Adaptive Query Opt.  
(PostgreSQL)

Team members: Aolei Zhou, Jiayin Zheng, Xinyi Jiang
Motivation

```
SELECT * FROM users
WHERE age > 25 AND married = true
AND position = 'CTO';
```
High-level Goal of this Project

Our goal is to research the possibility to switch query plan at the execution phase!
Adaptively Plan Node Switching (75% goal) – Done

Plan Node Can be adaptively replaced

- Template-based
- Implement as a wrapper
- Stop the query execution if certain conditions are satisfied
- Index scan -> Seq scan (finished before mid-term)
Plan Node Switching -> Plan Tree Switching

Goal: switch join method + switch join order

Why don’t we use plan node switching?
- Needs complex transformation between data structures. (6 transformers)
- Missing information.
- Single-level join order switching is not enough.
- It’s difficult to implement multi-level join ordering switch.
- Wants a unified and generalized method.

Nested Loop Join

Merge Join ——— Hash Join
Switch Join Method? – Done (90% goal)
Adaptively Plan Tree Switching

1. **Store** the sub-optimal plan in advance (different join methods)
2. If aqo is enabled & need switching,
   - Initialize the sub-optimal plan
   - **Re-execute using the sub-optimal plan**

We cannot guarantee the performance of the suboptimal plan since the production of the suboptimal plan can still **based on wrong estimations.**
How to solve the problem?

Let’s welcome Machine Learning!

-> **better estimation**

If the new plan is obviously better than the old one, we stop the execution and switch to the new plan.
Can we do better? (105% goal)

The current KNN is fast but we may need more complex methods later, which will possibly take more time.

Multi-processing!
Key Points:

How to start a new process:

+ `a$qo_bgworker_background_process_startup()`
  + `RegisterDynamicBackgroundWorker(&worker, &handle)`
  + `startup_background_process_main(Datum main_arg)`
+ Tried Using shared memory :(

How to achieve “communication”:

+ **Store** the old plan
+ The subprocess reads the old plan and **compares** it with the new plan
+ If better (estimated cost < old cost), send a **signal** + write down the new plan
+ If main process receives the signal -> stop execution + change plan + initialize and execute the plan
+ Main process -> Do no use for estimation but collect feed stats to the model

Background process -> Use ML for estimation
Baseline query plan:

Planning Time: 1.837 ms
Execution Time: 1508.714 ms

Multiprocess version:

Planning Time: 6.393 ms
Execution Time: 1439.578 ms
Evil bug 😈

323-04-28 03:00:28.163 UTC [38451] LOG: background worker "aqo background" (PID 38566) was terminated by signal 11: Segmentation fault
Evil bug 😈
There is a reason people treat warnings as failures!

```c
aqo.c: In function 'startup_background_process_main':
aqo.c:410:5: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
  410 |   const char * query_string = MyBgworkerEntry->bgw_extra;
      | ^~~~~
aqo.c:412:19: warning: implicit declaration of function 'pg_plan_queries' [-Wimplicit-function-declaration]
  412 |   List * plan = pg_plan_queries(DatumGetPointer(main_arg), query_string,
      | ^~~~~~~~~~~~~~~~~
aqo.c:412:19: warning: initialization of 'List *' from 'int' makes pointer from integer without a cast [-Wint-
```

```c
aqo.c:412:5: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
  412 |   List * plan = pg_plan_queries(DatumGetPointer(main_arg), query_string,
      | ^~~~
```
**Current Test Coverage**

1. Test case for correctness
   1. AQO make check to make sure that model is running correctly
   1. Run benchmark for both correctness and performance
Code quality

• Good:
  ○ Use a guc variable for control (flexible + generalized)
  ○ Abstract the common part (concise + readable)
  ○ Write comments (easy to understand + maintain)
  ○ Validation check (security + robustness)

• Bad:
  ○ Insufficient Script Check
  ○ Hard coding
Introduction to our benchmark: JOB

- **Join Ordering Benchmark:**
  - "How Good Are Query Optimizers, Really?" by Viktor Leis at., PVLDB Volume 9, No. 3, 2015

- **IMDB Dataset:**
  - Based on real-world dataset "Internet Movie Database"
  - Full of correlations and non-uniform data distributions
  - Contains 21 tables and is very large

- **JOB Queries:**
  - Based on IMDB Dataset
  - Focus on join ordering
  - Challenging for cardinality estimators

From paper “How Good Are Query Optimizers, Really?”
Benchmark Results on JOB (125% goal)

<table>
<thead>
<tr>
<th>base_1</th>
<th>base_2</th>
<th>base_3</th>
<th>base_avg</th>
<th>multi_1</th>
<th>multi_2</th>
<th>multi_3</th>
<th>multi_avg</th>
<th>ms</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>790.4</td>
<td>698.7</td>
<td>713.7</td>
<td>734.3</td>
<td>525.8</td>
<td>525.9</td>
<td>542.6</td>
<td>531.4</td>
<td>faster</td>
</tr>
<tr>
<td>1b</td>
<td>545.2</td>
<td>684.0</td>
<td>599.9</td>
<td>609.7</td>
<td>399.1</td>
<td>396.4</td>
<td>408.7</td>
<td>401.4</td>
<td>faster</td>
</tr>
<tr>
<td>1c</td>
<td>659.2</td>
<td>866.2</td>
<td>572.1</td>
<td>699.2</td>
<td>431.5</td>
<td>433.0</td>
<td>439.7</td>
<td>434.7</td>
<td>slower</td>
</tr>
<tr>
<td>2a</td>
<td>1213.7</td>
<td>1172.3</td>
<td>1191.6</td>
<td>1192.5</td>
<td>670.2</td>
<td>668.8</td>
<td>696.1</td>
<td>678.3</td>
<td>slower</td>
</tr>
<tr>
<td>2b</td>
<td>1156.5</td>
<td>1242.3</td>
<td>1232.8</td>
<td>1218.5</td>
<td>637.3</td>
<td>636.1</td>
<td>634.5</td>
<td>636.0</td>
<td>faster</td>
</tr>
<tr>
<td>2c</td>
<td>1374.5</td>
<td>1216.3</td>
<td>1552.4</td>
<td>1381.1</td>
<td>566.9</td>
<td>556.9</td>
<td>552.6</td>
<td>558.8</td>
<td>faster</td>
</tr>
<tr>
<td>3a</td>
<td>3459.112</td>
<td>3514.232</td>
<td>3443.99</td>
<td>3472.4</td>
<td>3255.412</td>
<td>3036.633</td>
<td>3305.476</td>
<td>3199.2</td>
<td>same</td>
</tr>
<tr>
<td>3b</td>
<td>1753.944</td>
<td>1882.727</td>
<td>1641.399</td>
<td>1759.4</td>
<td>1655.424</td>
<td>1599.157</td>
<td>1699.388</td>
<td>1648.3</td>
<td>same</td>
</tr>
<tr>
<td>3c</td>
<td>2175.199</td>
<td>2281.588</td>
<td>2293.991</td>
<td>2250.3</td>
<td>1943.013</td>
<td>1966.708</td>
<td>1931.699</td>
<td>1947.1</td>
<td>same</td>
</tr>
<tr>
<td>4a</td>
<td>833.81</td>
<td>756.113</td>
<td>826.271</td>
<td>805.398</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>654.4</td>
<td>644.6</td>
<td>1067.9</td>
<td>788.9</td>
<td>477.3</td>
<td>463.6</td>
<td>482.0</td>
<td>474.3</td>
<td>faster</td>
</tr>
<tr>
<td>4c</td>
<td>993.873</td>
<td>857.636</td>
<td>949.586</td>
<td>933.693333</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5a</td>
<td>586.276</td>
<td>521.884</td>
<td>456.812</td>
<td>521.7</td>
<td>279.7</td>
<td>264.5</td>
<td>310.9</td>
<td>285.0</td>
<td>faster</td>
</tr>
<tr>
<td>5b</td>
<td>389.8</td>
<td>339.8</td>
<td>486.0</td>
<td>378.3</td>
<td>260.1</td>
<td>288.5</td>
<td>294.3</td>
<td>281.0</td>
<td>faster</td>
</tr>
<tr>
<td>5c</td>
<td>2119.563</td>
<td>2353.573</td>
<td>2099.76</td>
<td>2191.0</td>
<td>1850.974</td>
<td>1858.63</td>
<td>2043.974</td>
<td>1915.2</td>
<td>same</td>
</tr>
<tr>
<td>6a</td>
<td>3386.672</td>
<td>3319.672</td>
<td>3477.385</td>
<td>3394.6</td>
<td>6740.0</td>
<td>8205.8</td>
<td>8286.3</td>
<td>7744.0</td>
<td>slower</td>
</tr>
<tr>
<td>6b</td>
<td>3371.631</td>
<td>3445.023</td>
<td>3484.529</td>
<td>3433.727667</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6c</td>
<td>3218.902</td>
<td>3352.921</td>
<td>3352.921</td>
<td>3308.2</td>
<td>11150.3</td>
<td>10317.5</td>
<td>11150.3</td>
<td>10872.7</td>
<td>slower</td>
</tr>
<tr>
<td>7a</td>
<td>4525.2</td>
<td>4700.9</td>
<td>4318.2</td>
<td>4514.8</td>
<td>2966.9</td>
<td>3150.3</td>
<td>2945.6</td>
<td>3026.9</td>
<td>faster</td>
</tr>
<tr>
<td>7b</td>
<td>3223.415</td>
<td>3376.853</td>
<td>3292.282</td>
<td>3297.5</td>
<td>3045.582</td>
<td>2998.835</td>
<td>3015.799</td>
<td>3017.4</td>
<td>same</td>
</tr>
<tr>
<td>7c</td>
<td>6202.958</td>
<td>5948.215</td>
<td>6111.585</td>
<td>6027.6</td>
<td>4165.6</td>
<td>3981.3</td>
<td>4171.9</td>
<td>4106.3</td>
<td>faster</td>
</tr>
<tr>
<td>8a</td>
<td>2963.995</td>
<td>3019.798</td>
<td>2821.164</td>
<td>2935.0</td>
<td>2963.8</td>
<td>3009.7</td>
<td>3032.3</td>
<td>3001.9</td>
<td>same</td>
</tr>
<tr>
<td>8b</td>
<td>2801.645</td>
<td>2938.337</td>
<td>3039.854</td>
<td>2926.6</td>
<td>27594.9</td>
<td>27242.0</td>
<td>27289.2</td>
<td>27375.4</td>
<td>slower</td>
</tr>
<tr>
<td>8c</td>
<td>7900.462</td>
<td>8017.525</td>
<td>7899.916</td>
<td>7939.3</td>
<td>7920.713</td>
<td>7738.812</td>
<td>8236.452</td>
<td>7965.3</td>
<td>same</td>
</tr>
<tr>
<td>9a</td>
<td>4371.496</td>
<td>4498.63</td>
<td>4411.735</td>
<td>4427.3</td>
<td>5708.5</td>
<td>5607.3</td>
<td>5604.1</td>
<td>5640.0</td>
<td>slower</td>
</tr>
<tr>
<td>9b</td>
<td>3252.645</td>
<td>3236.819</td>
<td>3081.874</td>
<td>3190.4</td>
<td>5954.9</td>
<td>6377.2</td>
<td>6340.1</td>
<td>5978.7</td>
<td>slower</td>
</tr>
<tr>
<td>9c</td>
<td>4336.901</td>
<td>4763.168</td>
<td>4548.448</td>
<td>4549.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution time calculated until ML converge (few trail trains not counted here)
Benchmark Results on JOB (125% goal)

50% Faster!!

<table>
<thead>
<tr>
<th></th>
<th>base_1</th>
<th>base_2</th>
<th>base_3</th>
<th>base_avg</th>
<th>multi_1</th>
<th>multi_2</th>
<th>multi_3</th>
<th>multi_avg</th>
<th>if</th>
<th>ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>790.4</td>
<td>698.7</td>
<td>713.7</td>
<td><strong>734.3</strong></td>
<td>525.8</td>
<td>525.9</td>
<td>542.6</td>
<td><strong>531.4</strong></td>
<td>faster</td>
<td>same</td>
</tr>
<tr>
<td>b</td>
<td>545.2</td>
<td>684.0</td>
<td>599.9</td>
<td><strong>609.7</strong></td>
<td>399.1</td>
<td>396.4</td>
<td>408.7</td>
<td><strong>401.4</strong></td>
<td>faster</td>
<td>&lt;0.8*base</td>
</tr>
<tr>
<td>c</td>
<td>659.2</td>
<td>866.2</td>
<td>572.1</td>
<td><strong>699.2</strong></td>
<td>431.5</td>
<td>433.0</td>
<td>439.7</td>
<td><strong>434.7</strong></td>
<td>faster</td>
<td>&gt;1.2*base</td>
</tr>
<tr>
<td>2a</td>
<td>1213.7</td>
<td>1172.3</td>
<td>1191.6</td>
<td><strong>1192.5</strong></td>
<td>670.2</td>
<td>668.8</td>
<td>696.1</td>
<td><strong>678.3</strong></td>
<td>faster</td>
<td>longer than 2min</td>
</tr>
<tr>
<td>2b</td>
<td>1156.5</td>
<td>1242.3</td>
<td>1232.8</td>
<td><strong>1210.5</strong></td>
<td>637.3</td>
<td>636.1</td>
<td>634.5</td>
<td><strong>636.0</strong></td>
<td>faster</td>
<td></td>
</tr>
<tr>
<td>2c</td>
<td>1374.5</td>
<td>1216.3</td>
<td>1552.4</td>
<td><strong>1381.1</strong></td>
<td>566.9</td>
<td>556.9</td>
<td>552.6</td>
<td><strong>558.8</strong></td>
<td>faster</td>
<td></td>
</tr>
<tr>
<td>3a</td>
<td>3459.112</td>
<td>3514.232</td>
<td>3443.99</td>
<td><strong>3472.4</strong></td>
<td>3255.412</td>
<td>3036.633</td>
<td>3305.476</td>
<td><strong>3199.2</strong></td>
<td>same</td>
<td></td>
</tr>
<tr>
<td>3b</td>
<td>1753.994</td>
<td>1882.727</td>
<td>1641.399</td>
<td><strong>1759.4</strong></td>
<td>1655.424</td>
<td>1599.157</td>
<td>1698.388</td>
<td><strong>1648.3</strong></td>
<td>same</td>
<td></td>
</tr>
</tbody>
</table>

- **50% Faster!!**

**Base Results:**
- **2a** 1213.7 1172.3 1192.5 670.2 668.8 696.1 **678.3** faster
- **2b** 1156.5 1242.3 1210.5 637.3 636.1 634.5 **636.0** faster
- **2c** 1374.5 1216.3 1552.4 **1381.1** 566.9 556.9 552.6 **558.8** faster

**Additional Data:**
- **7b** 3223.419 3376.853 3292.282
- **7c** 6022.958 5948.215 6111.585
- **8a** 2963.995 3019.798 2821.104
- **8b** 2801.645 2938.337 3039.854
- **8c** 7900.462 8017.525 7899.916
- **9a** 4371.496 4498.63 4411.735
- **9b** 3252.645 3236.819 3081.874
- **9c** 4336.901 4763.168 4548.448
- **10a** 3430.823 3401.248 3571.186
- **10b** 3350.946 3377.193 3417.622
- **10c** 4475.686 4425.701 4605.28

**Graph:**
- Too Slow: >2m
- 6 - 20.0%
Benchmark Results on JOB (125% goal)

- **Pro:**
  - Great performance (36.7%)
  - Improvement in Simple Query
  - ML have chance to learn better query plans through trial

- **Con:**
  - ML performance worse than baseline in first few runs
  - ML performance is **unstable**
  - Hard to converge on **complex** queries

Execution time calculated until ML converge (few trail trains not counted here)
Future Work

- Better testing: Unit test
- Add execution time to the current cost
- Using more complex ML algorithms
- Considering other techniques including sampling
Resources

1. Join Order Benchmark (JOB)
3. Computation resources
4. Code review pipeline
5. Kudos to various PostgreSQL extension resources from Wan and Abby