Query Planning

Lev Dubinets

SQL Nested Loop Semantics

```
SELECT DISTINCT column1, column2, column3
FROM table1 AS t1, table2 AS t2, table3 AS t3
WHERE <conditions>
```

```
result = set()
for x in t1
  for y in t2
    for z in t3
      if conditions(x,y,z)
        result.add((column1, column2, column3))

return result
```

Semantics - not implementation

Suppliers and Parts Database

```
CREATE TABLE Supplier (
  SID     int          primary key,
  SName   varchar(10)  NOT NULL,
  Status  int          NOT NULL,
  City    varchar(10)  NOT NULL
)

CREATE TABLE Part (
  PID     int          primary key,
  PName   varchar(10)  NOT NULL,
  Color   int          NOT NULL,
  Weight  real         NOT NULL,
  City    varchar(10)  NOT NULL
)

CREATE TABLE Shipment (
  SID     int          NOT NULL FOREIGN KEY REFERENCES Supplier(SID),
  PID     int          NOT NULL FOREIGN KEY REFERENCES Part(PID),
  Qty     int          NOT NULL,
  PRIMARY KEY (SID, PID)
)
```

https://en.wikipedia.org/wiki/Suppliers_and_Parts_database

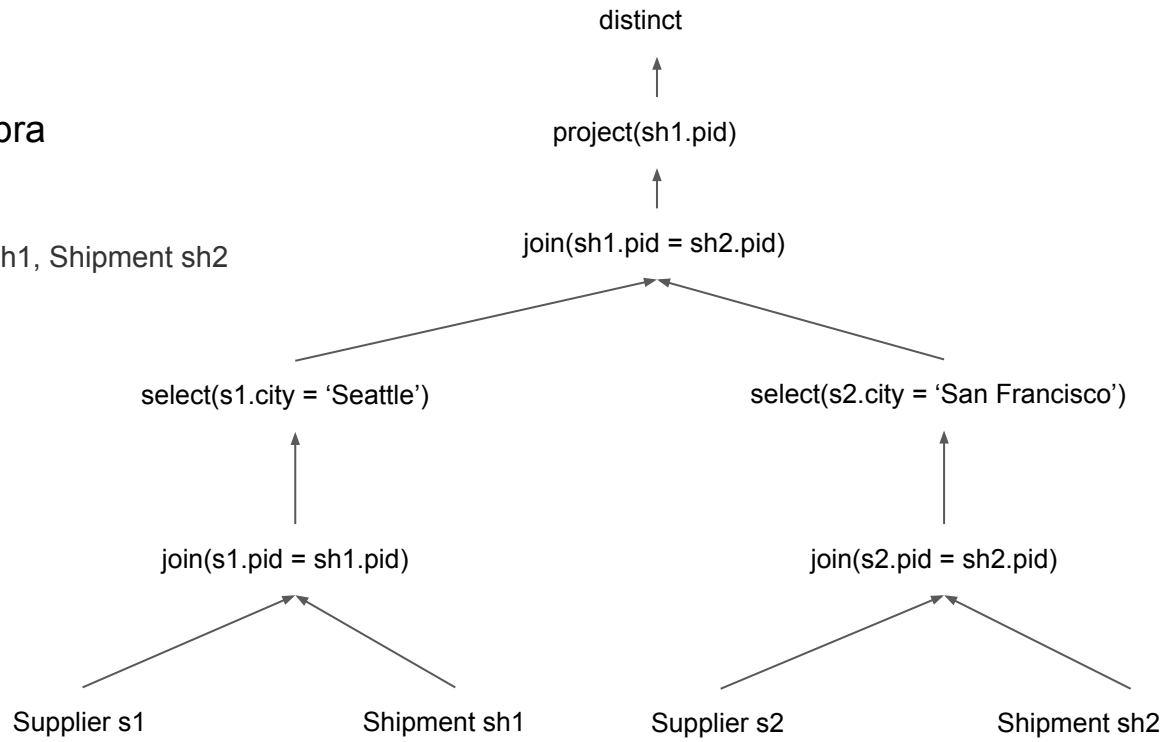
Query Planning Overview

1. Parse SQL into a relational algebra tree or logical plan
    a. Check that all relations and columns are known
2. Apply logical optimizations to plan
    a. Based on heuristics like “push selections down” or “pull projections up”
3. Convert logical plan to physical plan
    a. Allows tuning and knowledge of execution environment
    b. Where a majority of bad optimizations occur

1. Query Plan

Convert SQL into a relational algebra

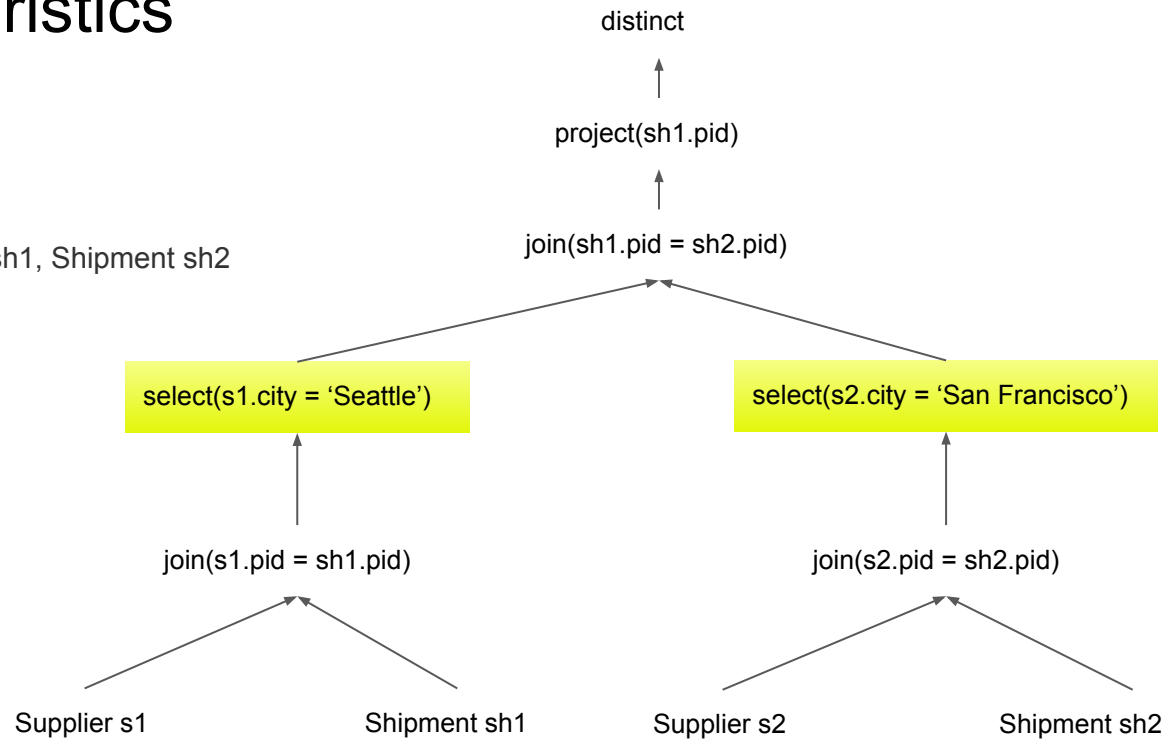
```
SELECT DISTINCT sh1.pid
FROM Supplier s1, Supplier s2, Shipment sh1, Shipment sh2
WHERE s1.city = 'Seattle'
AND s1.pid = sh1.pid
AND s2.city = 'San Francisco'
AND s2.pid = s2.pid
AND sh1.pid = sh2.pid
```



2. Optimize w/ Heuristics

Push selections down

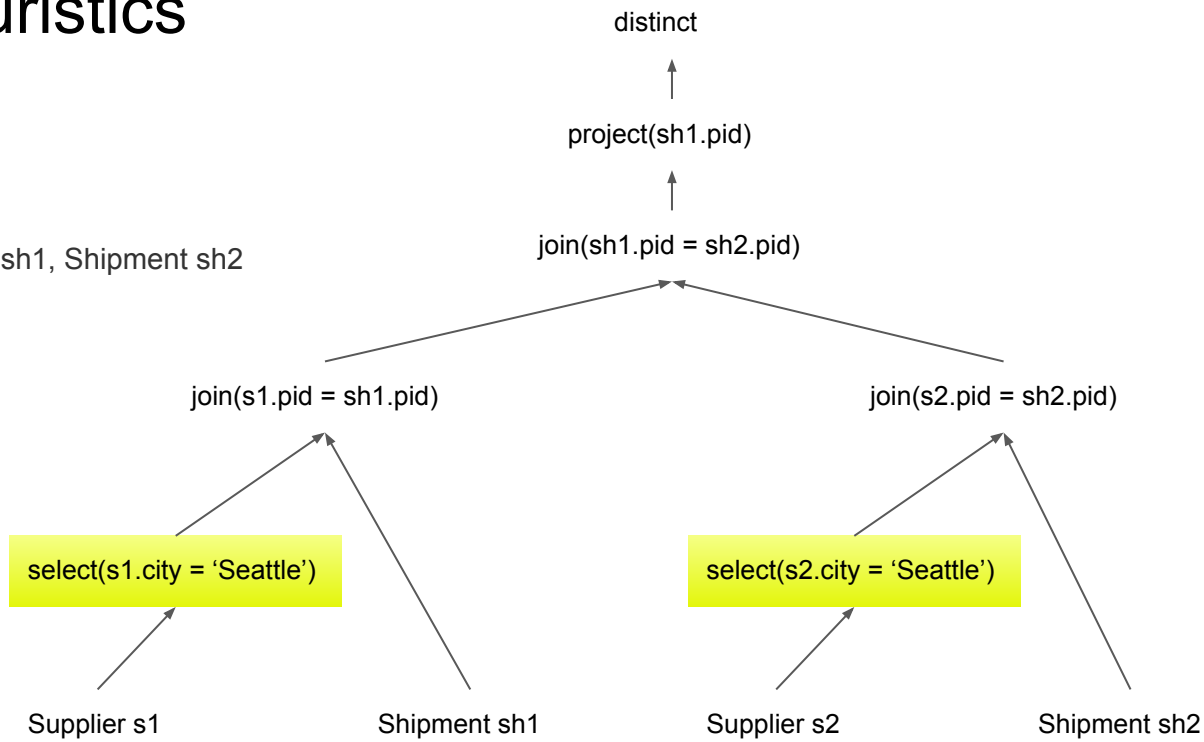
```
SELECT DISTINCT sh1.pid
FROM Supplier s1, Supplier s2, Shipment sh1, Shipment sh2
WHERE s1.city = 'Seattle'
AND s1.pid = sh1.pid
AND s2.city = 'San Francisco'
AND s2.pid = s2.pid
AND sh1.pid = sh2.pid
```



2. Optimize w/ Heuristics

Push selections down

```
SELECT DISTINCT sh1.pid
FROM Supplier s1, Supplier s2, Shipment sh1, Shipment sh2
WHERE s1.city = 'Seattle'
AND s1.pid = sh1.pid
AND s2.city = 'San Francisco'
AND s2.pid = s2.pid
AND sh1.pid = sh2.pid
```

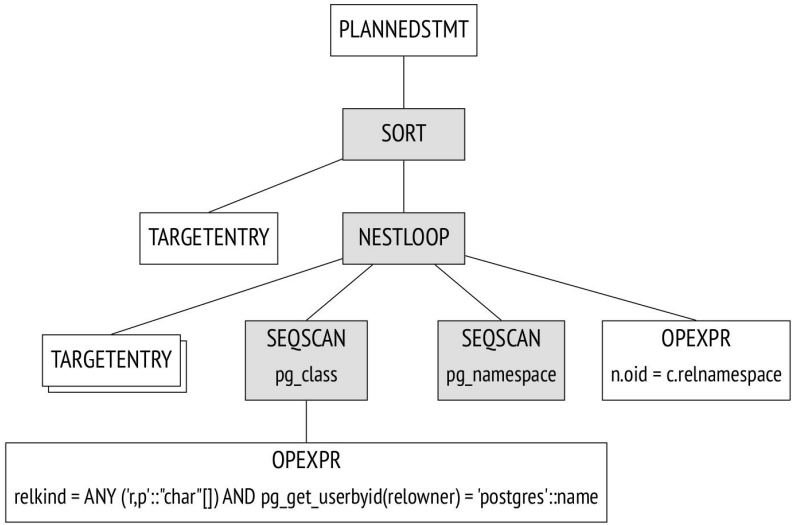


3. Physical Query Plan

Similar to logical plan but specifies exactly what types of scans and joins to use (e.g. index scan on idx1, hash join on join1, nested loop join on join2)

Major challenges

- Which type of join/scan to use?
- What order to join tables in?



“Queries in PostgreSQL”
https://habr.com/en/company/postgrespro/blog/649499/

3. Physical Query Plan

Which type of join to use?

- e.g. based on disk access speeds, and number of rows we'll need to join, we should use a nested loop join at a certain node

What order to join tables in?

- Is the query faster if I join supplier on supply first, or supply on shipment first?

Query Planner Options

Postgres allow configuration options to set the cost model for disk/memory speeds, as well as allow thresholds for when it should use a specific type of join/scan.

Postgres also provides per-query “planner hints” as configuration parameters

e.g. “SET enable_hashjoin = ‘off’; SELECT * from ….”

https://www.postgresql.org/docs/current/runtime-config-query.html

3. Physical Query Plan

The rest of this presentation is set in stage 3 - creating the physical query plan.

We assume that steps 1 and 2 (parsing SQL and logical optimizations) are relatively straightforward and there is a deterministic “correct answer” that is easy for the optimizer to compute every time.

Typical Query Optimizer

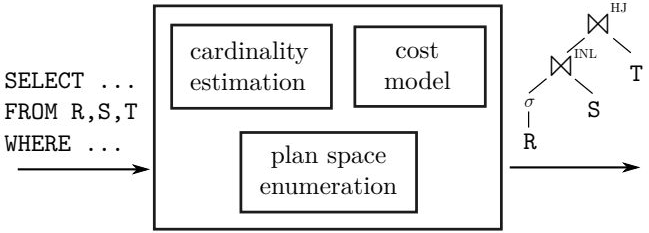


Figure 1: Traditional query optimizer architecture

"Using experiments that measure the impact of the cost model, we find that it has much less influence on query performance than the cardinality estimates."

"How Good Are Query Optimizers, Really?"
http://www.vldb.org/pvldb/vol9/p204-leis.pdf

Join Order Benchmark

- A set of 100 queries on a publicly available database from IMDB
- Benchmarks like TPCC don't contain real "production"-like data, so benchmarking join orders on those benchmarks doesn't make sense
- Used in basically every recent research paper I read on query optimization improvements

"How Good Are Query Optimizers, Really?"
http://www.vldb.org/pvldb/vol9/p204-leis.pdf

Plan Space Enumeration

Which type of join/scan to use?

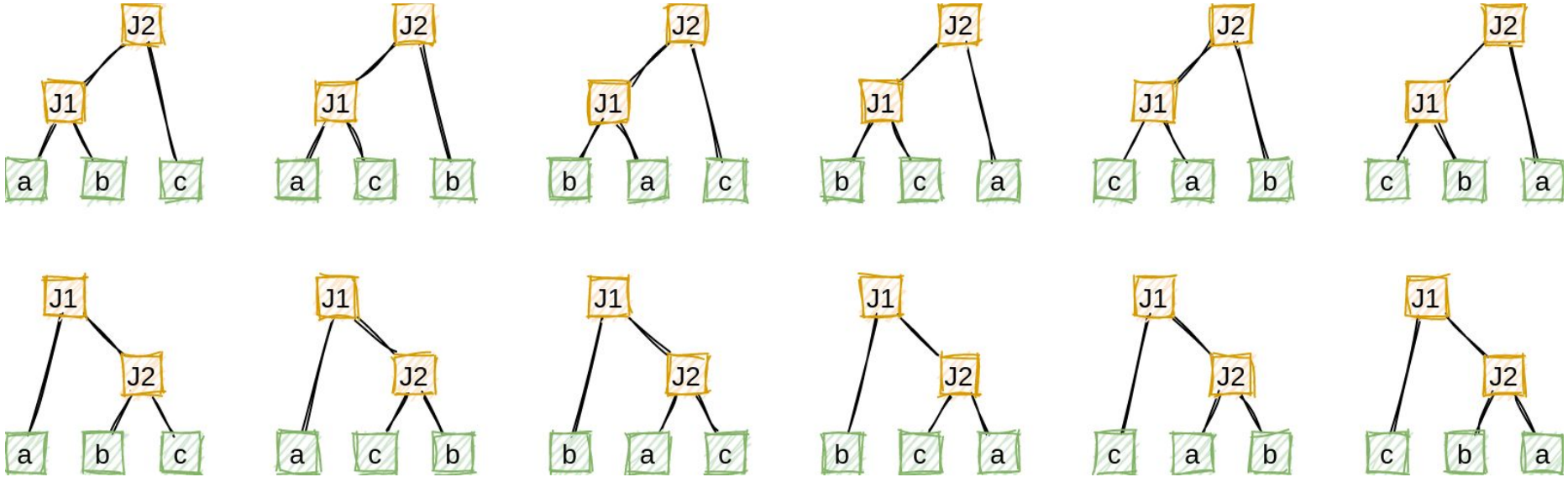
- Consider all types of joins/scan at each node to the plan space.
- Chose the one with the lowest cost according to cost model.

What order to join tables in?

- Number of possible join orders grows exponentially w/ number of joins
- Use dynamic programming and heuristics to search the space
- (Postgres12+) Use genetic programming to search the space
- Both algorithms also rely heavily on cost model

Plan Space Enumeration - Join Order

For 3 relations there are 12 valid join orders:



https://www.querifylabs.com/blog/introduction-to-the-join-ordering-problem

Plan Space Enumeration - Dynamic Program

Use memorization and dynamic programming to chose the join order for a given set of tables.
There are dozens of algorithms to pick from.
The one below is an ultra-simplification.

```
costs = {}
for s in [1, num_tables]:
    for each subset of tables S of size s:
        splits = all tuples of (table T, [set of s-1 tables S'])
        costs[S] = min_splits(costs[S'] + cost_model(T join S'))

return costs[all_tables]
```

Cost Model

Cost is an abstract concept. The cost model assigns a cost to each operator in the physical query plan, and typically the cost is multiplied by the size of the set that the operator is working on.

Cardinality estimation improvements typically dwarf any cost model tuning that can be done.

Cardinality Estimation

To estimate the size of the result set at any given node, Postgres maintains a number of statistics about the tables in a database:

- Number of records
- Number of physical pages
- Map of index => number of keys in the index
- Histogram on each column of a table
  - e.g. number of rows with city = ‘seattle’

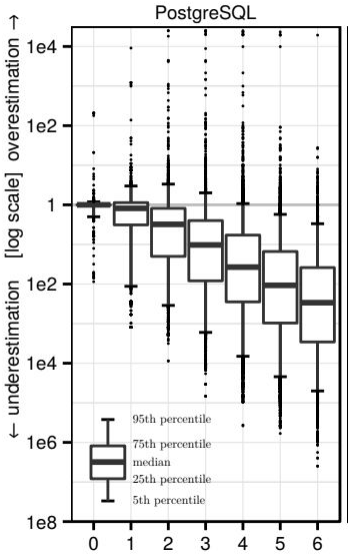
Computed periodically (tunable) and usually uses sampling (tunable)

Cardinality Estimator Errors

Estimator assumptions

- uniformity: all values, except for the most-frequent ones, are assumed to have the same number of tuples
- independence: predicates on attributes (in the same table or from joined tables) are independent
- principle of inclusion: the domains of the join keys overlap such that the keys from the smaller domain have matches in the larger domain

When these invariants don't hold – cardinality estimator has large errors



"How Good Are Query Optimizers, Really?"
http://www.vldb.org/pvldb/vol9/p204-leis.pdf

Cardinality Estimation Error Example

SELECT * FROM … JOIN …

WHERE city = ‘Seattle’

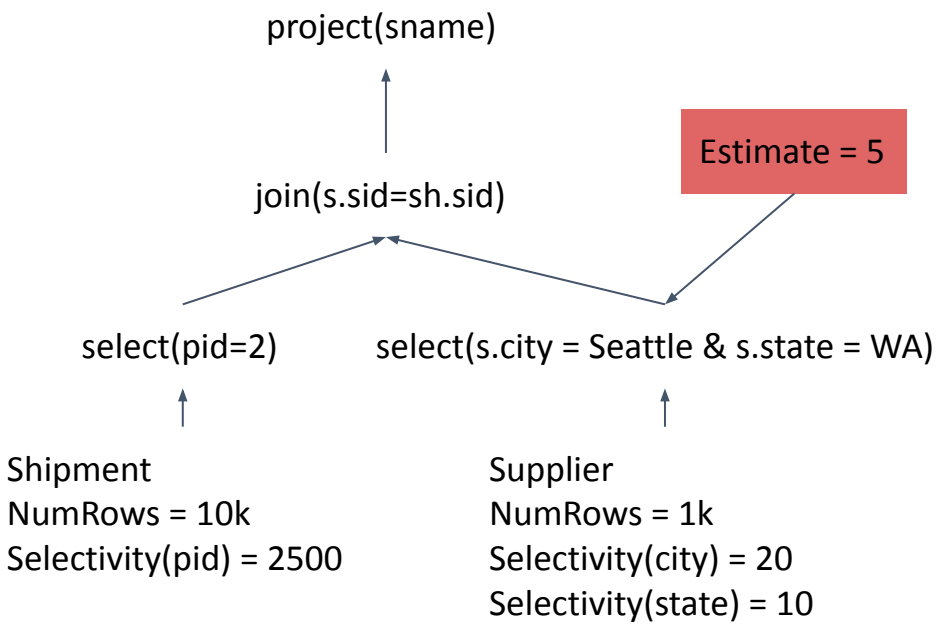
AND state = ‘Washington’

The city and state conditions are not independent. All records where city = Seattle will also have state = Washington.

The cardinality estimator does not know this, and severely underestimates the size of the result set.

Cardinality Estimation Error Example

```
SELECT sname
FROM Supplier s, Shipment Sh
WHERE s.sid = sh.sid
AND sh.pid = 2
AND s.city = 'Seattle'
AND s.state = 'WA'
```



Improving Cardinality Estimation w/ ML

- Input: query plan
- Output: cardinality estimate
- Training data: query execution statistics

Improving Cardinality Estimation w/ ML

- Map query plan nodes into feature vectors
  - Each predicate on a node is a feature name
  - Conflate: “age > 25” => “age > CONST”
  - DB histogram selectivity is feature value
- Use these features to train an ML model that can predict plan cardinality

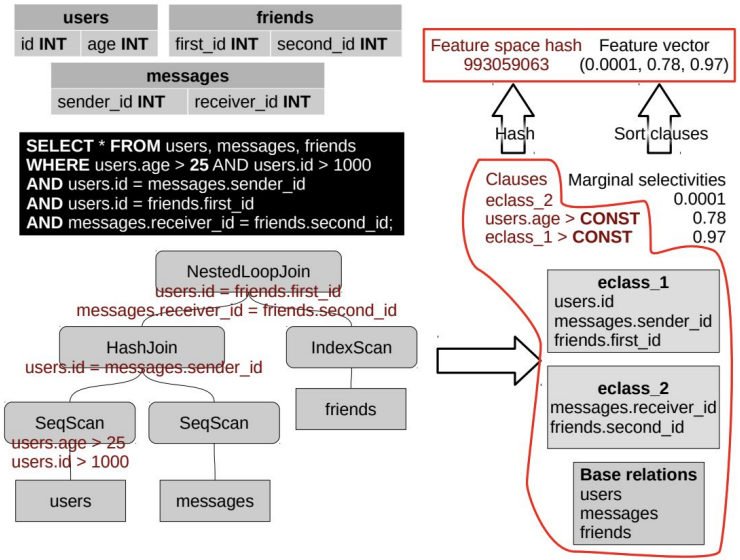
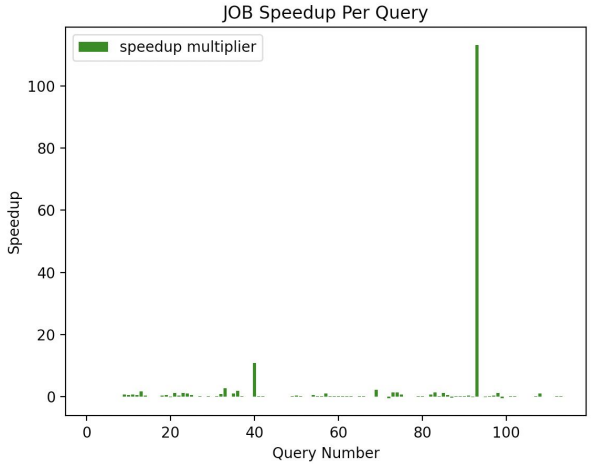


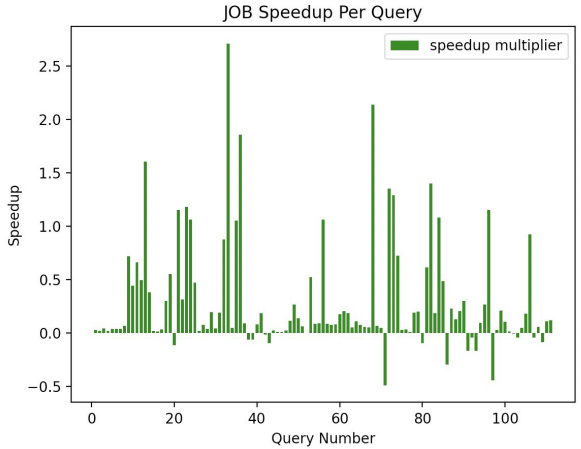
Figure 1: Building machine learning feature space

“Adaptive Cardinality Estimation”
https://arxiv.org/pdf/1711.08330.pdf

Adaptive Query Optimization

- Authors of the Adaptive Cardinality Estimation paper implemented their technique in a project called Adaptive Query Optimization or AQO
  - Implemented using a patch to postgres codebase + extension
- I trained an AQO model on the Join Order Benchmark and benchmarked result
  - AWS i3.xlarge: 30gb RAM + 4 vCPUs + NVMe SSD, shared_buffers = 8192mb





"Adaptive Cardinality Estimation"
https://arxiv.org/pdf/1711.08330.pdf

Query 26c

```sql
SELECT MIN(chn.name) AS character_name,
       MIN(mi_idx.info) AS rating,
       MIN(t.title) AS complete_hero_movie
FROM complete_cast AS cc,
     comp_cast_type AS cct1,
     comp_cast_type AS cct2,
     char_name AS chn,
     cast_info AS ci,
     info_type AS it2,
     keyword AS k,
     kind_type AS kt,
     movie_info_idx AS mi_idx,
     movie_keyword AS mk,
     name AS n,
     title AS t
WHERE cct1.kind = 'cast'
  AND cct2.kind LIKE '%complete%'
  AND chn.name IS NOT NULL
  AND (chn.name LIKE '%man%'
       OR chn.name LIKE '%Man%')
  AND it2.info = 'rating'
  AND k.keyword IN ('superhero', 'marvel-comics', 'based-on-comic', 'tv-special', 'fight',
                    'violence', 'magnet', 'web', 'claw', 'laser')
  AND kt.kind = 'movie' AND t.production_year > 2000 AND kt.id = t.kind_id
  AND t.id = mk.movie_id AND t.id = ci.movie_id AND t.id = cc.movie_id AND t.id = mi_idx.movie_id
  AND mk.movie_id = ci.movie_id AND mk.movie_id = cc.movie_id AND it2.id = mi_idx.info_type_id
  AND mk.movie_id = mi_idx.movie_id AND ci.movie_id = cc.movie_id AND ci.movie_id = mi_idx.movie_id
  AND cc.movie_id = mi_idx.movie_id AND chn.id = ci.person_role_id AND n.id = ci.person_id
  AND k.id = mk.keyword_id AND cct1.id = cc.subject_id AND cct2.id = cc.status_id;
```

Stock Query Plan

Settings

HTML | SOURCE | QUERY | REFORMATTED QUERY | STATS

#	exclusive	inclusive	rows x	rows	loops	node
1.	7.070	14,208.153	↑ 1.0	1	1	→ Aggregate (cost=10,733.32..10,733.33 rows=1 width=96) (actual time=14,208.113..14,208.153 rows=1 loops=1)
2.	1,505.320	14,201.083	↓ 5,400.0	5,400	1	→ Nested Loop (cost=7.68..10,733.31 rows=1 width=38) (actual time=9.362..14,201.083 rows=5,400 loops=1)
3.	1,481.580	9,259.317	↓ 63,637.9	1,145,482	1	→ Nested Loop (cost=7.26..10,725.18 rows=18 width=42) (actual time=7.620..9,259.317 rows=1,145,482 loops=1)
4.	691.628	6,858.137	↓ 1,045.0	10,450	1	→ Nested Loop (cost=6.83..10,707.40 rows=10 width=54) (actual time=7.608..6,858.137 rows=10,450 loops=1)
5.	980.390	5,394.677	↓ 1,581.6	385,916	1	→ Nested Loop (cost=6.40..10,595.71 rows=244 width=42) (actual time=0.255..5,394.677 rows=385,916 loops=1)
6.	514.435	2,870.623	↓ 1,581.6	385,916	1	→ Nested Loop (cost=5.97..10,484.61 rows=244 width=46) (actual time=0.242..2,870.623 rows=385,916 loops=1)
27.	3,436.446	3,436.446	↓ 0.0	0 - 1,145,482	1,145,482	→ Index Scan using keyword_pkey on keyword k (cost=0.42..0.45 rows=1 width=4) (actual time=0.003..0.003 rows=0 loops=1,145,482) Index Cond: (id = mk.keyword_id) Filter: (keyword = ANY ('{superhero,marvel-comics,based-on-comic,tv-special,fight,violence,magnet,web,claw,laser}'::text[])) Rows Removed by Filter: 1

https://explain.depesz.com/s/IBce

AQO Query Plan

Settings

HTML | SOURCE | QUERY | REFORMATTED QUERY | STATS

#	exclusive	inclusive	rows x	rows	loops	node
1.	5.973	1,042.093	↑ 1.0	1	1	→ Aggregate (cost=7,156.54..7,156.55 rows=1 width=96) (actual time=996.807..1,042.093 rows=1 loops=1)
2.	20.504	1,036.120	↓ 5,400.0	5,400	1	→ Nested Loop (cost=1,010.87..7,156.53 rows=1 width=39) (actual time=5.716..1,036.120 rows=5,400 loops=1)
3.	31.794	982.424	↓ 16,596.0	16,596	1	→ Nested Loop (cost=1,010.72..7,156.35 rows=1 width=43) (actual time=5.697..982.424 rows=16,596 loops=1) Join Filter: (t.id = mi_idx.movie_id)
4.	11.910	923.600	↓ 5,406.0	5,406	1	→ Nested Loop (cost=1,010.30..7,155.83 rows=1 width=49) (actual time=5.683..923.600 rows=5,406 loops=1)
5.	225.960	884.660	↓ 5,406.0	5,406	1	→ Gather (cost=1,009.87..7,155.38 rows=1 width=53) (actual time=5.668..884.660 rows=5,406 loops=1) Workers Planned: 2 Workers Launched: 2
6.	70.706	658.700	↓ 1,802.0	5,406	3 / 3	→ Nested Loop (cost=9.87..6,155.28 rows=1 width=53) (actual time=8.358..658.700 rows=1,802 loops=3)
14.	11.405	11.405	↑ 1.3	9 - 134,160	3 / 3	→ Parallel Seq Scan on keyword k (cost=0.03..1,787.59 rows=4 width=4) (actual time=0.216..11.405 rows=3 loops=3) Filter: (keyword = ANY ('{superhero,marvel-comics,based-on-comic,tv-special,fight,violence,magnet,web,claw,laser}'::text[])) Rows Removed by Filter: 44,720
15.	24.390	25.937	↓ 7.9	7,227	10 / 3	→ Bitmap Heap Scan on movie_keyword mk (cost=6.80..1,068.84 rows=305 width=8) (actual time=0.799..7.781 rows=2,409 loops=10) Recheck Cond: (keyword_id = k.id) Heap Blocks: exact=6,466
16.	1.547	1.547	↓ 7.9	7,227	10 / 3	→ Bitmap Index Scan on keyword_id_movie_keyword (cost=0.00..6.72 rows=305 width=0) (actual time=0.464..0.464 rows=2,409 loops=10) Index Cond: (keyword_id = k.id)

https://explain.depesz.com/s/zhG

BAO - Bandit Optimizer

Another research project that uses a different machine learning setting and technique to improve cardinality estimates.

Multi-Armed Bandits

The Multi-armed Bandit Problem – Problem Statement

You have:	**You want:**
A bag of quarters	Maximal Money
A row of slot machines	**You don't know:**
	Which slot machine is best

How do you balance **exploration** and **exploitation** in a way that **maximizes return** on your quarters spent?

Figure 1: Slot Machines!

"Application of Thompson Sampling to Multi-armed Bandits in Machine Learning" - Daniel Brice
https://mercurytechnologies.slack.com/archives/CPE2X5DMJ/p1614290691118300

Multi-Armed Bandits for Query Optimization

You Have
- A query to execute
- A set of possible query plans

You want
- Fastest execution time

You don’t know
- Which query plan is the best

You Have
- A query to execute
- A set of possible optimizer hints

You want
- Fastest execution time

You don’t know
- Which hints result in the best query plan

Contextual Multi-Armed Bandits

Just like a multi-armed bandit problem, but at every decision step you also have a feature vector representing the “context” of the environment you are in.

For example: (game=wheel_of_fortune, max_bet=$2, color=blue)

Definition 2.1 (Contextual bandit problem) *In a contextual bandits problem, there is a distribution P over $(x, r_1, ..., r_k)$, where x is context, $a \in \{1, \ldots, k\}$ is one of the k arms to be pulled, and $r_a \in [0, 1]$ is the reward for arm a. The problem is a repeated game: on each round, a sample $(x, r_1, ..., r_k)$ is drawn from P, the context x is announced, and then for precisely one arm a chosen by the player, its reward r_a is revealed.*

Definition 2.2 (Contextual bandit algorithm) *A contextual bandits algorithm $\mathcal{B}$ determines an arm $a \in \{1, \ldots, k\}$ to pull at each time step t, based on the previous observation sequence $(x_1, a_1, r_{a,1}), \ldots, (x_{t-1}, a_{t-1}, r_{a,t-1})$, and the current context x_t.*

“The Epoch-Greedy Algorithm for Contextual Multi-armed Bandits”
https://proceedings.neurips.cc/paper/2007/file/4b04a686b0ad13dce35fa99fa4161c65-Paper.pdf

Bandit Optimizer (BAO)

- Different strategy for learning an optimizer, which doesn't learn/predict cardinality estimates directly
- Bao learns which hints to provide to existing optimizer
  - e.g. SET enable_nestloop TO off;
  - This should have a similar effect to accurate cardinality estimation

```
imdb=# EXPLAIN SELECT * FROM ....

                                QUERY PLAN
 ------------------------------------------------------------------------

 Bao prediction: 61722.655 ms
 Bao recommended hint: SET enable_nestloop TO off;
                       (estimated 43124.023ms improvement)
 Finalize Aggregate  (cost=698026.88..698026.89 rows=1 width=64)
   ->  Gather  (cost=698026.66..698026.87 rows=2 width=64)
         ...
```

Figure 6: Example output from Bao's advisor mode.

"Bao: Making Learned Query Optimization Practical"
https://dl.acm.org/doi/pdf/10.1145/3448016.3452838

Bandit Optimizer (BAO)

- Map user-given query into set of query plans
  - One query plan for each set of optimizer hints
- Create one-hot vector for each tree node; append estimated cardinality and cost
- Use these features to train an ML model that predicts query execution time

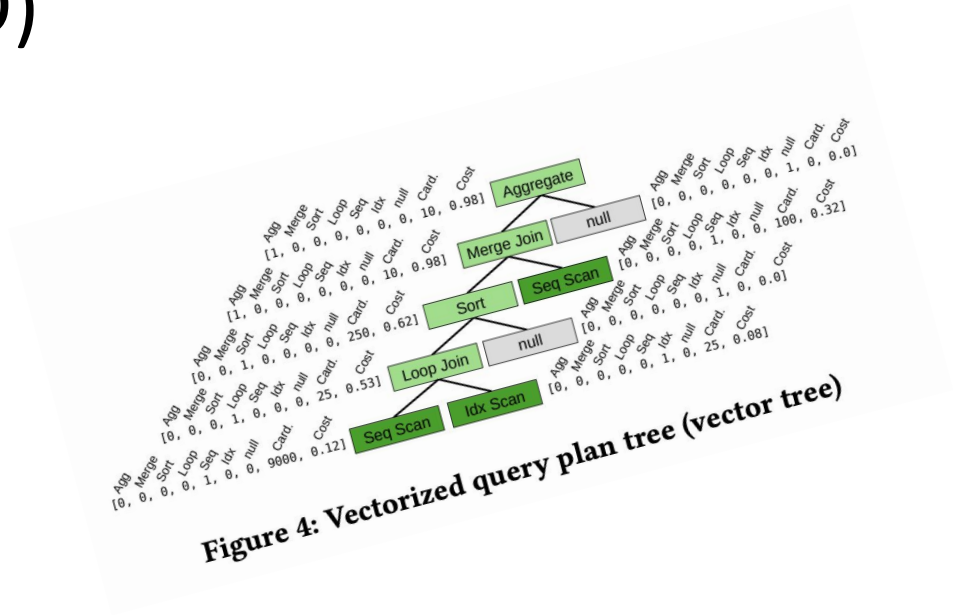


Figure 4: Vectorized query plan tree (vector tree)

"Bao: Making Learned Query Optimization Practical"
https://dl.acm.org/doi/pdf/10.1145/3448016.3452838

Questions?