

THINGS YOUR EXPLAIN PLAN IS NOT TELLING YOU

ANTS AASMA

pgconf.in 2025



Hello



About me

- Ants Aasma
- Lead Database Consultant @ Cybertec
- PostgreSQL contributor
- 13 years of helping people make PostgreSQL run fast





OpenSource DB + CYBERTEC <https://www.cybertec-postgresql.com/en/>

Our strategic and exclusive partnership with OpenSource DB brings together deep community expertise and cutting-edge PostgreSQL solutions, reinforcing our commitment to delivering comprehensive, end-to-end database services.

OpenSource DB provides:

- Expertise in PostgreSQL Consulting and Professional Services
- Comprehensive Training and Workshops
- Customized Database Solutions
- Global Reach and Experience
- Focus on Open-source



+





Modern **database scaling platform** and **SQL query caching** system.

Boost performance, optimize infrastructure costs, and improve engineering velocity with proactive enhancements for PostgreSQL & MySQL

Reduce hosting costs **by up to 70%**

In memory query read cache - sub millisecond

Auto synced to source db changes **in milliseconds**

No **TTLs**, no **cache invalidations**

Everybody loves explain



Explain yourself

- EXPLAIN tells us how the database planned to execute our query
- EXPLAIN ANALYZE collects statistics how well that went



Explain yourself

- EXPLAIN tells us how the database planned to execute our query
- EXPLAIN ANALYZE collects statistics how well that went
- If you are really curious, then:

EXPLAIN (ANALYZE, VERBOSE, SETTINGS, BUFFERS, WAL, SUMMARY, MEMORY, SERIALIZE



Explain yourself

- EXPLAIN tells us how the database planned to execute our query
- EXPLAIN ANALYZE collects statistics how well that went
- If you are really curious, then:

EXPLAIN (ANALYZE, VERBOSE, SETTINGS, BUFFERS, WAL, SUMMARY, MEMORY, SERIALIZE

- Maybe it's time for EXPLAIN EVERYTHING?



What are we going to talk about

- Explain is great!
- Everybody should be using it.



What are we going to talk about

- Explain is great!
- Everybody should be using it.
- This talk is about the parts that are not (yet) great.



Warning

- This talk will have code.



Warning

- This talk will have code.
- A lot of code.



Warning

- This talk will have code.
- A lot of code.
- Like really a lot of it.

```
SELECT 'If this is too small, you need to try to get closer';
```

Slides will be available on the conference website.

Or get it now from 2024.pgconf.de



Crash course on reading EXPLAIN



Parts of an explain plan

- Represents the tree of a volcano execution model.
- Each node pulls from those below it.
- Rows come from the inside out

```
EXPLAIN SELECT * FROM tasks JOIN jobs USING (job_id) WHERE value > 99.9 LIMIT 100;
```



Parts of an explain plan

- Represents the tree of a volcano execution model.
- Each node pulls from those below it.
- Rows come from the inside out

```
EXPLAIN SELECT * FROM tasks JOIN jobs USING (job_id) WHERE value > 99.9 LIMIT 100;
```

```
Limit (cost=0.42..149.83 rows=100 width=44)
  -> Nested Loop (cost=0.42..716.10 rows=479 width=44)
    -> Seq Scan on jobs (cost=0.00..188.00 rows=8 width=16)
        Filter: (value > 99.9)
    -> Index Scan using tasks_job_id_id_done_idx on tasks
        (cost=0.42..65.41 rows=60 width=32)
        Index Cond: (job_id = jobs.job_id)
```



Running it

```
EXPLAIN ANALYZE
```

```
SELECT * FROM tasks JOIN jobs USING (job_id) WHERE value > 99.9 LIMIT 100;
```



Running it

EXPLAIN ANALYZE

```
SELECT * FROM tasks JOIN jobs USING (job_id) WHERE value > 99.9 LIMIT 100;
```

```
Limit (cost=0.42..149.83 rows=100 width=44) (actual time=0.153..0.279 rows=100 loops=1)
```

```
-> Nested Loop (cost=0.42..716.10 rows=479 width=44) (actual time=0.152..0.272 rows=100 loops=1)
```

```
  -> Seq Scan on jobs (cost=0.00..188.00 rows=8 width=16) (actual time=0.144..0.167 rows=8 loops=1)
```

```
    Filter: (value > 99.9)
```

```
    Rows Removed by Filter: 1180
```

```
  -> Index Scan using tasks_job_id_id_done_idx on tasks (cost=0.42..65.41 rows=60 width=44) (actual time=0.008..0.010 rows=60 loops=1)
```

```
    Index Cond: (job_id = jobs.job_id)
```

```
Planning Time: 0.188 ms
```

```
Execution Time: 0.299 ms
```



Running it

```
EXPLAIN (ANALYZE, COSTS OFF)
SELECT * FROM tasks JOIN jobs USING (job_id) WHERE value > 99.9 LIMIT 100;
```

```
Limit (actual time=0.153..0.279 rows=100 loops=1)
  -> Nested Loop (actual time=0.152..0.272 rows=100 loops=1)
    -> Seq Scan on jobs (actual time=0.144..0.167 rows=2 loops=1)
        Filter: (value > 99.9)
        Rows Removed by Filter: 1180
    -> Index Scan using tasks_job_id_id_done_idx on tasks
        (actual time=0.006..0.045 rows=50 loops=2)
        Index Cond: (job_id = jobs.job_id)
Planning Time: 0.188 ms
Execution Time: 0.299 ms
```



Buffers

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT COUNT(*) FROM tasks;
```

```
Aggregate (actual time=56.689..56.690 rows=1 loops=1)
```

```
  Buffers: shared hit=2519 read=2153
```

```
  I/O Timings: shared read=2.774
```

```
    -> Seq Scan on tasks (actual time=0.008..35.051 rows=599524 loops=1)
```

```
      Buffers: shared hit=2519 read=2153
```

```
      I/O Timings: shared read=2.774
```

```
Planning Time: 0.059 ms
```

```
Execution Time: 56.711 ms
```



Buffers

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT COUNT(*) FROM tasks;
```

```
Aggregate (actual time=56.689..56.690 rows=1 loops=1)
  Buffers: shared hit=2519 read=2153
  I/O Timings: shared read=2.774
  -> Seq Scan on tasks (actual time=0.008..35.051 rows=599524 loops=1)
    Buffers: shared hit=2519 read=2153
    I/O Timings: shared read=2.774
Planning Time: 0.059 ms
Execution Time: 56.711 ms
```

- read means from OS, can't tell if it came from disk or not.
 - I/O Timings help, always set `track_io_timing = on`



Chapter 1: Why am I smelling TOAST



We need a schema

```
CREATE TABLE reports (  
  report_id int primary key,  
  ruleset_id int not null,  
  data jsonb not null -- {"metric1": 0.42, ..., "metric1000": 0.123}  
);  
CREATE TABLE rules (  
  rule_id int primary key,  
  ruleset_id int not null,  
  rule_nr int not null,  
  metric_field text not null,  
  max_value real not null -- reports.data->metric_field <= max_value  
);  
CREATE INDEX ON reports (ruleset_id);  
CREATE INDEX ON rules (ruleset_id);
```



And some data

```
-- 10000 reports with 1000 metrics each
INSERT INTO reports
SELECT id,
       floor(random()*100)+1 ruleset_id,
       (SELECT jsonb_object_agg('metric' || metric::text, random())
        FROM generate_series(1,1000) metric)
FROM generate_series(1, 10000) id;
-- 100 rulesets with 10 rules each
INSERT INTO rules
SELECT row_number() over (),
       ruleset_id,
       rule_nr,
       'metric' || floor(random()*1000 + 1)::text metric_field,
       0.95 + 0.1*random() max_value
FROM generate_series(1, 100) ruleset_id, generate_series(1,10) rule_nr;
```



The data

```
SELECT pg_size_pretty(avg(length(data::text))) avg_size,  
       pg_size_pretty(sum(length(data::text))) total_data_size,  
       pg_size_pretty(pg_total_relation_size('reports')) total_table_size  
FROM reports;
```

avg_size	total_data_size	total_table_size
32 kB	316 MB	159 MB



Lets read the data

```
EXPLAIN (ANALYZE, COSTS OFF)
SELECT * FROM reports;
```

- 1 Seq Scan on reports (actual time=0.008..0.591 rows=10000 loops=1)
- 2 Planning Time: 0.053 ms
- 3 Execution Time: **0.868 ms**



Lets read the data

```
EXPLAIN (ANALYZE, COSTS OFF)
SELECT * FROM reports;
```

```
1  Seq Scan on reports (actual time=0.008..0.591 rows=10000 loops=1)
2  Planning Time: 0.053 ms
3  Execution Time: 0.868 ms
```

- 316MB in 0.9ms → 339 GB/s ...
 - That's suspiciously fast...
 - Lets double check



Actually read the data

```
1 \timing on
2 \copy ( SELECT * FROM reports ) TO '/dev/null'
3 COPY 10000
4 Time: 1687.562 ms (00:01.688)
```



What's going on

- Large values are split up into chunks and stored in a secondary table (toasting)
- Main table contains only the identifier
- Value is transparently read in as needed. (detoasting)
- EXPLAIN ANALYZE doesn't need it.
 - The data is not serialized so detoasting is not triggered.



Fixed in PostgreSQL 17

```
EXPLAIN (ANALYZE, COSTS OFF, SERIALIZE TEXT)
SELECT * FROM reports;
```

```
1  Seq Scan on reports (actual time=0.009..0.728 rows=10000 loops=1)
2  Planning Time: 0.040 ms
3  Serialization: time=1169.736 ms output=323787kB format=text
4  Execution Time: 1171.082 ms
```



Detoasting can be anywhere

```
SELECT report_id, rule_id
FROM reports JOIN rules USING (ruleset_id)
WHERE (data->metric_field)::real > max_value AND rule_nr = 1;
```



Detoasting can be anywhere

```
SELECT report_id, rule_id
FROM reports JOIN rules USING (ruleset_id)
WHERE (data->metric_field)::real > max_value AND rule_nr = 1;
```

```
1 Merge Join (actual time=82.500..299.761 rows=108 loops=1)
2   Merge Cond: (rules.ruleset_id = reports.ruleset_id)
3   Join Filter: (((reports.data -> rules.metric_field))::real > rules.max_value)
4   Rows Removed by Join Filter: 9892
5   Buffers: shared hit=28351 read=17915 written=1
6   -> Index Scan using rules_ruleset_id_idx on rules (actual time=0.030..0.326 rows=100 loops=1)
7       Filter: (rule_nr = 1)
8       Rows Removed by Filter: 900
9       Buffers: shared hit=748 read=3
10  -> Index Scan using reports_ruleset_id_idx on reports (actual time=0.010..3.275 rows=10000 loops=1)
11      Buffers: shared hit=5515
12 Execution Time: 299.787 ms
```



Detoasting can be anywhere

```
SELECT report_id, rule_id
FROM reports JOIN rules USING (ruleset_id)
WHERE (data->metric_field)::real > max_value AND rule_nr = 1;
```

```
1 Merge Join (actual time=82.500..299.761 rows=108 loops=1)
2   Merge Cond: (rules.ruleset_id = reports.ruleset_id)
3   Join Filter: (((reports.data -> rules.metric_field))::real > rules.max_value)
4   Rows Removed by Join Filter: 9892
5   Buffers: shared hit=28351 read=17915 written=1
6   -> Index Scan using rules_ruleset_id_idx on rules (actual time=0.030..0.326 rows=100 loops=1)
7       Filter: (rule_nr = 1)
8       Rows Removed by Filter: 900
9       Buffers: shared hit=748 read=3
10  -> Index Scan using reports_ruleset_id_idx on reports (actual time=0.010..3.275 rows=10000 loops=1)
11      Buffers: shared hit=5515
12 Execution Time: 299.787 ms
```



Detoasting is not cached

```
/*+ MergeJoin(reports rules) Leading(reports rules) */  
SELECT report_id, rule_id  
FROM reports JOIN rules USING (ruleset_id)  
WHERE (data->metric_field)::real > max_value
```



Detoasting is not cached

```
/*+ MergeJoin(reports rules) Leading(reports rules) */  
SELECT report_id, rule_id  
FROM reports JOIN rules USING (ruleset_id)  
WHERE (data->metric_field)::real > max_value
```

```
1 Merge Join (actual time=45.499..2579.457 rows=1593 loops=1)  
2 Merge Cond: (rules.ruleset_id = reports.ruleset_id)  
3 Join Filter: (((reports.data -> rules.metric_field))::real > rules.max_value)  
4 Rows Removed by Join Filter: 98407  
5 Buffers: shared hit=436246 read=20105  
6 -> Index Scan using rules_ruleset_id_idx on rules (actual time=0.042..0.411 rows=1000 loops=  
7 Buffers: shared hit=740 read=11  
8 -> Index Scan using reports_ruleset_id_idx on reports (actual time=0.012..22.575 rows=99991  
9 Buffers: shared hit=55590 read=10  
10 Execution Time: 2579.533 ms
```



How to spot detoasting

- Look for unreasonably high buffer accesses.
- Look for large columns used in predicates and function calls (VERBOSE helps)



Example of early detoasting

```
EXPLAIN (BUFFERS, VERBOSE, ANALYZE, COSTS OFF)
SELECT LENGTH(data::text) FROM reports ORDER BY random() LIMIT 100;
```

```
1  Limit (actual time=1291.706..1291.725 rows=100 loops=1)
2    Output: ((data)::text), (random())
3    Buffers: shared hit=20342 read=19732
4    -> Sort (actual time=1291.704..1291.717 rows=100 loops=1)
5        Output: ((data)::text), (random())
6        Sort Key: (random())
7        Sort Method: top-N heapsort Memory: 3945kB
8        -> Seq Scan on public.reports (actual time=0.210..1279.087 rows=10000 loops=1)
9            Output: (data)::text, random()
10           Buffers: shared hit=20342 read=19732
11  Execution Time: 1291.764 ms
```



How to fix detoasting

Case 1: value is detoasted too early.

- Use subqueries with `OFFSET/LIMIT` as a boundary to limit evaluation push down.



Subquery boundary

```
SELECT LENGTH(data::text) FROM (SELECT data FROM reports ORDER BY random() LIMIT 100);
```



Subquery boundary

```
SELECT LENGTH(data::text) FROM (SELECT data FROM reports ORDER BY random() LIMIT 100);
```

```
1 Subquery Scan on unnamed_subquery (actual time=2.232..14.562 rows=100 loops=1)
2   Output: (unnamed_subquery.data)::text
3   Buffers: shared hit=442 read=32
4   -> Limit (actual time=2.076..2.089 rows=100 loops=1)
5     Output: reports.data, (random())
6     -> Sort (actual time=2.076..2.081 rows=100 loops=1)
7       Output: reports.data, (random())
8       Sort Key: (random())
9       Sort Method: top-N heapsort Memory: 37kB
10      -> Seq Scan on public.reports (actual time=0.008..0.990 rows=10000 loops=1)
11        Output: reports.data, random()
12        Buffers: shared hit=74
13 Execution Time: 14.582 ms
```



How to fix detoasting 2

Case 2: value is detoasted multiple times

- Force early detoasting by a dummy operation.



Add dummy operation

```
SELECT report_id, rule_id
FROM reports
  JOIN rules USING (ruleset_id)
WHERE (data->metric_field)::real > max_value
```

to

```
SELECT report_id, rule_id
FROM (SELECT report_id, ruleset_id, data || '{}' data FROM reports OFFSET 0)
  JOIN rules USING (ruleset_id)
WHERE (data->metric_field)::real > max_value;
```



Dummy operation explain

```
1 Hash Join (actual time=0.749..333.923 rows=1248 loops=1)
2   Hash Cond: (reports.ruleset_id = rules.ruleset_id)
3   Join Filter: (((((reports.data || '{}')::jsonb)) -> rules.metric_field))::real > rules.max_va
4   Rows Removed by Join Filter: 98752
5   Buffers:  shared hit=30216 read=9866
6   -> Seq Scan on reports (actual time=0.045..273.442 rows=10000 loops=1)
7       Buffers:  shared hit=30214 read=9860
8   -> Hash (actual time=0.213..0.214 rows=1000 loops=1)
9       Buckets: 1024 Batches: 1 Memory Usage: 63kB
10      Buffers: shared hit=2 read=6
11      -> Seq Scan on rules (actual time=0.004..0.099 rows=1000 loops=1)
12          Buffers: shared hit=2 read=6
13 Execution Time: 334.006 ms
```



Handling TOAST in queries

- Be concious of whether large values are involved in a query plan.
- The planner is completely oblivious about detoasting.
- Think if you need to be eager or lazy.
- Use tricks to force the planners hand.



Chapter 2: I (don't) see dead tuples



Schema time

- We are building a task queue

```
CREATE TYPE task_status AS ENUM ('Todo', 'Done', 'Failed');
```

```
CREATE TABLE tasks (  
  id bigserial primary key,  
  job_id int not null default random(1, 10),  
  status task_status not null,  
  added timestamptz not null default now(),  
  done timestamptz  
);
```

```
CREATE INDEX ON tasks (added) WHERE status = 'Todo';
```



Add some tasks

```
INSERT INTO tasks (status)
  SELECT 'Todo' FROM generate_series(1,100) i;
```



The workload

queue-insert.sql

```
INSERT INTO tasks (status) VALUES ('Todo');
```

queue-complete.sql

```
UPDATE tasks SET status = 'Done', done = NOW()  
WHERE id = (SELECT id FROM tasks  
            WHERE status = 'Todo' ORDER BY added  
            FOR UPDATE SKIP LOCKED LIMIT 1);
```



Running the workload

```
pgbench -n -f queue-insert.sql -f queue-complete.sql \  
  --rate=2000 -j8 -c8 \  
  -P 10 -T 600
```



Running the workload

```
pgbench -n -f queue-insert.sql -f queue-complete.sql \  
  --rate=2000 -j8 -c8 \  
  -P 10 -T 600
```

```
progress: 10.0 s, 1996.6 tps, lat 4.251 ms stddev 2.994, 0 failed, lag 1.945 ms  
progress: 20.0 s, 1991.9 tps, lat 3.897 ms stddev 2.686, 0 failed, lag 1.673 ms  
progress: 30.0 s, 1969.4 tps, lat 6.368 ms stddev 13.453, 0 failed, lag 4.003 ms  
progress: 40.0 s, 2006.0 tps, lat 4.353 ms stddev 3.135, 0 failed, lag 2.026 ms  
progress: 50.0 s, 2008.1 tps, lat 4.225 ms stddev 2.830, 0 failed, lag 1.905 ms
```



Meanwhile in another part of town

A business analyst using DBeaver:

```
BEGIN ISOLATION LEVEL REPEATABLE READ;  
SELECT COUNT(*) FROM tasks WHERE status = 'Todo';
```



Meanwhile in another part of town

A business analyst using DBeaver:

```
BEGIN ISOLATION LEVEL REPEATABLE READ;  
SELECT COUNT(*) FROM tasks WHERE status = 'Todo';
```

“Let’s go get a coffee to think about that number...”



Back at benchmark central

```
progress: 90.0 s, 1985.0 tps, lat 17.509 ms stddev 17.266, 0 failed, lag 14.128 ms
progress: 100.0 s, 1654.7 tps, lat 1188.524 ms stddev 343.977, 0 failed, lag 1183.696 ms
progress: 110.0 s, 1552.0 tps, lat 2753.375 ms stddev 686.912, 0 failed, lag 2748.222 ms
progress: 120.0 s, 1353.5 tps, lat 5476.902 ms stddev 987.954, 0 failed, lag 5470.992 ms
progress: 130.0 s, 1280.2 tps, lat 8885.448 ms stddev 1064.684, 0 failed, lag 8879.201 ms
progress: 140.0 s, 1177.2 tps, lat 12687.523 ms stddev 1277.164, 0 failed, lag 12680.729 ms
progress: 150.0 s, 1103.0 tps, lat 17038.791 ms stddev 1365.447, 0 failed, lag 17031.536 ms
progress: 160.0 s, 1047.2 tps, lat 21655.815 ms stddev 1455.754, 0 failed, lag 21648.179 ms
progress: 170.0 s, 985.5 tps, lat 26621.604 ms stddev 1573.951, 0 failed, lag 26613.488 ms
progress: 180.0 s, 923.9 tps, lat 31668.206 ms stddev 1587.211, 0 failed, lag 31659.547 ms
progress: 190.0 s, 918.5 tps, lat 37157.963 ms stddev 1645.525, 0 failed, lag 37149.256 ms
progress: 200.0 s, 877.3 tps, lat 42607.753 ms stddev 1721.361, 0 failed, lag 42598.632 ms
progress: 210.0 s, 853.5 tps, lat 48289.559 ms stddev 1720.792, 0 failed, lag 48280.190 ms
progress: 220.0 s, 788.5 tps, lat 54187.022 ms stddev 1852.551, 0 failed, lag 54176.878 ms
progress: 230.0 s, 751.6 tps, lat 60477.265 ms stddev 1899.692, 0 failed, lag 60466.617 ms
```



Incident resolution

- “Our CPUs are on fire, what is going on?”
- “pg_stat_statements says that the queue completion query is 100x slower.”
- “I know, let’s get an explain plan!”



The explain plan

```
1 Update on tasks (actual time=25.273..25.274 rows=0 loops=1)
2   Buffers: shared hit=95467 dirtied=1 written=1
3   I/O Timings: shared write=0.028
4   InitPlan 1 (returns $1)
5     -> Limit (actual time=25.204..25.205 rows=1 loops=1)
6         Buffers: shared hit=95454
7         -> LockRows (actual time=25.204..25.204 rows=1 loops=1)
8             Buffers: shared hit=95454
9             -> Index Scan using tasks_added_idx on tasks tasks_1 (actual time
10                 =25.143..25.148 rows=10 loops=1)
11                 Filter: (status = 'Todo'::task_status)
12                 Buffers: shared hit=95428
13             -> Index Scan using tasks_pkey on tasks (actual time=25.214..25.215 rows=1 loops=1)
14                 Index Cond: (id = $1)
15                 Buffers: shared hit=95458
16 Execution Time: 25.292 ms
```



What's going on

- The open transaction is preventing autovacuum from cleaning up completed jobs.
- Index fills up with old row versions that have actually already been updated.
- Due to the open transaction we can't cache the dead status in the index.
 - See "Killed Index Tuples" blogpost by Laurenz
- Every time we look for a task, we have to scan over the index entries for already completed tasks.
 - For each one go and look at the row in the table to see that it has been updated.



What's going on

- The open transaction is preventing autovacuum from cleaning up completed jobs.
- Index fills up with old row versions that have actually already been updated.
- Due to the open transaction we can't cache the dead status in the index.
 - See “Killed Index Tuples” blogpost by Laurenz
- Every time we look for a task, we have to scan over the index entries for already completed tasks.
 - For each one go and look at the row in the table to see that it has been updated.
- None of this is visible in the explain numbers.



Fixing it

- Avoid mixing long queries/transactions and update heavy workloads.
- Use `statement_timeout`, `idle_in_transaction_session_timeout` to have a backstop against accidents.
- PostgreSQL 17 also has `transaction_timeout`.



After terminating the naughty connection

```
1 Update on tasks (actual time=0.308..0.308 rows=0 loops=1)
2   Buffers: shared hit=516
3   InitPlan 1 (returns $1)
4     -> Limit (actual time=0.290..0.291 rows=1 loops=1)
5         Buffers: shared hit=506
6         -> LockRows (actual time=0.290..0.290 rows=1 loops=1)
7             Buffers: shared hit=506
8             -> Index Scan using tasks_added_idx on tasks tasks_1 (actual time
                =0.284..0.285 rows=2 loops=1)
9                 Filter: (status = 'Todo'::task_status)
10                 Buffers: shared hit=504
11             -> Index Scan using tasks_pkey on tasks (actual time=0.294..0.295 rows=1 loops=1)
12                 Index Cond: (id = $1)
13                 Buffers: shared hit=510
14 Execution Time: 0.328 ms
```



The missing information

- How many rows were scanned but found not visible
- How many killed index tuples were skipped over
- This also affects sequential scans, it's just not as easy to see



The invisible visibility map

We need a larger table for this:

```
CREATE TABLE bigger AS SELECT i, repeat(' ', 100)
  FROM generate_series(1,2) j, generate_series(1,3000000) i;
```

```
CREATE INDEX ON bigger(i);
```

```
VACUUM ANALYZE bigger;
```



Holy buffer hit count Batman

```
EXPLAIN (BUFFERS, ANALYZE, COSTS OFF) SELECT i FROM bigger;
```



Holy buffer hit count Batman

```
EXPLAIN (BUFFERS, ANALYZE, COSTS OFF) SELECT i FROM bigger;
```

```
1  Index Only Scan using bigger_i_idx on bigger (actual time=0.016..799.596 rows=6000000
    loops=1)
2   Heap Fetches: 0
3   Buffers: shared hit=6014781
4   Planning Time: 0.044 ms
5   Execution Time: 958.258 ms
```



What's going on

- Index only scan looks at visibility map to check if we can skip the heap fetch
- This happens for each row
- It caches the location of the last looked at VM page and skips buffer lookup if next one is the same.
- Example was constructed so this never works out.
- Happens in the real world too with random access to tables >256MB
 - See “Unexpected downsides of UUID keys in PostgreSQL” blogpost



Fixing it

- There are caches everywhere.
- Data locality matters.
- Use CLUSTER, fillfactor and other tricks to keep data sorted by access patterns.



Fixing it

- There are caches everywhere.
- Data locality matters.
- Use `CLUSTER`, `fillfactor` and other tricks to keep data sorted by access patterns.

```
CLUSTER bigger USING bigger_i_idx;
```



Results

```
1  Index Only Scan using bigger_i_idx on bigger (actual time=0.017..342.865 rows=6000000
    loops=1)
2    Heap Fetches: 0
3    Buffers: shared hit= 14785
4  Planning Time: 0.042 ms
5  Execution Time: 500.547 ms
```

~2x performance difference just from avoiding visibility map buffer lookups.



Chapter 3: Hello? Is this thing on?!



Trip down the memory lane

Taking our tasks table from before:

```
CREATE TABLE tasks (  
  id bigserial primary key,  
  job_id int not null default floor(random()*10 + 1)::int,  
  status task_status not null,  
  added timestamptz not null default now(),  
  done timestamptz  
);
```



New goal

We have a query:

```
SELECT id FROM tasks
WHERE job_id = 3 AND added < '1969-07-20 15:17:40-05'
ORDER BY id;
```

Lets try a couple of indexes to make it fast



Tale of two indexes

```
CREATE INDEX j_i_a ON tasks (job_id, id, added);
```

```
1  Index Only Scan using j_i_a on tasks (actual time=1.207..1.207 rows=0 loops=1)
2  Index Cond: ((job_id = 3) AND (added < '1969-07-20 23:17:40+03'::timestamp with time
   zone))
3  Heap Fetches: 0
4  Buffers: shared hit=300
```

```
CREATE INDEX j_a_i ON tasks (job_id, added, id);
```

```
1  Sort (actual time=0.020..0.021 rows=0 loops=1)
2  -> Index Only Scan using j_a_i on tasks (actual time=0.013..0.013 rows=0 loops=1)
3  Index Cond: ((job_id = 3) AND (added < '1969-07-20 23:17:40+03'::timestamp with
   time zone))
4  Heap Fetches: 0
5  Buffers: shared hit=3
```



The answer

- Index range scans ($col < const$) can only be used if **all** preceding index columns have equality on them.
- With $(job_id, id, added)$ we cannot use `added` for scanning as it's unordered:

<code>job_id</code>	[3	[3	[3	[3	[3
<code>id</code>	1	2	3	4	7
<code>added</code>	13:35]	17:49]	11:05]	19:12]	09:12]

- But we can scan all for a single `job_id` and use the `added` for filtering.
- The fact that Index Cond is only used for filtering is not visible anywhere in the explain plan.
 - Neither is the amount of index tuples scanned and discarded.
- Rows Removed by Filter: only includes filters done on table values.



Fin



What did we learn today

- Explain still doesn't explain everything
- In particular, hidden detoasting and bloat scanning might make things slow.
- EXPLAIN is always improving, hopefully we will soon have more visibility.



Thank you



Q & A



Bonus content



More things to improve

- How many of updates were HOT.
- How many pages were pruned while scanning
- How many index probes were done during planning
- Hint bit updates log WAL, but this doesn't show up with EXPLAIN (WAL)
- SLRU accesses are completely hidden.
- Getting explain plans from within functions is quite tricky.
- When are extended statistics consulted.
- How much time was spent waiting on locks.

