Where do performance cliffs come from?

Tomas Vondra <tomas.vondra@enterprisedb.com>

EDB

Goal(s) of this talk

- discuss one class of performance issues
    - fairly common problem
    - affects cost-based optimization (inherent issue)
- explain why this happens
- maybe give some mitigation hints
    - but no promises, sorry :-(

What is a performance cliff?

- sudden (step) change of performance
- sudden = not proportional to change in "inputs"
- example
    - SELECT * FROM my_table WHERE column = $1
    - value "A" matches 1000 rows, query takes 1000 ms
    - value "B" matches 1050 rows, what duration is "expected"?
    - not much more than 1000ms? what if it takes 10000 ms?

Cost vs. Duration

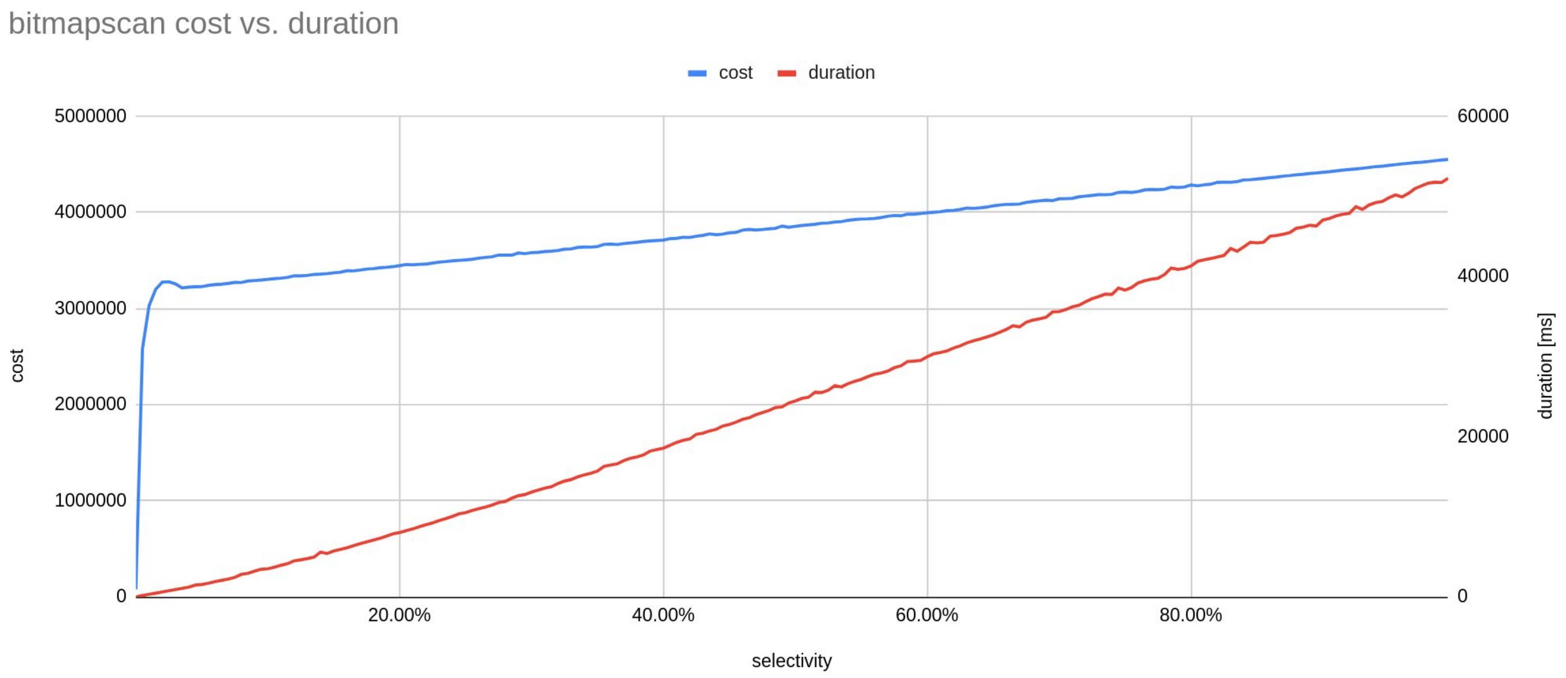
- most databases rely on cost estimates
  - how much "resources" will the plan require (CPU, I/O)
  - assumption: more resources => more time to execute
- cost is ...
  - monotonic and continuous function
  - ... with respect to costing parameters
  - ... selectivity of WHERE condition, number of groups, ...

Garbage in - garbage out

- selectivity estimates
- crucial input of the query planning process
- bogus estimate = anything can happen
- we assume selectivities are “good enough”

Example

small selectivity difference => small cost difference => small duration difference



Eh?! Where's the discontinuity?

- before: performance cliff is a sudden change in performance
- just now: cost is nice, smooth, without steps, ...

- cost is not timing, but should be correlated
- But why would the timing change in a step?

Ideas?

- ?
- ?
- ?
- ?

Ideas?

- cost is relies on estimates - if wildly wrong, anything can happen
- various things are ultimately decided at runtime
  - e.g. hashjoin / hashagg spilling, on-disk sort, ...
  - on/off decision - one row triggers a lot of work
- we're dealing with multiple plans
  - the whole point of why we calculate costs
  - cost and duration may not "align" perfectly

Runtime decisions

Example: ... IN (list)

```
CREATE TABLE test (a text);

INSERT INTO test
SELECT 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' || md5(random()::text)
  FROM generate_series(1,10000000) s(i);

VACUUM ANALYZE test;

-- table has ~965MB
```

Example: ... IN (list)

```
EXPLAIN (ANALYZE, TIMING OFF, COSTS OFF)
SELECT * FROM test WHERE a IN (
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac4ca4238a0b923820dcc509a6f75849b', -- 1
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac81e728d9d4c2f636f067f89cc14862c', -- 2
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaeccbc87e4b5ce2fe28308fd9f2a7baf3', -- 3
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa87ff679a2f3e71d9181a67b7542122c', -- 4
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaae4da3b7fbbce2345d7772b0674a318d5', -- 5
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa1679091c5a880faf6fb5e6087eb1b2dc', -- 6
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa8f14e45fceea167a5a36dedd4bea2543', -- 7
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac9f0f895fb98ab9159f51fd0297e236d', -- 8
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa45c48cce2e2d7fbdea1afc51c7c6ad26'  -- 9
);

==> 1000 ms
```

Example: ... IN (list)

```
EXPLAIN (ANALYZE, TIMING OFF, COSTS OFF)
SELECT * FROM test WHERE a IN (
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac4ca4238a0b923820dcc509a6f75849b', -- 1
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac81e728d9d4c2f636f067f89cc14862c', -- 2
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaeccbc87e4b5ce2fe28308fd9f2a7baf3', -- 3
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa87ff679a2f3e71d9181a67b7542122c', -- 4
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaae4da3b7fbbce2345d7772b0674a318d5', -- 5
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa1679091c5a880faf6fb5e6087eb1b2dc', -- 6
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa8f14e45fceea167a5a36dedd4bea2543', -- 7
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac9f0f895fb98ab9159f51fd0297e236d'  -- 8
);

==> 2000 ms (EH?! twice the timing of a longer IN list?)
```

Example: ... IN (list)

```
                          QUERY PLAN
----------------------------------------------------------------
 Seq Scan on test (actual rows=0 loops=1)
   Filter: (a = ANY ('{aaaaaaaaaaaaaaaaaaaaaaa..., ...}'::text[]))
   Rows Removed by Filter: 10000000
 Planning Time: 0.092 ms
 Execution Time: 1386.788 ms
(5 rows)
```

Example: ... IN (list)

- lookup in hash table with >= 9 elements
  - fewer elements => linear search
  - but 9 is hard-coded threshold
- ideal threshold depends on cost of comparison
  - specific to data-type and values (e.g. long prefix like here)
  - impossible to know in advance / during execution

Other runtime decisions

- query with in-memory vs. on-disk sort
- query with hashjoin/hashagg in memory vs. spilling to disk
- JIT can be quite expensive & useless
  - enabled depending on total cost of a query
  - ongoing effort to make more granular

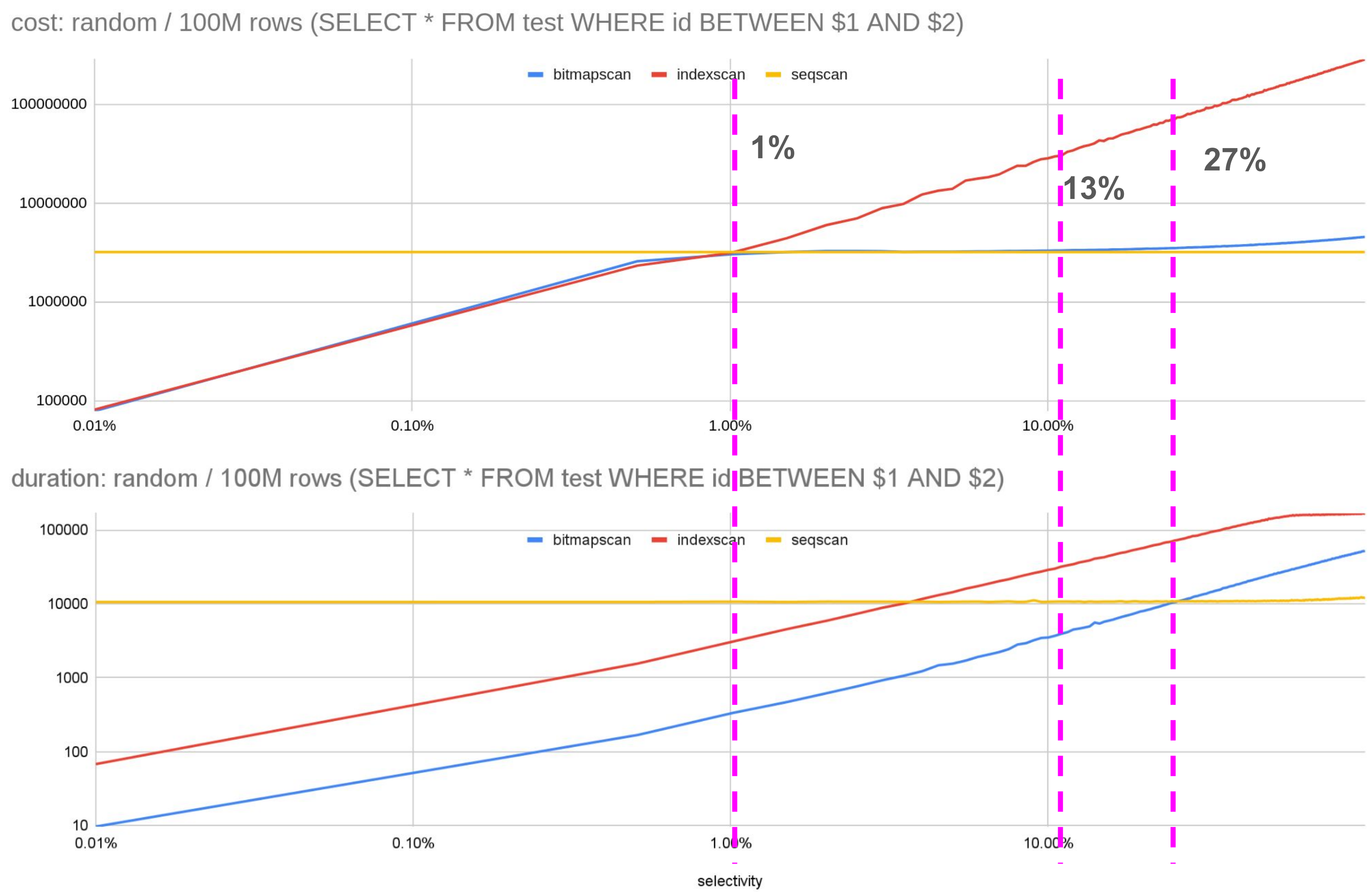
Path switch

100M rows, random data

```
CREATE TABLE test (a INT, b TEXT) WITH (fillfactor=50);

-- 59 rows/page, each page has the same (random) value
INSERT INTO test SELECT a, b FROM (
    SELECT a, b, generate_series(1,59) FROM (
        SELECT 10_000 * random() a,
               md5(random()::text) b
        FROM generate_series(1, 100_000_000/59)
    ) AS x
) AS y;

CREATE INDEX ON test (a);
```


cost: random / 100M rows (SELECT * FROM test WHERE id BETWEEN $1 AND $2)
bitmapscan
indexscan
seqscan
100000000
10000000
1000000
100000
0.01%
0.10%
1.00%
10.00%
1%
13%
27%
duration: random / 100M rows (SELECT * FROM test WHERE id BETWEEN $1 AND $2)
bitmapscan
indexscan
seqscan
100000
10000
1000
100
10
0.01%
0.10%
1.00%
10.00%
selectivity

```
SELECT * FROM test WHERE id BETWEEN 1000 AND 1127;
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on test (actual rows=1293280 loops=1)
   Recheck Cond: ((id >= 1000) AND (id <=1127))
   Heap Blocks: exact=21920
   ->  Bitmap Index Scan on test_id_idx (actual rows=1293280 loops=1)
          Index Cond: ((id >= 1000) AND (id <= 1127))
 Planning Time: 9.268 ms
 Execution Time: 412.993 ms
(7 rows)

SELECT * FROM test WHERE id BETWEEN 1000 AND1128;
                    QUERY PLAN
------------------------------------------------
 Seq Scan on test (actual rows=1301894 loops=1)
   Filter: ((id >= 1000) AND (id <= 1128))
   Rows Removed by Filter: 98698091
 Planning Time: 8.289 ms
 Execution Time: 10706.679 ms
(5 rows)
```

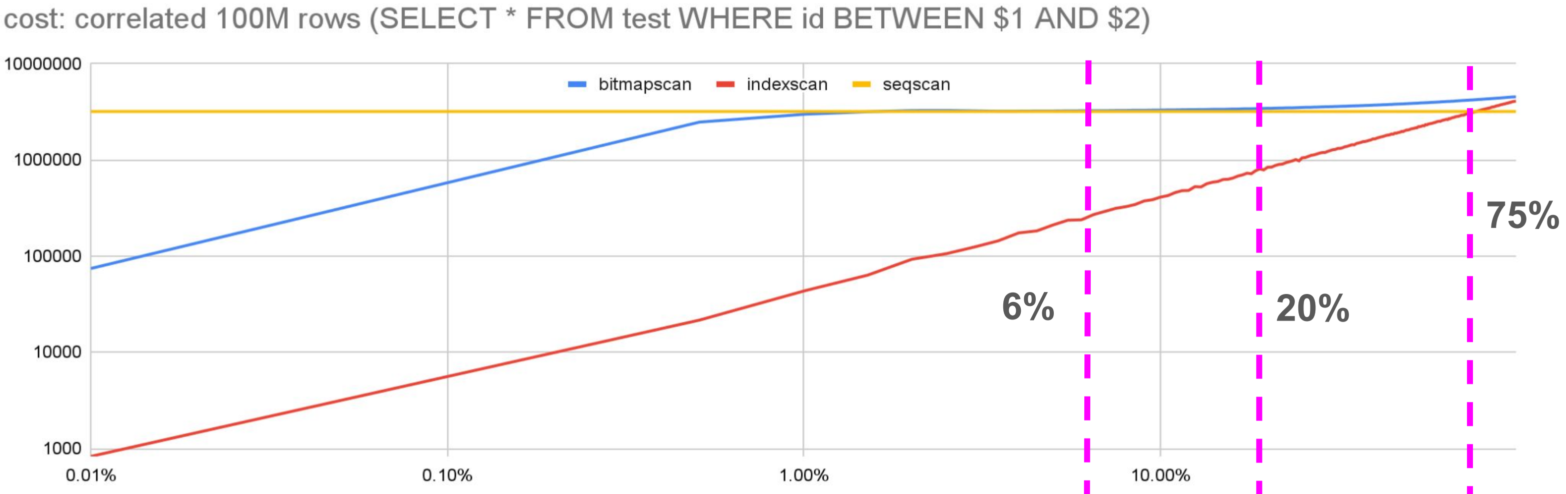
EDB

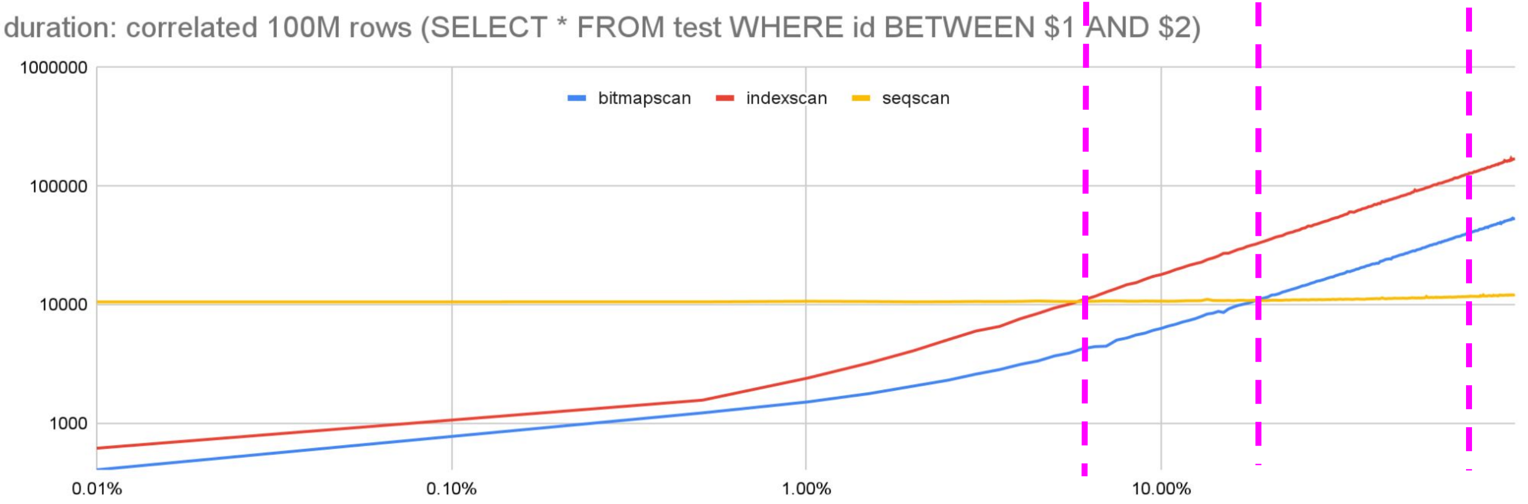
100M rows, sequential/correlated data

```
CREATE TABLE test (a INT, b TEXT) WITH (fillfactor=50);

-- monotonic growth, with a bit of random "fuzz"
INSERT INTO test
SELECT (i * 1.0 * 10_000) / 100_000_000 +
       (10_000 * (random() - 0.5)) / 50,
       md5(random()::text)
FROM generate_series(1, 100_000_000) s(i);

CREATE INDEX ON test (a);
```


cost: correlated 100M rows (SELECT * FROM test WHERE id BETWEEN $1 AND $2)
bitmapscan
indexscan
seqscan
10000000
1000000
100000
10000
1000
0.01%
0.10%
1.00%
10.00%
6%
20%
75%


duration: correlated 100M rows (SELECT * FROM test WHERE id BETWEEN $1 AND $2)
bitmapscan
indexscan
seqscan
1000000
100000
10000
1000
0.01%
0.10%
1.00%
10.00%

```
select * from test where id between 1000 and 8650;
                    QUERY PLAN
-------------------------------------------------
 Seq Scan on test (actual rows=76510346 loops=1)
   Filter: ((id >= 1000) AND (id <= 8650))
   Rows Removed by Filter: 23489654
 Planning Time: 0.072 ms
 Execution Time: 11905.432 ms
(5 rows)

select * from test where id between 1000 and 8600;
                                QUERY PLAN
---------------------------------------------------------------------
 Index Scan using test_id_idx on test (actual rows=76009271 loops=1)
   Index Cond: ((id >= 1000) AND (id <= 8600))
 Planning Time: 8.398 ms
 Execution Time: 130789.542 ms
(4 rows)
```

Mitigations?

Mitigations

- really hard to fix (during planning)
- inherent to cost-based planning in general
- costing is approximation
  - simplified model + incomplete data => imperfection
  - G. Graefe: "choice is confusion" [1]
- So, what options do you have?

Mitigations

- try to ensure the "flip" does not trigger
  - increase work_mem, for example
  - it "only" moves the threshold ahead
- try to reduce the impact of the "flip"
  - fast but ephemeral storage for temp files?
  - ...

Mitigations

- bit of tuning the cost parameters?
  - random_page_cost, cpu_tuple_cost, ...
  - can the cost / duration charts align better?
- don't bother to fine-tune the parameter values
  - no parameter value is perfect for all queries
  - the flip needs to happen "close enough"
- some important parameters do not affect costing
  - e.g. effective_io_concurrency

Would be better ...

- adaptive execution
  - replace "a priori" decisions with exec time ones
  - ideal: adaptive, smooth transition, not just on/off
  - example: scan type selection vs. "Smooth Scan"
- might also help with estimation errors
- replacement for implementations of a logical node
  - one for scans, another for joins, ...

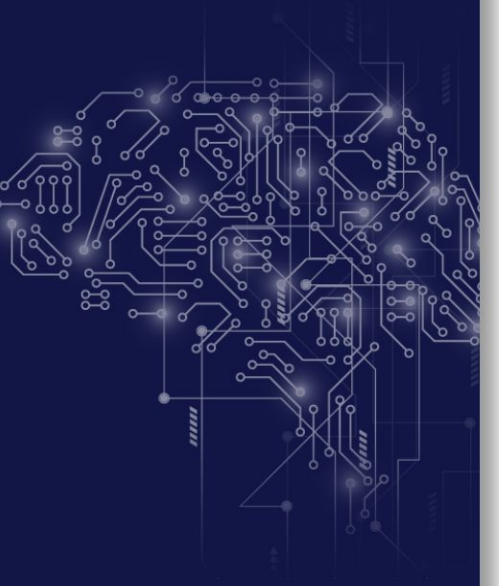
Robustness / Research papers ...

- Smooth Scan: One Access Path to Rule Them All
  R. Borovica, S. Idreos, A. Ailamaki, M. Zukowski, C. Fraser
  https://stratos.seas.harvard.edu/files/stratos/files/smoothscan.pdf

- A generalized join algorithm
  G. Graefe
  https://dl.gi.de/server/api/core/bitstreams/ce8e3fab-0bac-45fc-a6d4-66edaa52d574/content

- Profile of G. Graefe
  https://sigmodrecord.org/publications/sigmodRecord/2009/pdfs/05_Profiles_Graefe.pdf

EDB™



POSTGRESQL 16 ADMINISTRATION COOKBOOK

Take part in EDB's prize draw
to win a brand new book
PostgreSQL 16 Administration Cookbook.

SCAN THE QR CODE AND FILL IN THE FORM TO ENTER

enterprisedb.com