Where do performance cliffs come from?

Tomas Vondra <vondratomas@microsoft.com> / <tomas@vondra.me>

Malmö Meetup, April 24, 2025

Microsoft

About me

- developer, contributor, committer
- Microsoft
- tomas@vondra.me
- https://vondra.me
- office hours

Agenda

Microsoft

- intro
    - What is a performance cliff?
- runtime decisions
    - A simple example of a performance cliff.
- multiple paths
    - Performance cliffs related to cost-based planning.
- mitigations
    - What can we do about this?

Performance cliffs

- multiple / ambiguous definitions
  - sudden change of performance
- a class of performance (robustness) issues
  - fairly common problem (but somewhat hidden)
  - affects cost-based planning (inherent issue)
- why it happens?
- what can you do about it?
  - mitigation ideas (but no promises)
  - ideas - patches / development / research

What is a performance cliff?

- sudden (step) change of performance
  - sudden = not proportional to change in "inputs"
  - input = selectivity of a condition

```
SELECT * FROM my_table WHERE column = $1
```

  - $1 = 'A': 1000 rows, duration 1,000 ms
  - $1 = 'B': 1001 rows, duration ??? ms
  - ~1,000 ms? What if it's 10,000 ms?

Sources of discontinuity?

- flips between different "execution strategies"
- various things are ultimately decided at runtime
  - on/off decision - one row may trigger a lot of work
  - e.g. hashjoin / hashagg spilling, on-disk sort, ...
- switching to a different "path" (ways to execute query)
  - the whole point of why we calculate costs
  - cost and duration may not "align" perfectly

Runtime decisions

Example: ... IN (list)

```
SELECT * FROM test WHERE a IN ('aaaaaa...a', ..., 'aaaaaa...x');

-- table has ~965MB
-- random strings with long prefixes (expensive comparisons)
CREATE TABLE test (a text);

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

VACUUM ANALYZE test;
```

Example: ... IN (list)

```
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)

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

How long will this take?

Example: ... IN (list)

```
SELECT * FROM test WHERE a IN (
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac4ca4238a0b923820dcc509a6f75849b', -- 1
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac81e728d9d4c2f636f067f89cc14862c', -- 2
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaeccbc87e4b5ce2fe28308fd9f2a7baf3', -- 3
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa87ff679a2f3e71d9181a67b7542122c', -- 4
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaae4da3b7fbbce2345d7772b0674a318d5', -- 5
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa1679091c5a880faf6fb5e6087eb1b2dc', -- 6
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa8f14e45fceea167a5a36dedd4bea2543', -- 7
  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac9f0f895fb98ab9159f51fd0297e236d'  -- 8
);
```

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

Example: ... IN (list)

- two strategies
  - short list => linear seach
  - long list => hash table
- hard-coded threshold of 9 items for hash table
  - seems reasonable ...
- ideal threshold depends on cost of a comparison
  - specific to data-type and values (e.g. long prefix like here)
  - impossible to know in advance / during execution

Other runtime decisions ...

- in-memory vs. on-disk
  - sort
  - hashjoin
  - hashagg
- JIT can be quite expensive & useless
  - enabled depending on total cost of a query
  - ongoing effort to improve (planning & execution)

Multiple paths

Cost-based planning

- plan cost
  - amount of "resources" used byt plan (CPU, I/O)
  - more resources → higher cost → higher duration
- assumptions about cost
  - monotonic & continuous
  - w.r.t. to inputs (selectivity) and outputs (duration)
- we assume estimates are correct (for this talk)
  - bogus estimates → arbitrarily wrong plan

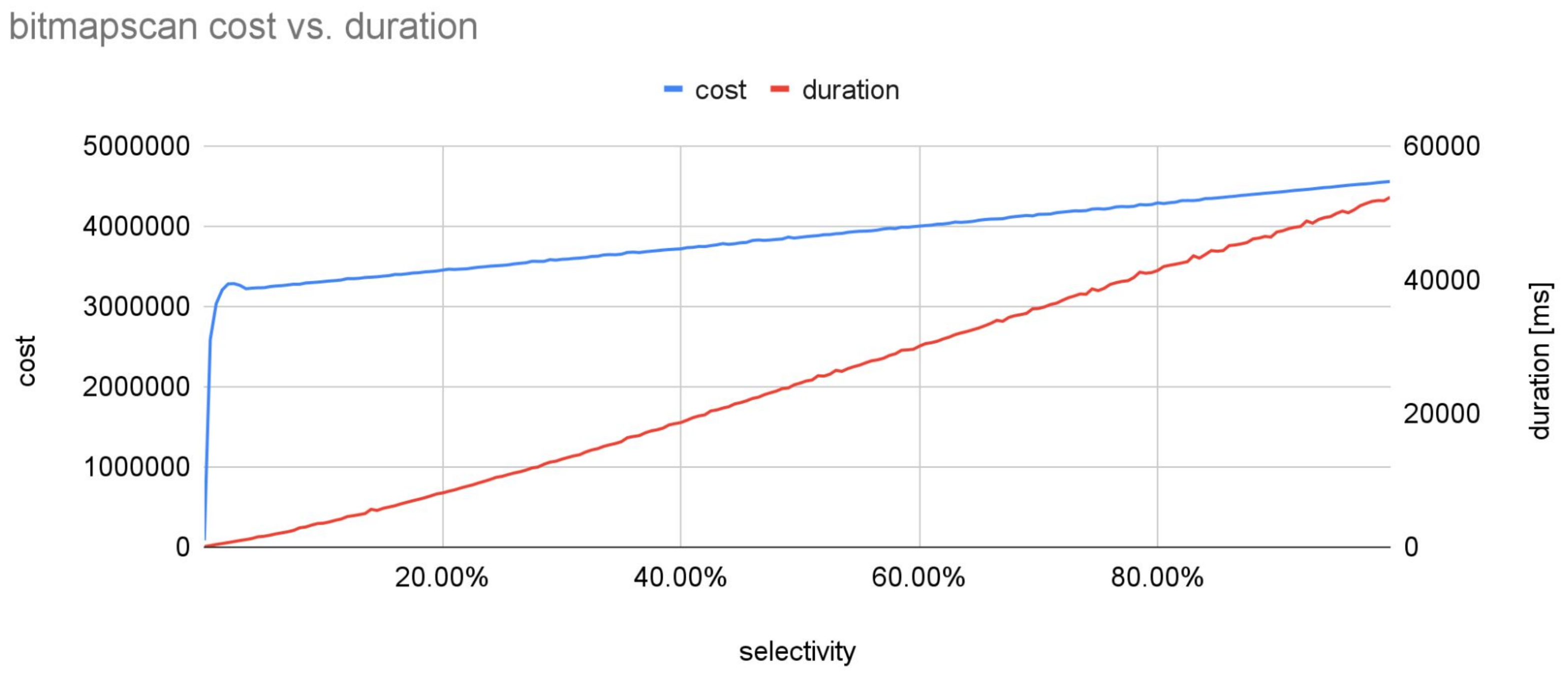
Visualization

selectivity	cost	duration
1%	24091.4	72.586
2%	35875.6	93.345
...	...	...
100%	74189.1	642.525

Microsoft

```
SET enable_indexscan = off;
SET enable_seqscan = off;
SET max_parallel_workers_per_gather = 0;

EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM test WHERE  a BETWEEN 100 AND 200;   ==> 1%

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=137.66..24091.40 rows=9877 width=4)
                           (actual rows=10039 loops=1)
   Recheck Cond: ((a >= 100) AND (a <= 200))
   Heap Blocks: exact=8983
   ->  Bitmap Index Scan on test_a_idx  (cost=0.00..135.19 rows=9877 width=0)
                                        (actual rows=10039 loops=1)
     Index Cond: ((a >= 100) AND (a <= 200))
 Planning Time: 0.212 ms
 Execution Time: 72.586 ms
(7 rows)
```

SELECT * FROM t WHERE a BETWEEN $1 AND $2

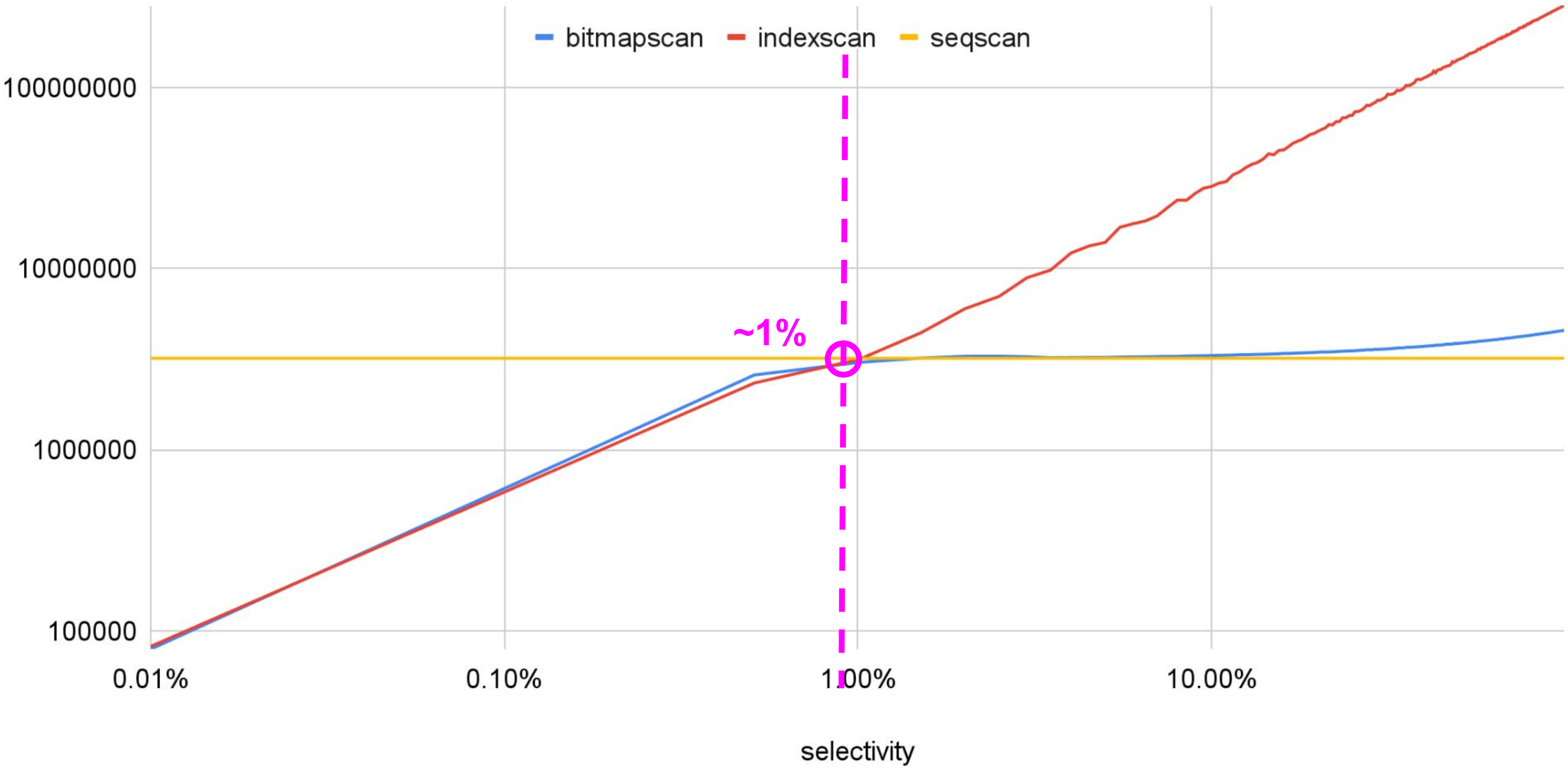


100M rows, random data

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

-- 13GB table, 10k distinct values
-- 59 rows/page, each page has the same 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);
```

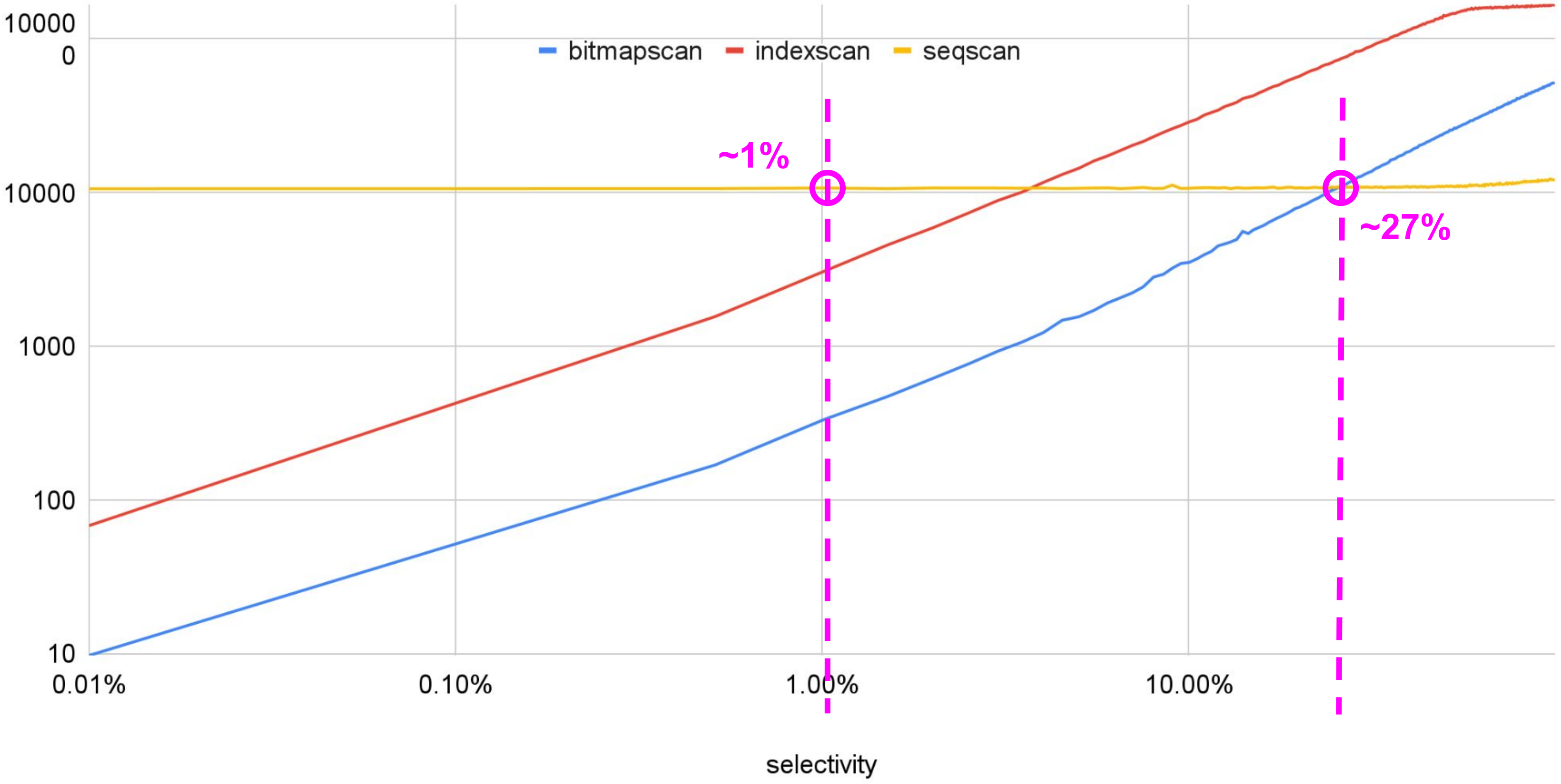
cost: random / 100M rows

SELECT * FROM test WHERE id BETWEEN $1 AND $2



duration: random / 100M rows

SELECT * FROM test WHERE id BETWEEN $1 AND $2



```
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 AND 1128;
                   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)
```

Mitigations?

Mitigations

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

Making the cliff smaller

- ensure the "flip" does not happen
  - e.g. increase work_mem to do in-memory sorts
  - it "only" moves the threshold ahead
- reduce the impact of the "flip"
  - fast but ephemeral storage for temp files?
  - ...

Tuning cost model

- tune basic cost parameters
  - random_page_cost, cpu_tuple_cost, ...
  - try to align cost / duration charts better
- don't bother to fine-tune the parameter values
  - "ideal" values are query-specific
  - the flip needs to happen "close enough"

Future / Patch ideas

- 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"
- would also help with estimation errors
- performance vs. robustness

Robustness / Research papers ...

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

[2] Smooth Scan: Robust Access Path Selection without Cardinality Estimation
R. Borovica, S. Idreos, A. Ailamaki, M. Zukowski, C. Fraser
https://stratos.seas.harvard.edu/files/stratos/files/smoothscan.pdf
https://scholar.harvard.edu/files/stratos/files/smooth_vldbj.pdf

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

Robustness / Research papers ...

Microsoft

Dagstuhl seminars / Robust Performance in Database Query Processing

- 2010
  https://www.dagstuhl.de/en/seminars/seminar-calendar/seminar-details/10381
- 2012
  https://www.dagstuhl.de/en/seminars/seminar-calendar/seminar-details/12321
- 2017
  https://www.dagstuhl.de/en/seminars/seminar-calendar/seminar-details/17222
- 2022
  https://www.dagstuhl.de/en/seminars/seminar-calendar/seminar-details/22111
- 2024
  https://www.dagstuhl.de/en/seminars/seminar-calendar/seminar-details/24101

Joining the community

- pgsql-hackers
- my office hours
- hacking workshop & mentoring
  - https://rhaas.blogspot.com/
  - https://discord.gg/gyDQBeZA
- https://planet.postgresql.org