \text{ rows}=89 \text{ width}=23) \\
Filter: \ (\text{gpa} > '4'::\text{double precision}) \\
\hline
\text{-> Hash} \ (\text{cost}=25.88..25.88 \text{ rows}=6 \text{ width}=36) \\
\text{-> Seq Scan on departments} \ (\text{cost}=0.00..25.88 \text{ rows}=6 \text{ width}=36) \\
Filter: \ (\text{name} = 'Data Science'::\text{text}) \\
\hline
\end{tabular}
CostFunction ( Plan )

Cost: \( C_1 N + C_2 N \log N \)

Number of tuples:
- Sort: \( N \)
- MergeJoin: \( N_1, N_2 \)
- NestedLoop: \( N_1, N_2 \)

Cost: \( C (N_1 + N_2) \)

Cost: \( C N_1 N_2 \)
The main problem is in Cardinality Estimation

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

Adaptive Cardinality Estimation
O. Ivanov, S. Bartunov,
Arxiv, Nov. 2017
Cardinality Estimation: no clauses

| Table                  | Cardinality is known |
Cardinality Estimation: one clause

```
SELECT *
FROM students
WHERE students.GPA > 4;
```
Cardinality Estimation: two or more clauses

Table + Clause1
+ Clause2

SELECT *
FROM students
WHERE students.name = 'Igor'
AND students.gender = 'Female';

SELECT *
FROM students
WHERE students.name = 'Igor'
AND students.gender = 'Male';
PostgreSQL default assumption: clauses independence

```
SELECT *
FROM students
WHERE students.name = 'Igor'
AND students.department_name = 'Data Science';
```
Multidimensional Histograms (aka Cross-Column Statistics)
Query Optimization: conclusions

+ OLTP
- OLAP
- ORM (sometimes)
Adaptive Cardinality Estimation
What information do we have?

**Hash Join** (actual time=0.061..0.147 rows=41 loops=1)
Hash Cond: (students.department_id = departments.id)
-\> **Seq Scan** on students (actual time=0.018..0.058 rows=92 loops=1)
  Filter: (gpa > '4':double precision)
  Rows Removed by Filter: 9
-\> **Hash** (actual time=0.014..0.014 rows=1 loops=1)
  Buckets: 1024  Batches: 1  Memory Usage: 9kB
-\> **Seq Scan** on departments (actual time=0.007..0.009 rows=1 loops=1)
  Filter: (name = 'Data Science':text)
  Rows Removed by Filter: 1
Information in one relation

```
students  hashing  fd528b449f20
```
Information in one clause

GPA > 4 \rightarrow \text{normalization} \rightarrow \text{GPA} > \text{const} \rightarrow \text{hashing} \rightarrow \text{a61468a206f}

\text{PostgreSQL selectivity estimation}

0.3 \rightarrow \text{log} \rightarrow -1.20
One plan node = Object

<table>
<thead>
<tr>
<th>Clauses info</th>
<th>Relations hashes</th>
<th>Log Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a61468a206ff, -1.20)</td>
<td>fd528b449f20</td>
<td>7.08</td>
</tr>
<tr>
<td>(654101251a11, -9.21)</td>
<td>27cb8c47b899</td>
<td></td>
</tr>
<tr>
<td>(9141e9f5e5f5, -0.11)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
K Nearest Neighbours

Similarity between objects

<table>
<thead>
<tr>
<th>65410f251a11</th>
<th>fd528b449f20</th>
<th>5.32</th>
</tr>
</thead>
<tbody>
<tr>
<td>-9.21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9141e9f5e5f5</th>
<th>fd528b449f20</th>
<th>2.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Incomparable
K Nearest Neighbours

Similarity between objects

<table>
<thead>
<tr>
<th>9141e9f5e5f5</th>
<th>a61468a206ff</th>
<th>fd528b449f20</th>
<th>5.32</th>
</tr>
</thead>
<tbody>
<tr>
<td>-9.21</td>
<td>-1.20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9141e9f5e5f5</th>
<th>a61468a206ff</th>
<th>fd528b449f20</th>
<th>7.44</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8.67</td>
<td>-0.11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Distance = $||(-9.21, -1.20) - (-8.67, -0.11)||$

Similarity = $1 / (\text{Distance} + \text{EPS})$
K Nearest Neighbours

Stored objects

<table>
<thead>
<tr>
<th>Object 1</th>
<th>Object 2</th>
<th>Object 3</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>9141e9f5e5f5</td>
<td>a61468a206ff</td>
<td>fd528b449f20</td>
<td>5.32</td>
</tr>
<tr>
<td>9141e9f5e5f5</td>
<td>a61468a206ff</td>
<td>fd528b449f20</td>
<td>7.44</td>
</tr>
<tr>
<td>65410f251a11</td>
<td>fd528b449f20</td>
<td></td>
<td>3.17</td>
</tr>
<tr>
<td>9141e9f5e5f5</td>
<td>fd528b449f20</td>
<td></td>
<td>2.75</td>
</tr>
</tbody>
</table>

S = 1.67

S = 0.5

Idea: cardinalities for similar objects must be similar.

\[
\text{Weight}_1 = \frac{S_1}{S_1 + S_2} = \frac{1.67}{1.67 + 0.5} = 0.77
\]

\[
? = 5.32 \times 0.77 + 7.44 \times 0.23 = 5.80
\]
## K Nearest Neighbours

### Stored objects

<table>
<thead>
<tr>
<th>ID1</th>
<th>ID2</th>
<th>ID3</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>65410f251a11</td>
<td>fd528b449f20</td>
<td>3.17</td>
<td></td>
</tr>
<tr>
<td>9141e9f5e5f5</td>
<td>a61468a206ff</td>
<td>9141e9f5e5f5</td>
<td>5.32</td>
</tr>
<tr>
<td>9141e9f5e5f5</td>
<td>a61468a206ff</td>
<td>fd528b449f20</td>
<td>7.44</td>
</tr>
<tr>
<td>9141e9f5e5f5</td>
<td>fd528b449f20</td>
<td>2.75</td>
<td></td>
</tr>
</tbody>
</table>

### New object

<table>
<thead>
<tr>
<th>ID1</th>
<th>ID2</th>
<th>ID3</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>6f410f251a11</td>
<td>a61468a206ff</td>
<td>fd528b449f20</td>
<td>?</td>
</tr>
</tbody>
</table>

No similar objects -> use PostgreSQL estimator
K Nearest Neighbours

Number of stored objects of each type is limited

<table>
<thead>
<tr>
<th>9141e9f5e5f5</th>
<th>a61468a206ff</th>
<th>fd528b449f20</th>
<th>5.32</th>
</tr>
</thead>
<tbody>
<tr>
<td>-9.21</td>
<td>-1.19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9141e9f5e5f5</th>
<th>a61468a206ff</th>
<th>fd528b449f20</th>
<th>7.44</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8.71</td>
<td>-0.11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>65410f251a11</th>
<th>fd528b449f20</th>
<th></th>
<th>3.17</th>
</tr>
</thead>
<tbody>
<tr>
<td>-9.21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9141e9f5e5f5</th>
<th>fd528b449f20</th>
<th></th>
<th>2.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
K Nearest Neighbours: Results

➢ TPC-DS*:
  ○ Maximum performance boost is 95x
  ○ Mean execution time decreased by 50%

➢ Join Order Benchmark:
  ○ Maximum performance boost is 6.2x
  ○ Mean execution time decreased by 30%

➢ Rarely AQO slows down query (errors in optimizator)
➢ Theoretical guarantees for convergence
Planning Time: 51.227 ms  
Execution Time: 913.359 ms
Finalize Aggregate (cost=45364.09..45364.10 rows=1 width=96) (actual time=441.826..444.935 rows=1 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Partial Aggregate (cost=44363.86..44363.87 rows=1 width=96) (actual time=437.915..437.920 rows=1 loops=3)
    Hash Join (cost=4800.09..44016.67 rows=46292 width=65) (actual time=138.523..428.773 rows=37034 loops=3)
    Hash Cond: (t.kind_id = kt.id)
    ->  Nested Loop (cost=4798.99..44015.25 rows=76 width=69) (actual time=71.000..422.457 rows=3 loops=3)
      Hash Join (cost=4798.56..43980.32 rows=76 width=60) (actual time=70.982..347.084 rows=44730 loops=3)
      Hash Cond: (mi.info_type_id = it2.id)
    ->  Nested Loop (cost=4795.70..43433.79 rows=274 width=14) (actual time=70.902..196.418 rows=23765 loops=3)
      Hash Join (cost=4795.30..43347.09 rows=30943 width=18) (actual time=70.814..149.115 rows=49377 loops=3)
      Hash Cond: (miidx.info_type_id = it.id)
    ->  Nested Loop (cost=4793.27..43437.09 rows=30943 width=18) (actual time=70.880..193.050 rows=31523 loops=3)
      Hash Join (cost=4792.84..37544.66 rows=10990 width=4) (actual time=70.814..149.115 rows=49377 loops=3)
      Hash Cond: (mc.company_type_id = ct.id)
    Parallel Hash Join (cost=4791.78..37305.94 rows=43958 width=8) (actual time=5.751..144.378 rows=32583 loops=3)
    Hash Cond: (mc.company_id = cn.id)
  Parallel Seq Scan on movie_companies mc (cost=0.00..29660.37 rows=1087137 width=12) (actual time=0.012..56.202 rows=869710 loops=3)
  Parallel Hash (cost=4721.92..4721.92 rows=5589 width=4) (actual time=0.007..7.826 rows=32583 loops=3)
  Buckets: 16384  Batches: 1  Memory Usage: 544kB
  Parallel Index Scan on movie_id_movie_info_idx miidx (cost=0.43..0.50 rows=3 width=14) (actual time=0.001..0.002 rows=1 loops=1)
  Index Cond: (movie_id = mc.movie_id)
  Planning Time: 54.386 ms
Execution Time: 446.338 ms
Planning Time: 52.883 ms
Execution Time: 800.859 ms
Finalize Aggregate (cost=72802.97..72802.98 rows=1 width=96) (actual time=497.952..498.077 rows=1 loops=1)
  Workers Planned: 2
  Workers Launched: 2
    -> Partial Aggregate (cost=71802.75..71802.76 rows=1 width=96) (actual time=494.947..494.952 rows=1 loops=3)
      Hash Cond: (miidx.info_type_id = it.id)
      Hash Cond: (mi.info_type_id = it2.id)
    -> Nested Loop (cost=4796.39..62985.55 rows=29706 width=29) (actual time=72.763..213.854 rows=520885 loops=3)
      Join Filter: (t.kind_id = kt.id)
      Hash Cond: (t.kind_id = kt.id)
      Hash Cond: (mc.company_type_id = ct.id)
      Hash Cond: (mc.company_id = cn.id)
-> Hash Join (cost=4794.79..37377.76 rows=61872 width=8) (actual time=10.158..144.766 rows=49377 loops=3)
  Buckets: 16384  Batches: 1  Memory Usage: 576kB
Index Scan using title_pkey on title t (cost=0.43..0.52 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=12889)
  Index Cond: (id = mc.movie_id)
-> Hash (cost=2.41..2.41 rows=1 width=4) (actual time=0.006..0.007 rows=1 loops=3)
Finalize Aggregate (cost=73834.19..73834.20 rows=1 width=96) (actual time=356.744..359.896 rows=1 loops=1)
   -> Gather (cost=73833.97..73834.18 rows=2 width=96) (actual time=356.724..359.890 rows=3 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial Aggregate (cost=72833.97..72833.98 rows=1 width=96) (actual time=353.917..353.922 rows=1 loops=3)
         -> Hash Join (cost=4802.10..72486.78 rows=46292 width=65) (actual time=119.148..345.717 rows=37034 loops=3)
            Hash Cond: (mi.info_type_id = it2.id)
            -> Nested Loop (cost=4799.67..71898.44 rows=215136 width=69) (actual time=119.012..333.552 rows=172109 loops=3)
               Join Filter: (t.id = mi.movie_id)
               -> Hash Join (cost=4794.86..37642.70 rows=29706 width=14) (actual time=73.310..148.577 rows=23765 loops=3)
                  Hash Cond: (miidx.info_type_id = it.id)
                  -> Nested Loop (cost=4792.92..72191.92 rows=5750 width=4) (actual time=73.189..138.988 rows=49377 loops=3)
                     Hash Join (cost=4791.71..52842.31 rows=13129 width=14) (actual time=73.310..138.988 rows=49377 loops=3)
                        Hash Cond: (mc.company_id = cn.id)
                        -> Parallel Hash Join (cost=4790.69..37607.96 rows=61722 width=8) (actual time=10.114..143.898 rows=49377 loops=3)
                           Hash Cond: (mc.company_type_id = ct.id)
                           -> Parallel Seq Scan on movie_companies mc (cost=0.00..29660.37 rows=1087137 width=12) (actual time=73.310..143.896 rows=49377 loops=3)
                              Filter: ((country_code)::text = '[de]'::text)
                              Rows Removed by Filter: 75074
                              Buckets: 16384 Batches: 1 Memory Usage: 769kB
                              -> Parallel Seq Scan on company_name cn (cost=4791.71..52842.31 rows=13129 width=14) (actual time=73.189..138.988 rows=49377 loops=3)
                                 Filter: ((country_code)::text = '[de]'::text)
                                 Rows Removed by Filter: 75074
                                 Buckets: 16384 Batches: 1 Memory Usage: 769kB
                                 -> Index Scan using movie_id_movie_info_idx on movie_info_idx miidx (cost=4791.71..52842.31 rows=13129 width=14) (actual time=73.189..138.988 rows=49377 loops=3)
                                    Filter: ((country_code)::text = '[de]'::text)
                                    Rows Removed by Filter: 75074
                                    Buckets: 16384 Batches: 1 Memory Usage: 769kB
                                    -> Parallel Hash (cost=4791.71..52842.31 rows=13129 width=14) (actual time=73.189..138.988 rows=49377 loops=3)
                                       BUCKETS 16384 Batches 1 Memory Usage 576kB
                                       Filter: ((country_code)::text = '[de]'::text)
                                       Rows Removed by Filter: 75074
      Planning Time: 54.904 ms
      Execution Time: 361.197 ms
AQO now

- Have a production code: [https://github.com/postgrespro/aqo](https://github.com/postgrespro/aqo)
- Is a part of PostgresPro Enterprise
- Helps DBA to find a better plan for the query
Adaptive Cardinality Estimation: ongoing research
K Nearest Neighbours: challenges

➢ Memory consumption
➢ Stability = low generalization ability = local optima
# Neural Networks

<table>
<thead>
<tr>
<th>Clauses info</th>
<th>Relations hashes</th>
<th>Log Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a61468a206ff, -1.20)</td>
<td>fd528b449f20</td>
<td>7.08</td>
</tr>
<tr>
<td>(654101251a11, -9.21)</td>
<td>27cb8c47b899</td>
<td></td>
</tr>
<tr>
<td>(9141e9f5e5f5, -0.11)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Neural Networks
Neural Networks
Neural Networks: Results

- Join Order Benchmark*:
  - Maximum performance boost is 2x
  - Mean execution time decreased by 70%

- Join Order Benchmark* - joint:
  - Maximum performance boost is 2x
  - Mean execution time decreased by 40%
Neural Networks: Results
Neural Networks:

➢ are less stable
➢ generalize better and find better plans
➢ can use information from other queries
Challenges
Challenges: low level

- Better regressor
- Better support for complex clauses
- “Learnable histograms” for uncommon operators and data types
- Utilization of rolled back queries statistics
Challenges: high level

Goal: to select the best plan *automatically*

Means:

- Meta-optimization
- Plans space exploration
- Interruption and re-execution of bad plans
- ...
Major Contributors

Company where AQO was created and evolved

Oleg Bartunov
Vision, Idea

Oleg Ivanov
Idea, Research, First version of the code

Andrey Lepikhov
Production code refactoring and support

Yerzhaisang Taskali
AQO-NN research
Takeaways

- Cardinality estimation errors are common in OLAP workloads
- AQO uses query execution statistics for query optimization
- AQO implementation is available for everyone
  - It usually provides significant speed up for complex OLAP queries
- AQO is a framework
  - There is still a lot of work to do
    - In progress...