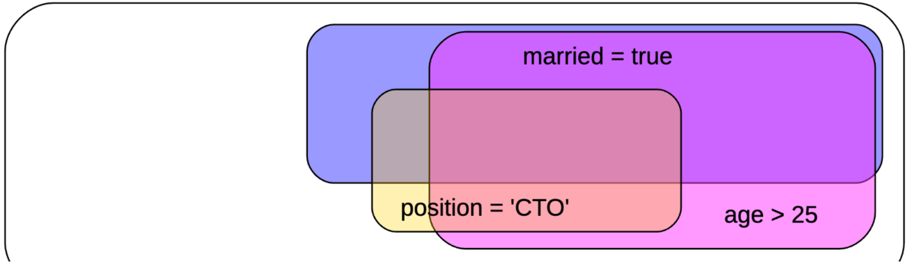
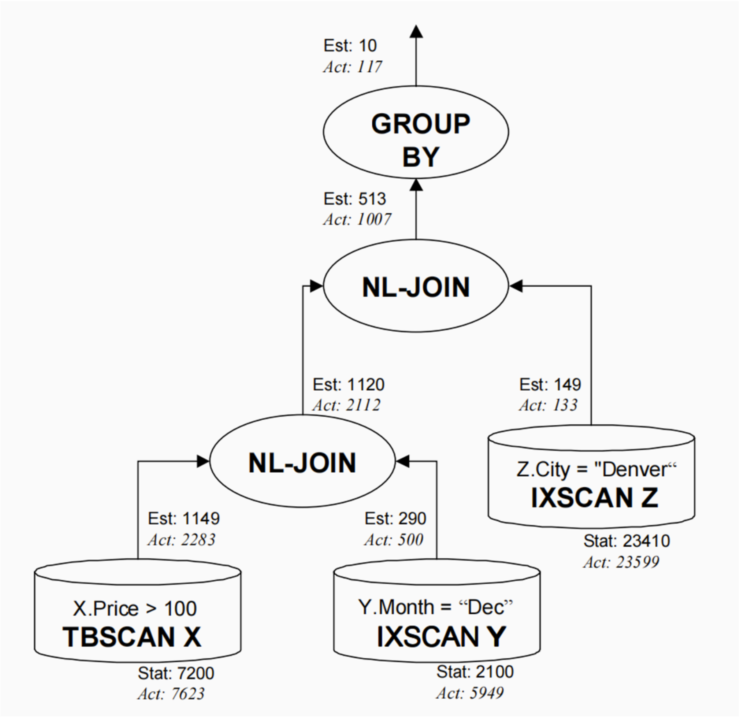
Adaptive Query Opt. (PostgreSQL)

Team members: Aolei Zhou, Jiayin Zheng, Xinyi Jiang

Motivation

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

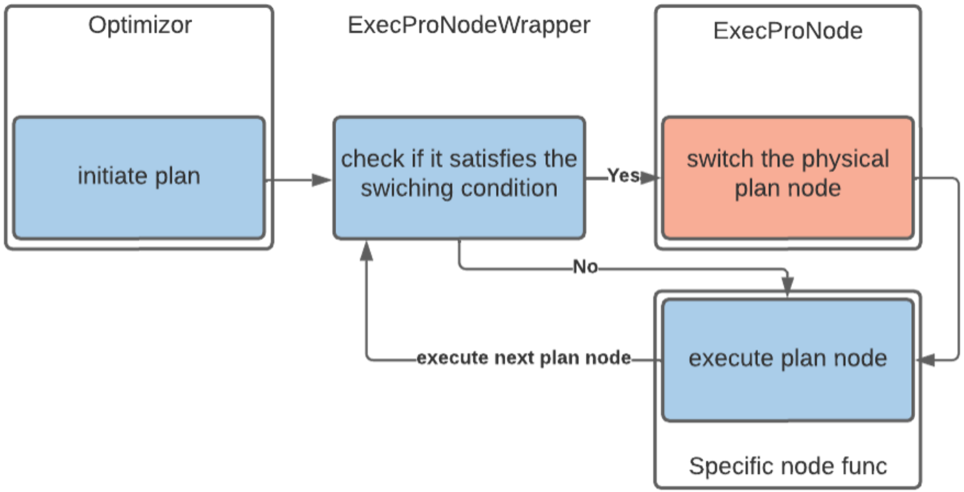




High-level Goal of this Project

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



Approach 1: Adaptively Plan Node Switching

```
noisepage_db=# CREATE INDEX identifier_index on chicken (identifier);
CREATE INDEX
noisepage_db=# EXPLAIN Select * from chicken where identifier < 30;
                                              QUERY PLAN
------------------------------------------------------------------------------------------------
 Index Scan using identifier_index on chicken  (cost=0.42..2001.72 rows=40000 width=140)
   Index Cond: (identifier < 30)
(2 rows)
```

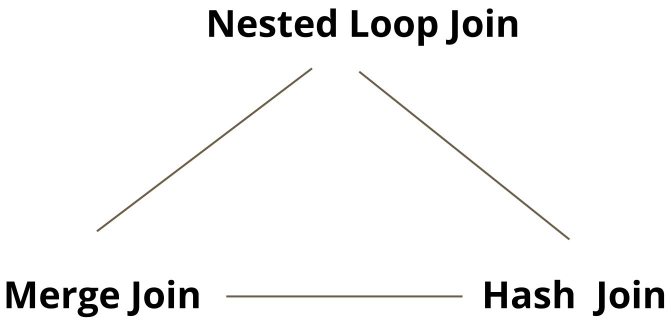
- Template-based
- Implement as a wrapper
- Stop the query execution if certain conditions are satisfied
- Index scan -> Seq scan (finished before mid-term)

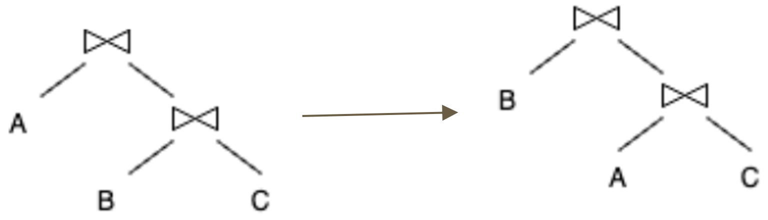
Plan Node Switching -> Plan Tree Switching

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.





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**

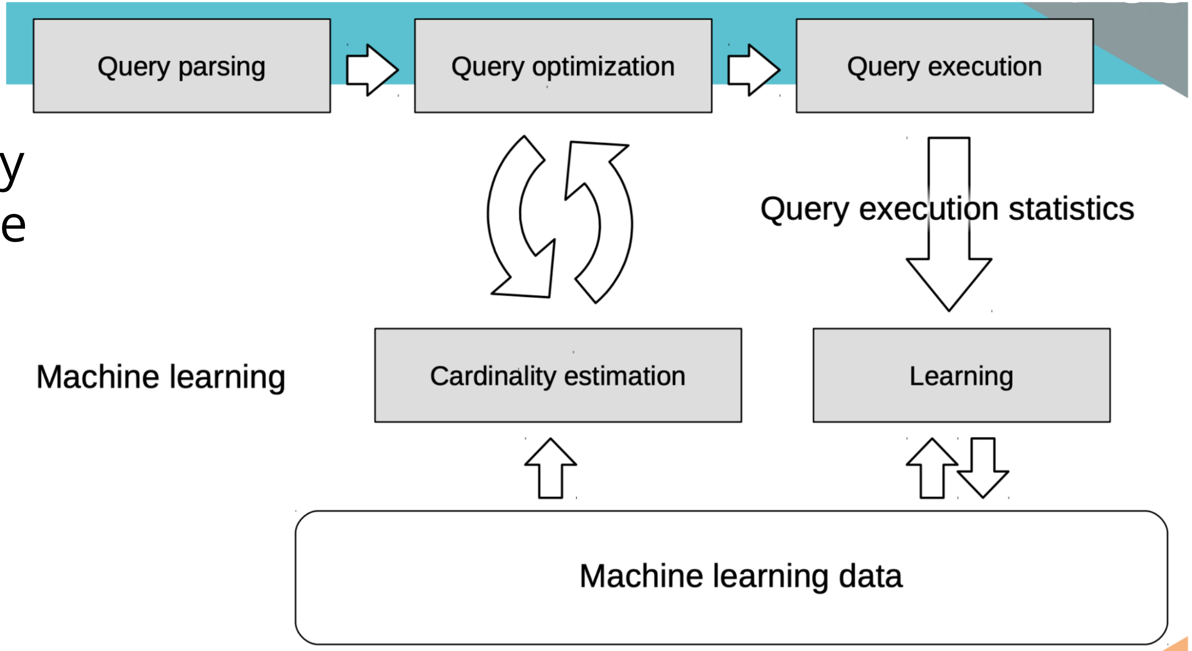
```
2023-05-04 16:54:03.380 UTC [125248] LOG:  do recycle new
2023-05-04 16:54:03.380 UTC [125248] STATEMENT:  select f1.farm_name,f2.farm_name,f3.farm_name from farm f1 join farm f2 on f1.min_age_weeks =
2023-05-04 16:54:03.380 UTC [125248] LOG:  aqo best path:{HASHPATH :pathtype 40 :parent_relids (b 1 2) :required_outer (b) :parallel_aware fals
00 :pathkeys <> :jointype 0 :inner_unique false :outerjoinpath {PATH :pathtype 20 :parent_relids (b 1) :required_outer (b) :parallel_aware fals
 :pathkeys <>} :innerjoinpath {PATH :pathtype 20 :parent_relids (b 2) :required_outer (b) :parallel_aware false :parallel_safe true :parallel_w
({RESTRICTINFO :clause {OPEXPR :opno 620 :opfuncid 287 :opresulttype 16 :opretset false :opcollid 0 :inputcollid 0 :args ({VAR :varno 1 :varatt
ocation 75} {VAR :varno 2 :varattno 2 :vartype 700 :vartypmod -1 :varcollid 0 :varlevelsup 0 :varnosyn 2 :varattnosyn 2 :location 94}) :locatio
:leakproof false :has_volatile 2 :security_level 0 :clause_relids (b 1 2) :required_relids (b 1 2) :outer_relids (b) :nullable_relids (b) :left
ergeopfamilies (o 1970) :left_em {EQUIVALENCEMEMBER :em_expr {VAR :varno 1 :varattno 2 :vartype 700 :vartypmod -1 :varcollid 0 :varlevelsup 0 :
onst false :em_is_child false :em_datatype 700} :right_em {EQUIVALENCEMEMBER :em_expr {VAR :varno 2 :varattno 2 :vartype 700 :vartypmod -1 :var
lable_relids (b) :em_is_const false :em_is_child false :em_datatype 700} :outer_is_left true :hashjoinoperator 620 :left_hasheqoperator 620 :ri
ncid 287 :opresulttype 16 :opretset false :opcollid 0 :inputcollid 0 :args ({VAR :varno 1 :varattno 2 :vartype 700 :vartypmod -1 :varcollid 0 :
```

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**

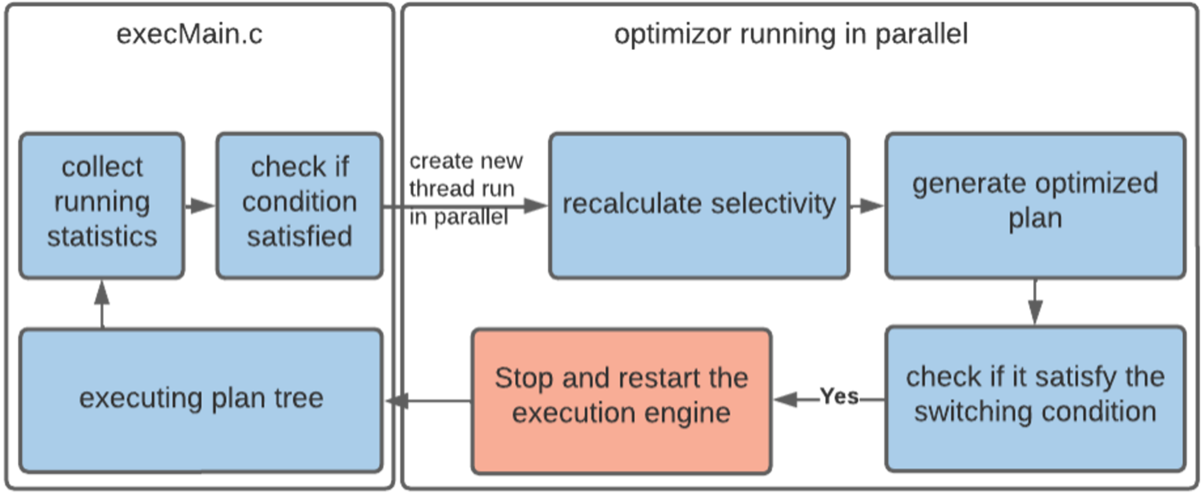
If the new plan is obviously better than the old one, we **stop the execution and switch to the new plan.**



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 !



Approach 2: Adaptively Plan Tree Switching

Key Points:

How to start a new process:

+ aqo_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:

```
----------------------------------------------------------------
Aggregate  (cost=85645.56..85645.57 rows=1 width=32) (actual ti
  ->  Gather  (cost=51879.08..85645.55 rows=2 width=17) (actual
        Workers Planned: 2
        Workers Launched: 2
        ->  Nested Loop  (cost=50879.08..84645.35 rows=1 width=
              ->  Nested Loop  (cost=50878.65..84644.88 rows=1
                    ->  Parallel Hash Join  (cost=50878.23..846
                          Hash Cond: (mc.movie_id = mk.movie_ic
                          ->  Parallel Seq Scan on movie_compar
                          ->  Parallel Hash  (cost=50878.05..50
                                Buckets: 65536 (originally 1024
                                ->  Hash Join  (cost=2626.14..5
                                      Hash Cond: (mk.keyword_ic
                                      ->  Parallel Seq Scan on
                                      ->  Hash  (cost=2626.12..
                                            Buckets: 1024  Batc
                                            ->  Seq Scan on key
                                                  Filter: (keyw
                                                  Rows Removed
                    ->  Index Scan using company_name_pkey on c
                          Index Cond: (id = mc.company_id)
                          Filter: ((country_code)::text = '[nl]
                          Rows Removed by Filter: 1
              ->  Index Scan using title_pkey on title t  (cost
                    Index Cond: (id = mc.movie_id)
Planning Time: 1.837 ms
Execution Time: 1508.714 ms
```

Multiprocess version:

```
------------------------------------------------------------
Finalize Aggregate  (cost=97427.81..97427.82 rows=1 width=32
  ->  Gather  (cost=97427.59..97427.80 rows=2 width=32) (act
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=96427.59..96427.60 rows
              ->  Parallel Hash Join  (cost=63819.48..96424.
                    Hash Cond: (mc.movie_id = t.id)
                    ->  Parallel Hash Join  (cost=4740.12..3
                          Hash Cond: (mc.company_id = cn.id)
                          ->  Parallel Seq Scan on movie_com
                          ->  Parallel Hash  (cost=4722.92..
                                Buckets: 4096  Batches: 1  M
                                ->  Parallel Seq Scan on com
                                      Filter: ((country_code
                                      Rows Removed by Filter
                    ->  Parallel Hash  (cost=58861.45..58861
                          Buckets: 65536  Batches: 1  Memory
                          ->  Nested Loop  (cost=2626.57..58
                                ->  Hash Join  (cost=2626.14
                                      Hash Cond: (mk.keyword
                                      ->  Parallel Seq Scan
                                      ->  Hash  (cost=2626.1
                                            Buckets: 1024  B
                                            ->  Seq Scan on
                                                  Filter: (k
                                                  Rows Remov
                                ->  Index Scan using title_p
                                      Index Cond: (id = mk.m
Planning Time: 6.393 ms
Execution Time: 1439.578 ms
```

Evil bug 😈

```
023-04-28 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!

```
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-
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

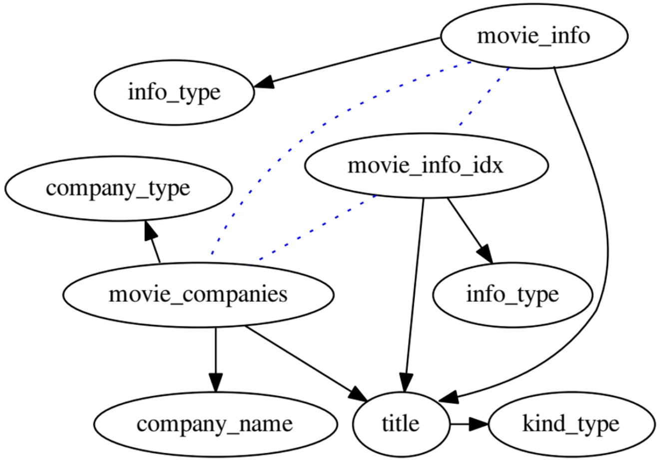


Figure 2: Typical query graph of our workload

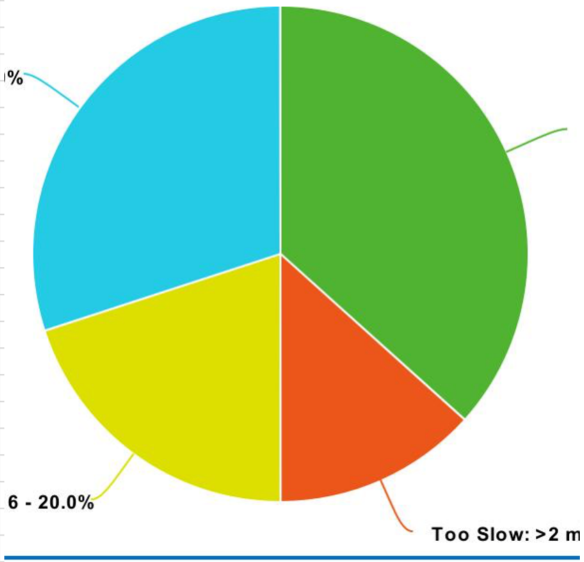
From paper "How Good Are Query Optimizers, Really?"

Benchmark Results on JOB (125% goal)

	base_1	base_2	base_3	**base_avg**	multi_1	nulti_2	multi_3	**multi_avg**	if
1a	790.4	698.7	713.7	**734.3**	525.8	525.9	542.6	**531.4**	faster
1b	545.2	684.0	599.9	**609.7**	399.1	396.4	408.7	**401.4**	faster
1c	659.2	866.2	572.1	**699.2**	431.5	433.0	439.7	**434.7**	faster
2a	1213.7	1172.3	1191.6	**1192.5**	670.2	668.8	696.1	**678.3**	faster
2b	1156.5	1242.3	1232.8	**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
3a	3459.112	3514.232	3443.99	**3472.4**	3255.412	3036.633	3305.476	**3199.2**	same
3b	1753.994	1882.727	1641.399	**1759.4**	1655.424	1599.157	1690.388	**1648.3**	same
3c	2175.199	2281.588	2293.991	**2250.3**	1943.013	1966.708	1931.699	**1947.1**	same
4a	833.81	756.113	826.271	**805.398**					
4b	654.4	644.6	1067.9	**788.9**	477.3	463.6	482.0	**474.3**	faster
4c	993.873	857.636	949.586	**933.6983333**					
5a	586.276	521.884	456.812	**521.7**	279.7	264.5	310.9	**285.0**	faster
5b	389.0	339.8	406.0	**378.3**	260.1	288.5	294.3	**281.0**	faster
5c	2119.563	2353.573	2099.76	**2191.0**	1850.974	1850.63	2043.974	**1915.2**	same
6a	3386.672	3319.672	3477.385	**3394.6**	6740.0	8205.8	8286.3	**7744.0**	slower
6b	3371.631	3445.023	3484.529	**3433.727667**					
6c	3218.902	3352.92	3352.921	**3308.2**	11150.3	10317.5	11150.3	**10872.7**	slower
7a	4525.2	4700.9	4318.2	**4514.8**	2966.9	3150.3	2945.6	**3020.9**	faster
7b	3223.415	3376.853	3292.282	**3297.5**	3045.502	2990.835	3015.799	**3017.4**	same
7c	6022.958	5948.215	6111.585	**6027.6**	4165.6	3981.3	4171.9	**4106.3**	faster
8a	2963.995	3019.798	2821.104	**2935.0**	2963.8	3009.7	3032.3	**3001.9**	same
8b	2801.645	2938.337	3039.854	**2926.6**	27594.9	27242.0	27289.2	**27375.4**	slower
8c	7900.462	8017.525	7899.916	**7939.3**	7920.713	7738.812	8236.452	**7965.3**	same
9a	4371.496	4498.63	4411.735	**4427.3**	5708.5	5607.3	5604.1	**5640.0**	slower
9b	3252.645	3236.819	3081.874	**3190.4**	5954.9	6377.2	5604.1	**5978.7**	slower
9c	4336.901	4763.168	4548.448	**4549.5**					
10a	3430.023	3401.248	3571.106	**3467.5**	3376.594	3320.884	3304.734	**3334.1**	same
10b	3350.946	3377.193	3417.622	**3381.9**	3326.711	3271.639	3076.179	**3224.8**	same
10c	4475.686	4425.701	4605.28	**4502.2**	4764.283	4641.384	4604.979	**4670.2**	same

ms	
same	0.8*base ~ 1.2*base
faster	<0.8*base
slower	>1.2*base
	longer than 2min

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



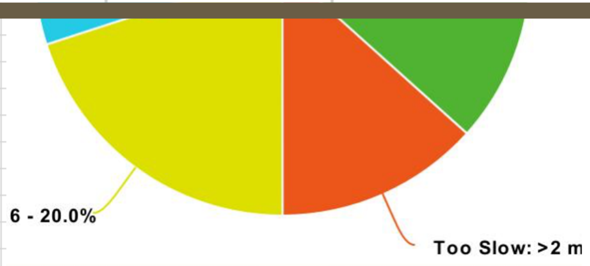
Benchmark Results on JOB (125% goal)

	base_1	base_2	base_3	base_avg	multi_1	nulti_2	multi_3	multi_avg	if
1a	790.4	698.7	713.7	734.3	525.8	525.9	542.6	531.4	faster
1b	545.2	684.0	599.9	609.7	399.1	396.4	408.7	401.4	faster
1c	659.2	866.2	572.1	699.2	431.5	433.0	439.7	434.7	faster
2a	1213.7	1172.3	1191.6	1192.5	670.2	668.8	696.1	678.3	faster
2b	1156.5	1242.3	1232.8	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
3a	3459.112	3514.232	3443.99	3472.4	3255.412	3036.633	3305.476	3199.2	same
3b	1753.994	1882.727	1641.399	1759.4	1655.424	1599.157	1690.388	1648.3	same
7b	3223.415	3376.853	3292.282	3297.5	3045.502	2990.835	3015.799	3017.4	same
7c	6022.958	5948.215	6111.585	6027.6	4165.6	3981.3	4171.9	4106.3	faster
8a	2963.995	3019.798	2821.104	2935.0	2963.8	3009.7	3032.3	3001.9	same
8b	2801.645	2938.337	3039.854	2926.6	27594.9	27242.0	27289.2	27375.4	slower
8c	7900.462	8017.525	7899.916	7939.3	7920.713	7738.812	8236.452	7965.3	same
9a	4371.496	4498.63	4411.735	4427.3	5708.5	5607.3	5604.1	5640.0	slower
9b	3252.645	3236.819	3081.874	3190.4	5954.9	6377.2	5604.1	5978.7	slower
9c	4336.901	4763.168	4548.448	4549.5					
10a	3430.023	3401.248	3571.106	3467.5	3376.594	3320.884	3304.734	3334.1	same
10b	3350.946	3377.193	3417.622	3381.9	3326.711	3271.639	3076.179	3224.8	same
10c	4475.686	4425.701	4605.28	4502.2	4764.283	4641.384	4604.979	4670.2	same

	base_1	base_2	base_3	base_avg	multi_1	nulti_2	multi_3	multi_avg	if
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	#####	1381.1	566.9	556.9	552.6	558.8	faster

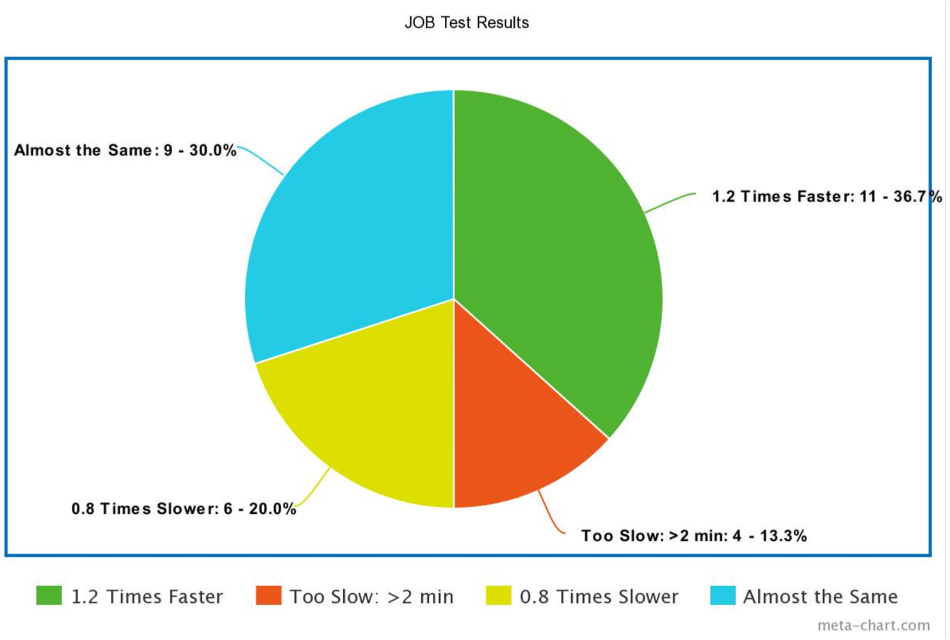
ms	
same	0.8*base ~ 1.2*base
faster	<0.8*base
slower	>1.2*base
	longer than 2min

50% Faster!!



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)
2. Adaptive query optimization: https://github.com/postgrespro/aqo
3. Computation resources
4. Code review pipeline
5. Kudos to various PostgreSQL extension resources from Wan and Abby