Adaptive query optimization in PostgreSQL

Oleg Ivanov
Postgres Professional
Outline

• Problem statement
  – Query optimization
  – Correlated clauses issue

• Machine learning
  – Gradient K Nearest Neighbours method

• Adaptive query optimization
  – Theory
  – Implementation

• Experimental evaluation
Query optimization
The SQL query is:

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

Plan node execution cost
Plan node cardinality
Plan execution cost

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)
How does PostgreSQL optimize queries?

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

\[ \text{Cost} = n_s c_s + n_r c_r + n_t c_t + n_i c_i + n_o c_o \]

| \( c_s \) | 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

```sql
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
- Genetic algorithm

Plan's cost estimation

Cardinality estimation

Cost model

Cost = 439429

Cost = 304528
Correlated clauses issue
Query clauses

Cardinality estimation

Information about stored data

Cost model

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
SELECT * FROM users
WHERE age < 25;

Selectivity ≈ 0.3
Cardinality = \( N_{\text{tuples}} \cdot \text{Selectivity} \)
SELECT * FROM users
WHERE age < 25 AND city = 'Ottawa';

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

\[
\text{Selectivity}_{\text{age}} = \frac{1}{3}
\]

\[
\text{Selectivity}_{\text{city}} = \frac{1}{7}
\]

\[
\text{Selectivity}_{\text{age, city}} = ?
\]
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 \( Selectivity_{25<age\ AND\ age<57} = Selectivity_{25<age<57} \)
SELECT * FROM users
WHERE age < 12 AND married = true;
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

Contrasts:
- 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

SQL query → Query optimization → Query execution → Result

A priori cost estimation

Histograms → Cost models
Adaptive query optimization: idea

1. SQL query
2. Query optimization
3. Query execution
4. Result

- A priori cost estimation
- Feedback

- Histograms
- Cost models
- Execution statistics
Adaptive query optimization: idea

SQL query → Query optimization → Query execution

A priori cost estimation → Feedback → Result

Histories → Cost models → Execution statistics
Adaptive query optimization: idea

- SQL query
- Query optimization
  - A priori cost estimation
- Query execution
  - Feedback
- Result

- Histograms
- Cost models
- Execution statistics
Machine learning
Machine learning

Features
- Feature 1
- Feature 2
- Feature 3

Hidden variables

Objects

Train set

Test set
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

<table>
<thead>
<tr>
<th>Age</th>
<th>25</th>
<th>47</th>
<th>55</th>
<th>32</th>
<th>22</th>
<th>45</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salary</td>
<td>50</td>
<td>120</td>
<td>100</td>
<td>80</td>
<td>30</td>
<td>90</td>
<td>?</td>
</tr>
</tbody>
</table>
### K Nearest Neighbours method

<table>
<thead>
<tr>
<th>Age</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>47</td>
<td>120</td>
</tr>
<tr>
<td>55</td>
<td>100</td>
</tr>
<tr>
<td>32</td>
<td>80</td>
</tr>
<tr>
<td>22</td>
<td>30</td>
</tr>
<tr>
<td>45</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>27</td>
<td>47</td>
<td>28</td>
</tr>
<tr>
<td>Salary</td>
<td>53</td>
<td>103</td>
<td>?</td>
</tr>
</tbody>
</table>
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

10
13
12
15
20
17
Gradient approach to kNN

$K = 3$
Gradient approach to kNN

$K = 3$
Adaptive query optimization

Theory
The object is a node with its subtree

- **NestedLoopJoin**
  - **u1.id = messages.sender_id**
  - **u2.id = messages.receiver_id**
  - **u2.married = true**
  - **u2.age < 25**
Histograms

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

Node cardinality is 105 tuples!

PostgreSQL estimator
Node cardinality is 1017 tuples!

Clause selectivities
- 0.0001
- 0.73
- 0.23

Relations to join
- users
- messages

Clauses list

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

*equal clauses must be handled specially: transform into clause with an equivalence class as an argument
Machine learning problem statement

**Base relations**
- users
- messages

**Features (clause types)**
- users.id = messages.receiver_id
- users.married = **const**
- users.age < **const**

**Hidden variable**
- Node cardinality

Object is a plan node

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

1. Query parsing
2. Query optimization
3. Query execution

Machine learning data

- Cardinality estimation
- Learning

Query execution statistics

Machine learning
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
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:
```sql
SELECT * FROM users WHERE age > const AND city = const;
```

Queries:
```sql
SELECT * FROM users WHERE age > 18 AND city = 'Ottawa';
SELECT * FROM users WHERE age > 65 AND city = 'New York';
...
## AQO tables

<table>
<thead>
<tr>
<th>aqoQueries</th>
<th>aqoQueryTexts</th>
<th>aqoData</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Query_hash</td>
<td>• Query_hash</td>
<td>• Feature_space</td>
</tr>
<tr>
<td>• Learn AQO</td>
<td>• Query_text</td>
<td>• Feature_subspace</td>
</tr>
<tr>
<td>• Use AQO</td>
<td></td>
<td>• Features</td>
</tr>
<tr>
<td>• Feature_space</td>
<td></td>
<td>• Target</td>
</tr>
</tbody>
</table>

**Settings**

**For user**

**Machine learning**
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

<table>
<thead>
<tr>
<th><strong>aqo_queries</strong></th>
<th><strong>aqo_query_texts</strong></th>
<th><strong>aqo_data</strong></th>
<th><strong>aqo_query_stat</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Query_hash</td>
<td>Query_hash</td>
<td>Feature_space</td>
<td>Query_hash</td>
</tr>
<tr>
<td>Learn AQO</td>
<td>Query_text</td>
<td>Feature_subspace</td>
<td>Planning time</td>
</tr>
<tr>
<td>Use AQO</td>
<td></td>
<td>Features</td>
<td>Execution time</td>
</tr>
<tr>
<td>Feature_space</td>
<td></td>
<td>Target</td>
<td>Cardinality error</td>
</tr>
<tr>
<td>Auto_tuning</td>
<td></td>
<td></td>
<td>Number of executions</td>
</tr>
</tbody>
</table>

**Settings**      
**For user**       
**Machine learning**  
**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

Machine learning

Cardinality estimation

Learning

Query execution statistics

Machine learning data
Fair workflow

Query parsing → Query optimization → Query execution

Query execution statistics

Query settings definition → Cardinality estimation → Learning → Auto tuning

aqo_queries (cached) → aqo_data → aqo_query_stat

aqo_query_texts → aqo_queries (cached)
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

Number of queries

- Original PostgreSQL
- Linear regression
- Neural network
- K nearest neighbours
- Gradient boosting
Cardinality estimation error

Number of queries vs. error for different methods:
- Blue: Original PostgreSQL
- Black: Gradient k nearest neighbours
- Pink: k nearest neighbours limited
- Red: K nearest neighbours

Error decreases as the number of queries increases.
Cardinality estimation error

Number of queries

- Original PostgreSQL
- Neural network
- Gradient k nearest neighbours
- K nearest neighbours
- Gradient boosting
Performance improvement

TPC-H fast

+1.3%

TPC-H slow
Performance improvement

TPC-H fast

TPC-H slow

-4.4%
Performance improvement

TPC-DS very fast
TPC-DS fast
TPC-DS normal
TPC-DS slow
TPC-DS very slow

Original
Adaptive

+12%
Performance improvement

Adaptive vs Original

- TPC-DS very fast
- TPC-DS fast
- TPC-DS normal
- TPC-DS slow
- TPC-DS very slow

Performance improvement: +24%
Performance improvement

TPC-DS very fast
TPC-DS fast
TPC-DS normal
TPC-DS slow
TPC-DS very slow

Original  Adaptive

+41%
Performance improvement

+285%
Performance improvement

TPC-DS very fast
TPC-DS fast
TPC-DS normal
TPC-DS slow
TPC-DS very slow

Original
Adaptive

Performance improvement +115%
Maximum acceleration
Learning progress
Learning progress

![Graph showing learning progress with a plot of execution time vs. iteration number.](image)
Learning progress

TPC-DS 26
Example: complicated query

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;
```

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

Example: complicated query
Bad cardinality estimates
Using previous statistics to refine estimates
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)
              -> 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)
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
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

- SQL query
- Query optimization
- A priori cost estimation
- Query execution
- Feedback
- Result

- Histograms
- Cost models
- Execution statistics
Questions

Contacts:
- o.ivanov@postgrespro.ru
- +7 (916) 377-55-63
Postgres Professional

http://postgrespro.ru/
+7 495 150 06 91
info@postgrespro.ru