Troubleshooting performance issues in PostgreSQL

Camille Baldock, Salesforce

LISA2017

Hello

Administration

- **13:30 Welcome and introduction**
  - Turn all devices into “silent” mode please!
- **15:00 Afternoon break**
- **15.30 End of break**
- **17:00 End of class**

Follow along

- **https://camillebaldock.com/postgres**
  - Installation instructions
  - 🔷 1
  - psql

Troubleshooting performance issues in PostgreSQL

- Query execution
- Query plans
- At scale: high traffic, growing data
- Other performance factors (PostgreSQL version, hardware) and how to benchmark them
- What to monitor

Query execution

- 📖 how PostgreSQL executes queries and handles crash recovery
- 🖥️ how to modify PostgreSQL settings to optimize query execution and data persistence

Query plans

- 📖 how the query optimizer works
- 🖥️ how to modify queries and PostgreSQL settings to get queries running as efficiently as possible

Other performance factors

- 📖 impact of PostgreSQL versions/hardware on performance
- 🖥️ how to measure and benchmark those changes

At scale: high traffic, growing data

- 🖥️ leveraging replication
- 🖥️ managing connections
- 🖥️ managing increased write rates
- 🖥️ large tables

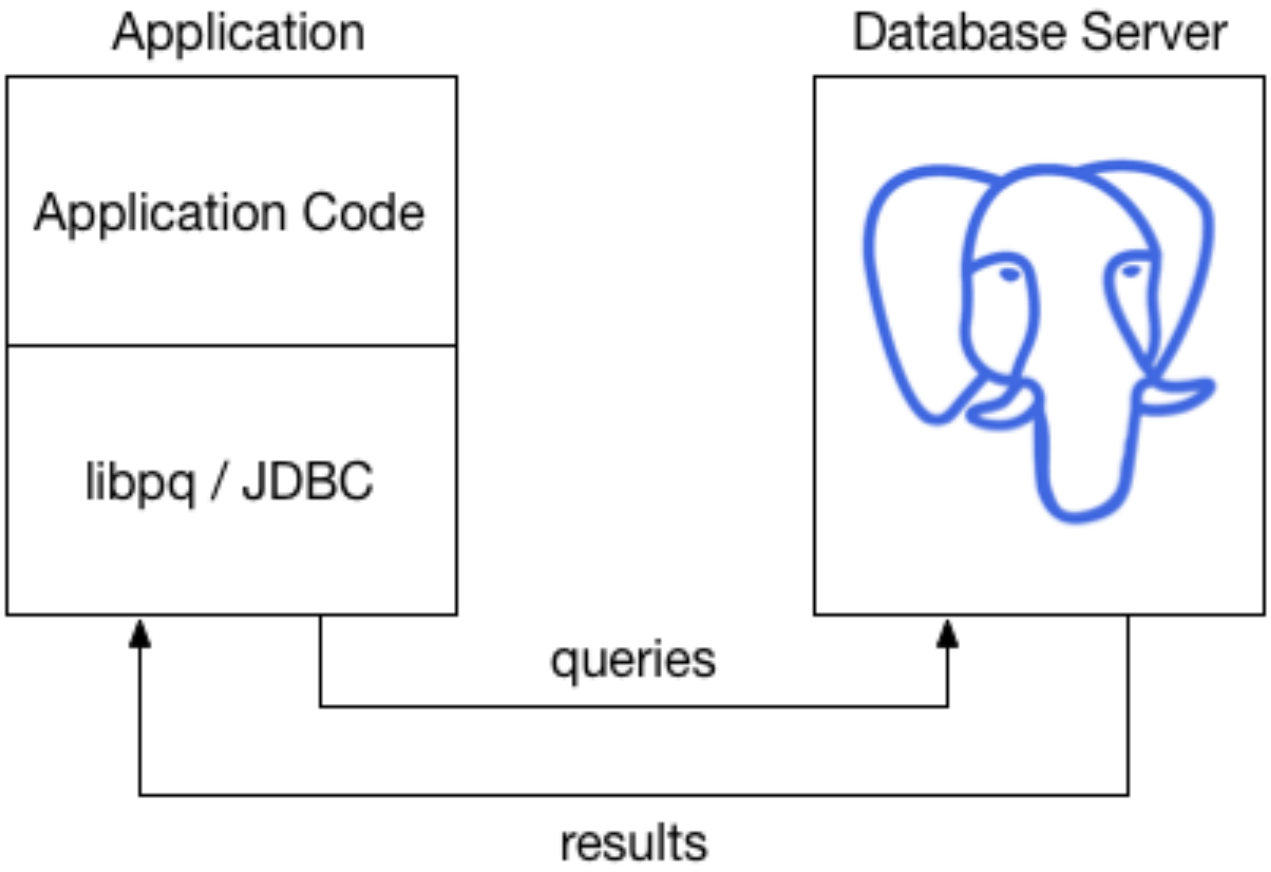
What to monitor

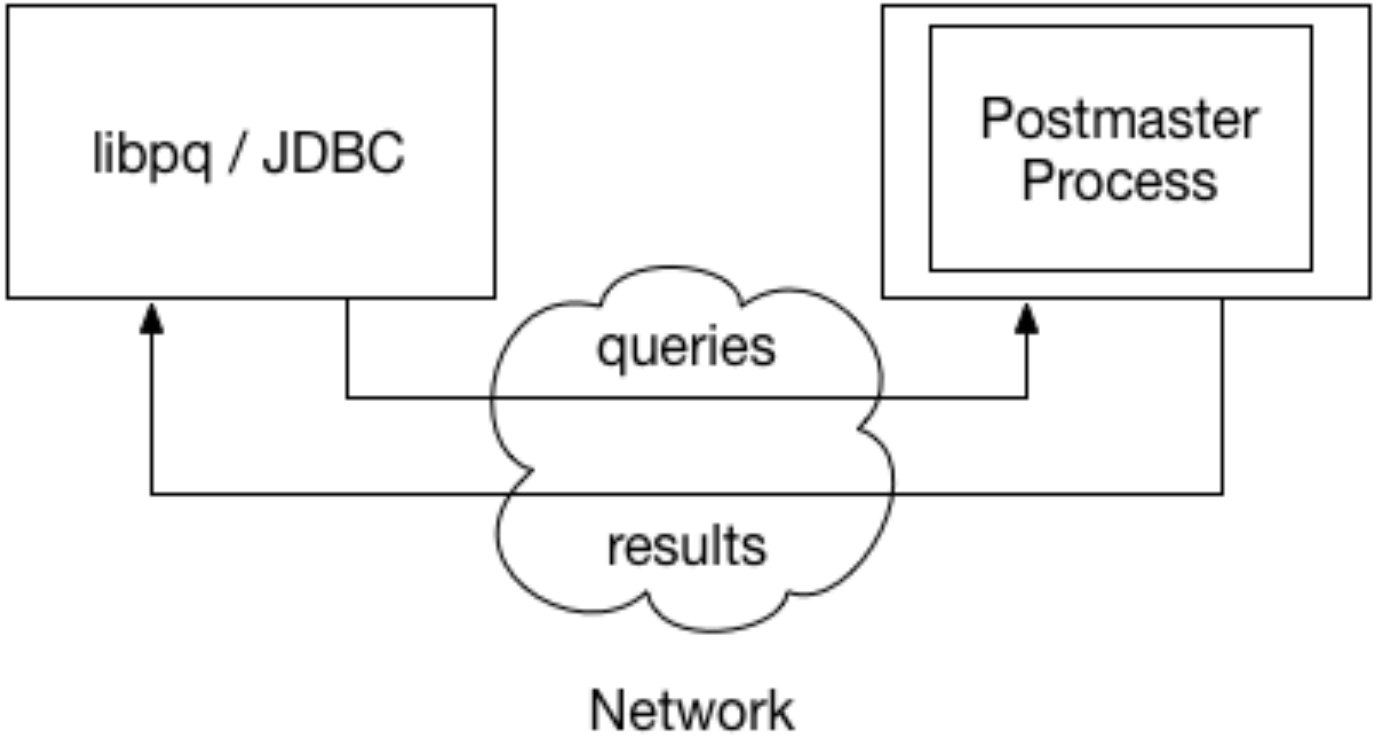
- 🖥️ finding slow queries through pg_stat_statements
- 🖥️ finding blocked queries, locks
- 🖥️ monitoring vacuuming and cache hit rates

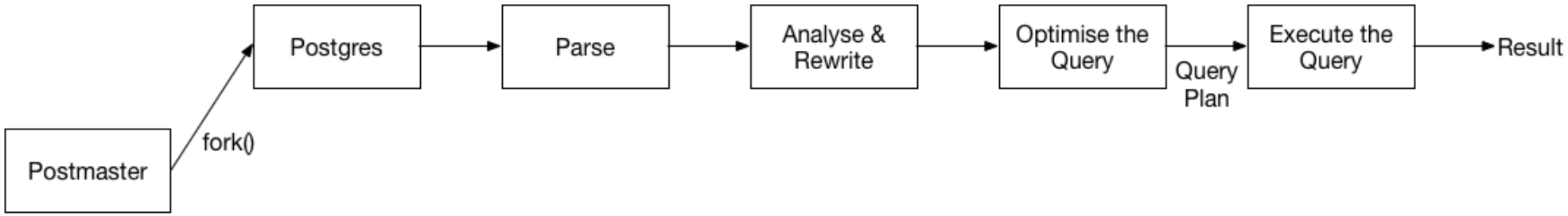
Administration

- **13:30 Welcome and introduction**
- 13.45 Query execution
- 14.10 Query planning
- **15:00 Afternoon break**
- **15.30 End of break**
- 15:45 At scale
- 16:10 Other performance factors
- 16:30 Monitoring
- 16:50 Summary and final questions
- **17:00 End of class**

```
SELECT name FROM conferences WHERE
description = 'Scaling the future';
   name
----------
 LISA2017
(1 row)
```

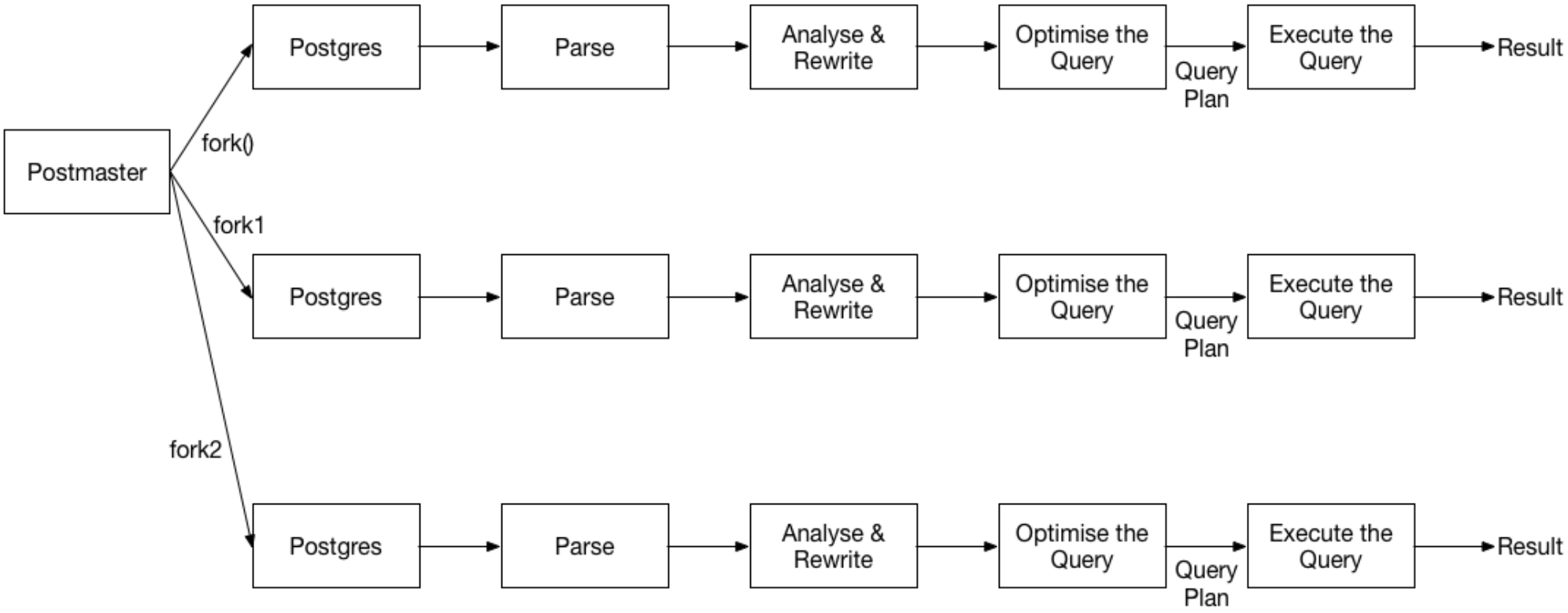

Application
Database Server
Application Code
libpq / JDBC
queries
results


libpq / JDBC
Postmaster Process
queries
results
Network


Postmaster
fork()
Postgres
Parse
Analyse & Rewrite
Optimise the Query
Query Plan
Execute the Query
Result

```
=# SELECT name FROM
conferences WHERE description
= 'Scaling the future';
DEBUG:
StartTransactionCommand
DEBUG:  StartTransaction
[...]
LOG:  PARSER STATISTICS
[...]
LOG:  statement: SELECT name
FROM conferences WHERE
description = 'Scaling the
future';
LOG:  PARSE ANALYSIS
STATISTICS
[...]
LOG:  parse tree:
[...]
```

```
LOG:  REWRITER STATISTICS
[...]
LOG:  rewritten parse tree:
[…]
LOG:  PLANNER STATISTICS
[...]
LOG:  plan:
[...]
DEBUG:  PortalRun
LOG:  EXECUTOR STATISTICS
[...]
    name
----------
 LISA2017
(1 row)
```

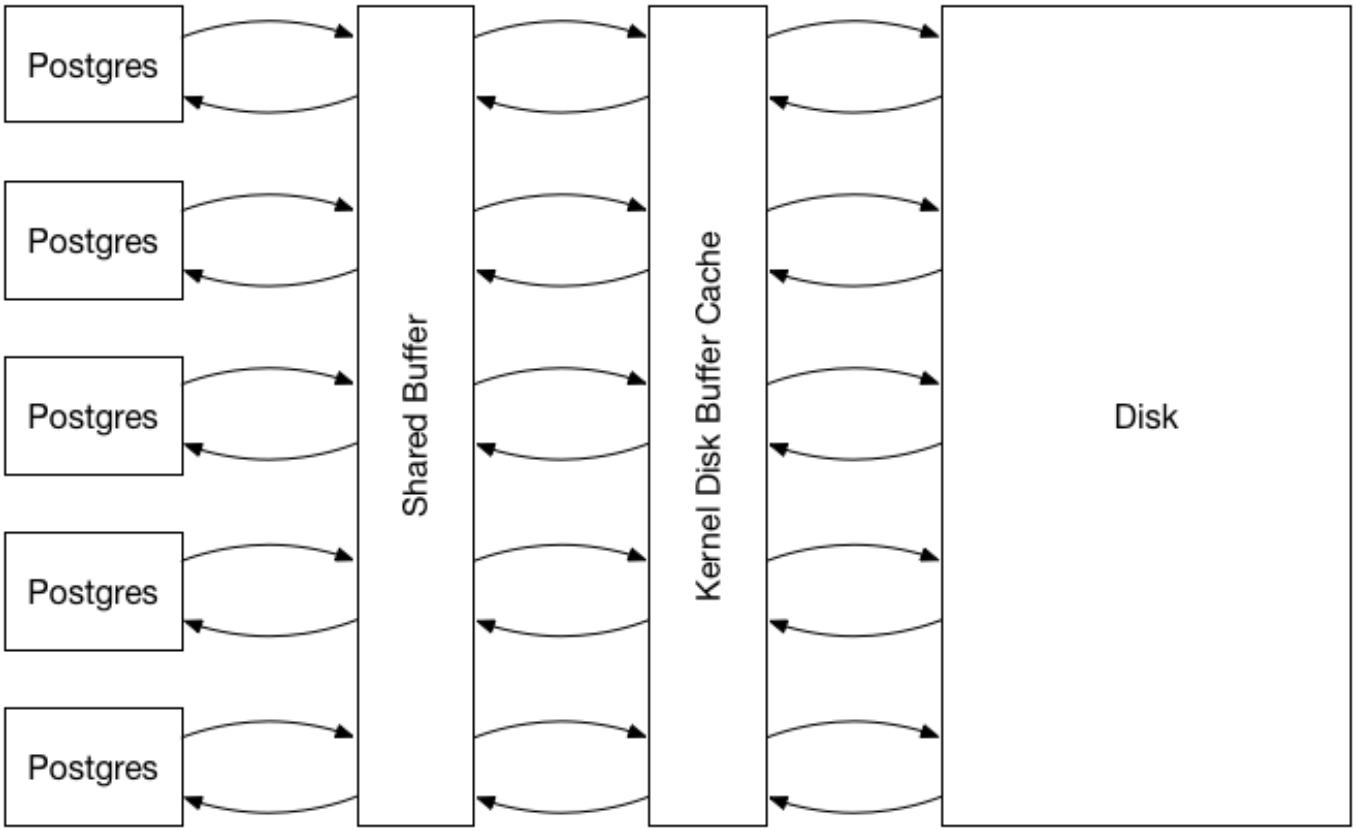

Postmaster
fork()
fork1
fork2
Postgres
Parse
Analyse & Rewrite
Optimise the Query
Query Plan
Execute the Query
Result
Postgres
Parse
Analyse & Rewrite
Optimise the Query
Query Plan
Execute the Query
Result
Postgres
Parse
Analyse & Rewrite
Optimise the Query
Query Plan
Execute the Query
Result

Troubleshooting performance issues in PostgreSQL

- **Query execution**
- Query plans
- At scale: high traffic, growing data
- Other performance factors (PostgreSQL version, hardware) and how to benchmark them
- What to monitor

Query execution

- 📖 How PostgreSQL saves your data
- 📖 How PostgreSQL deals with crash recovery
- 🖥️ What parameters can be changed to make execution faster


Postgres
Postgres
Postgres
Postgres
Postgres
Shared Buffer
Kernel Disk Buffer Cache
Disk

1

```
CREATE TABLE demo_buffers(value int);

SELECT
pg_relation_filepath('demo_buffers');
pg_relation_filepath
----------------------
base/16385/16504
```

2

```
SELECT relname, oid, relfilenode
FROM pg_class WHERE relname='demo_buffers';
relname |  oid  | relfilenode
————+———————+—————————
foo     | 16504 |       16504
```

🔷 3 \d pg_catalog.pg_class;

relname	relpages
relnamespace	reltuples
reltype	oid
reloftype	relhasindex
relowner	…
relam	
relfilenode	
reltablespace	

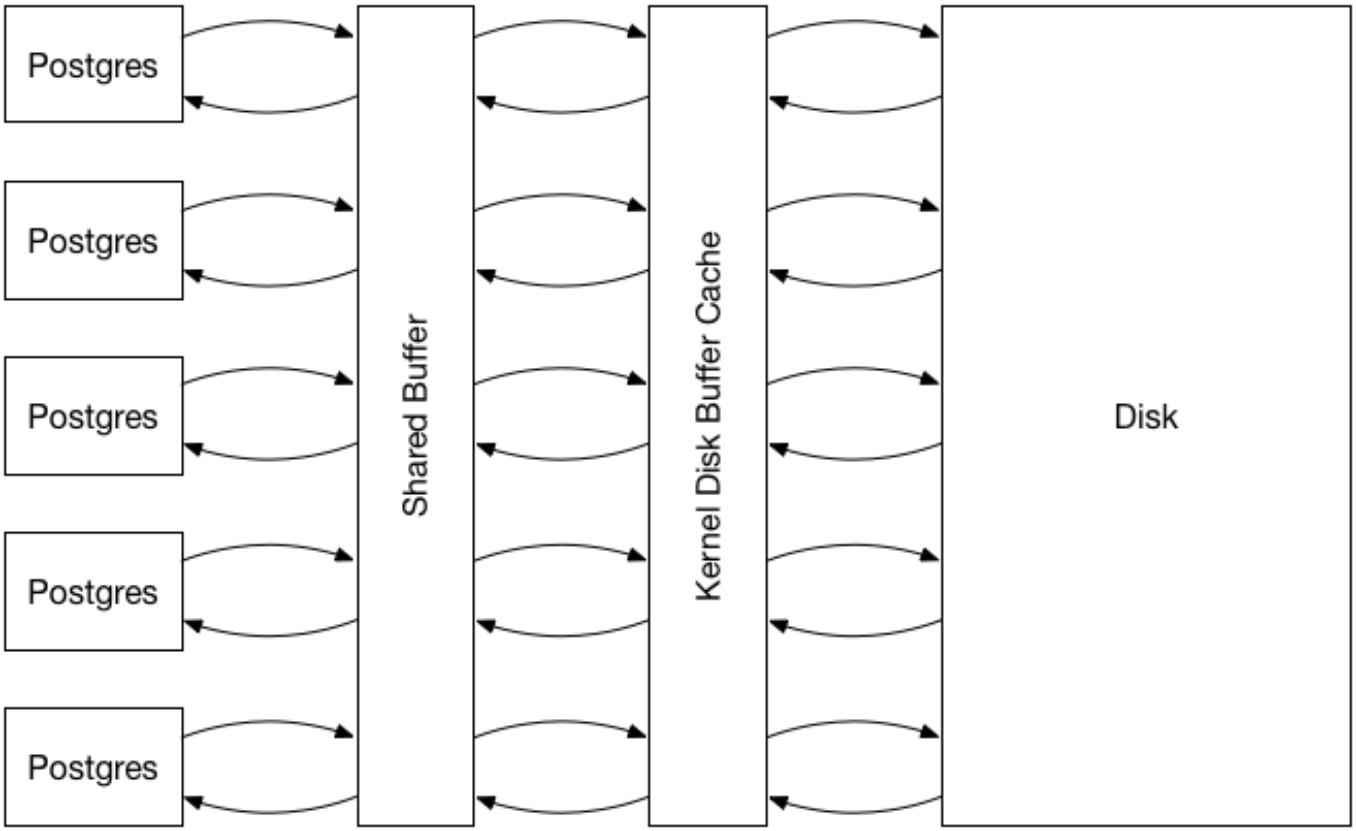
🔷 4

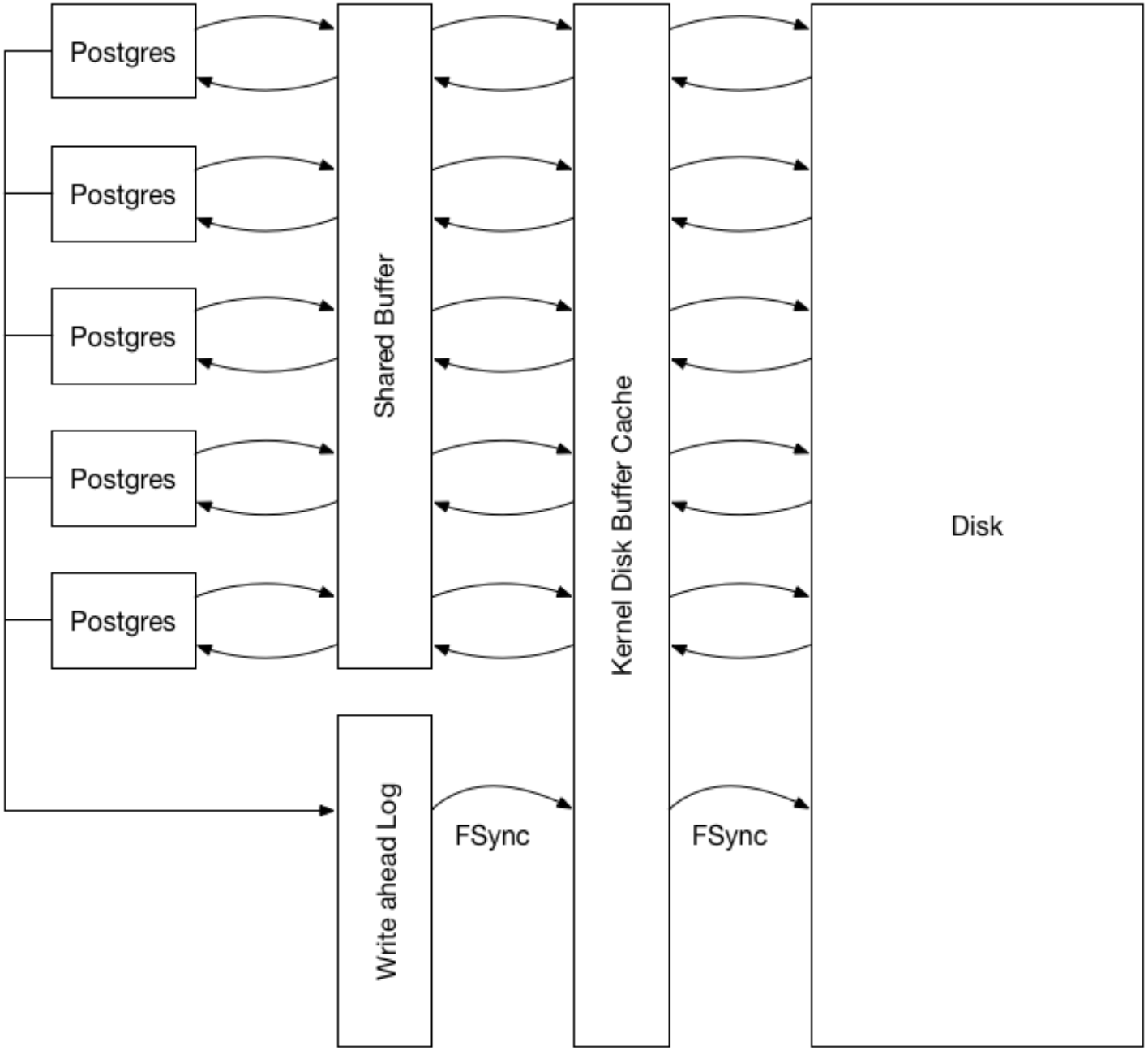
```
CREATE EXTENSION pg_buffercache;

INSERT INTO demo_buffers VALUES (1);

SELECT reldatabase, relfilenode, isdirty,
usagecount from pg_buffercache WHERE
relfilenode=16504;
```

```
reldatabase | relfilenode | isdirty | usagecount
------------+-------------+---------+------------
    16385   |   16504     |   t     |       1
```

🔷 5

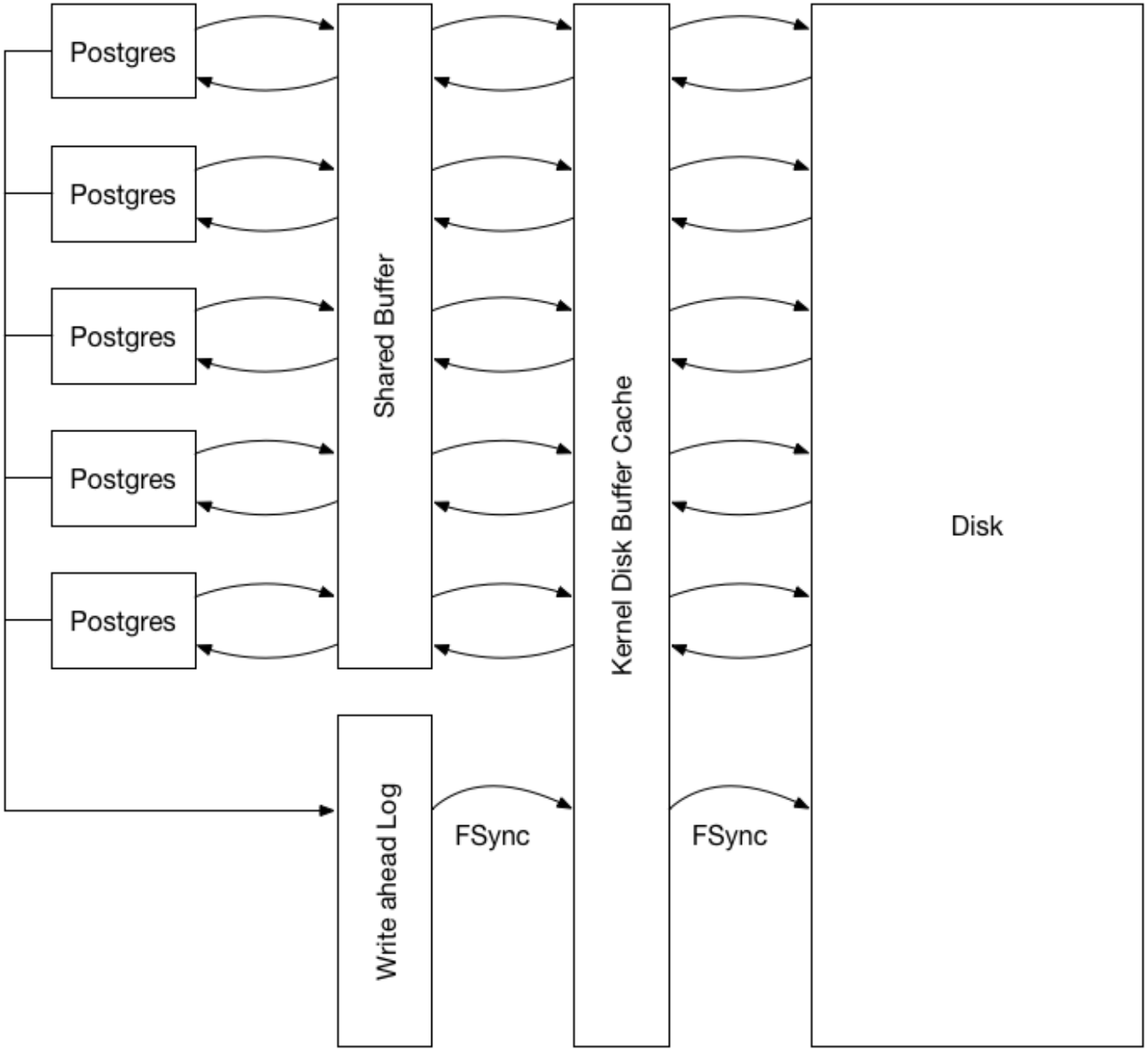
```
CHECKPOINT;

SELECT reldatabase,relfilenode,
isdirty, usagecount FROM pg_buffercache
WHERE relfilenode=16504;

reldatabase | relfilenode | isdirty | usagecount
------------+-------------+---------+-----------
   16385    |    16504    |    f    |     1
```


Postgres
Postgres
Postgres
Postgres
Postgres
Shared Buffer
Kernel Disk Buffer Cache
Disk


Postgres
Postgres
Postgres
Postgres
Postgres
Shared Buffer
Write ahead Log
FSync
Kernel Disk Buffer Cache
FSync
Disk

checkpoint settings

- Checkpoints reduce the amount of time to recover from a crash.
- Checkpoint writes are expensive and the first time a page is looked at after a check point, it has to be written in its entirety to the WAL.
- Settings for checkpoint frequency
  - **`min_wal_size`**/**`max_wal_size`** (or **`checkpoint_segments`** in versions before 9.6)
  - **`checkpoint_timeout`** (when that time has passed, do a checkpoint)


Postgres
Postgres
Postgres
Postgres
Postgres
Shared Buffer
Write ahead Log
FSync
Kernel Disk Buffer Cache
FSync
Disk

shared_buffers

- Whenever a block is used in shared memory, it increments a clock-sweep algorithm that ranges from 1-5, 5 being extremely high use data blocks.
  - High usage blocks are likely to be kept in shared_buffers
  - Low usage blocks will get moved out if space for higher usage ones is needed.

shared_buffers (2)

- 1GB or more of RAM, a reasonable starting value for shared_buffers is 25% of the memory in your system
- Unlikely to find over 40% to work better than small amount (due to how Postgres uses OS cache)

shared_buffers (3)

- **`shared_buffers`** need to be big enough to accommodate all quite popular pages (usagecount > 3)
  - **`SELECT pg_size_pretty(count(*) * 8192) as ideal_shared_buffers FROM pg_buffercache b WHERE usagecount >= 3;`** is a good place to start

Query execution

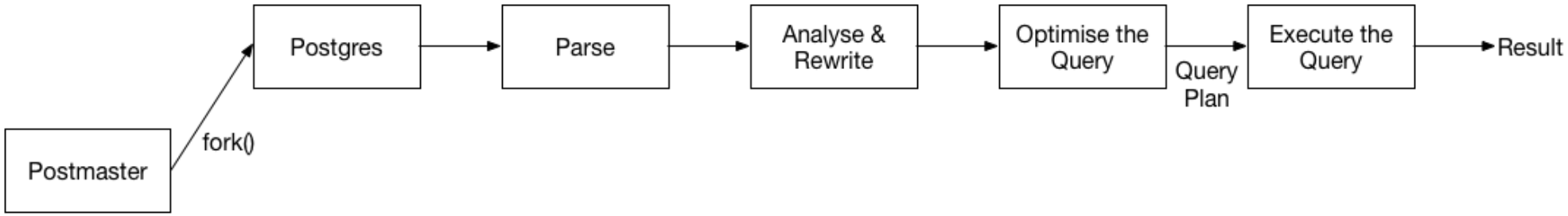
- 📖 shared buffers, write ahead log, checkpoints
- 🖥️ checkpoint settings, **`shared_buffers`** and using pg_buffercache for estimating size needed

Troubleshooting performance issues in PostgreSQL

- Query execution
- **Query plans**
- At scale: high traffic, growing data
- Other performance factors (PostgreSQL version, hardware) and how to benchmark them
- What to monitor

Query plans

- 📖 Understand how the PostgreSQL query optimizer works
  - scan methods
  - join methods
  - how cardinality and indexes affect them
- 🖥️ how to get and analyze query plans, query planner settings, memory settings, what to index


Postmaster
fork()
Postgres
Parse
Analyse & Rewrite
Optimise the Query
Query Plan
Execute the Query
Result

6

```
CREATE TABLE conferences (
  name text,
  description text,
  conference_id int,
  total_attendees int DEFAULT 0,
  expected_attendees int DEFAULT 0,
  start_date date);

INSERT INTO conferences
(name, description, conference_id, total_attendees,
expected_attendees)
SELECT 'name ' || s,
       'description ' || s,
       s,
       (random() * 500 + 1)::int,
       (random() * 500 + 1)::int
FROM generate_series(1,20000) s;
```

🔷 7

```
SELECT * FROM conferences
WHERE conference_id = 42;
```

🔷 8

```
EXPLAIN SELECT * FROM conferences WHERE
conference_id = 42;

QUERY PLAN
-----------------------------------------
Seq Scan on conferences
(cost=0.00..427.00 rows=1 width=39)
  Filter: (conference_id = 42)
```

```
EXPLAIN SELECT * FROM conferences WHERE
conference_id = 42;

QUERY PLAN
-----------------------------------------
Seq Scan on conferences
(cost=0.00..427.00 rows=1 width=39)
  Filter: (conference_id = 42)
```

```
EXPLAIN SELECT * FROM conferences WHERE
conference_id = 42;

QUERY PLAN
-----------------------------------------
Seq Scan on conferences
(cost=0.00..427.00 rows=1 width=39)
  Filter: (conference_id = 42)
```

9

```
select relpages, reltuples from
pg_catalog.pg_class where relname =
'conferences';
```

```
 relpages | reltuples
----------+-----------
      177 |     20000
```

```
select histogram_bounds from pg_stats
where tablename = 'conferences';
```

```
...
...
{1,200,400,600,800,1000,1200,1400,1600,1800,2000,2200,24
00,2600,2800,3000,3200,3400,3600,3800,4000,4200,4400,460
0,4800,5000,5200,5400,5600,5800,6000,6200,6400,6600,6800
,7000,7200,7400,7600,7800,8000,8200,8400,8600,8800,9000,
9200,9400,9600,9800,10000,10200,10400,10600,10800,11000,
11200,11400,11600,11800,12000,12200,12400,12600,12800,13
000,13200,13400,13600,13800,14000,14200,14400,14600,1480
0,15000,15200,15400,15600,15800,16000,16200,16400,16600,
16800,17000,17200,17400,17600,17800,18000,18200,18400,18
600,18800,19000,19200,19400,19600,19800,20000}
{1,5,10,15,20,25,30,34,39,44,50,54,60,64,70,74,78,83,87,
92,97,102,107,113,117,122,128,133,137,141,145,150,155,16
1,167,173,178,184,191,195,200,204,209,214,219,223,228,23
3,237,243,247,252,257,262,266,272,277,282,287,291,296,30
1,305,313,318,324,328,333,338,342,347,351,356,361,365,37
1,376,384,389,393,399,403,408,413,418,422,429,434,438,44
4,450,455,461,467,472,477,482,486,491,496,501}
...
```

10

```
EXPLAIN ANALYZE SELECT * FROM
conferences WHERE conference_id = 42;

QUERY PLAN
-----------------------------------------
Seq Scan on conferences
(cost=0.00..427.00 rows=1 width=39)
(actual time=0.031..7.886 rows=1
loops=1)
  Filter: (conference_id = 42)
  Rows Removed by Filter: 19999
Planning time: 0.073 ms
Execution time: 7.923 ms
```

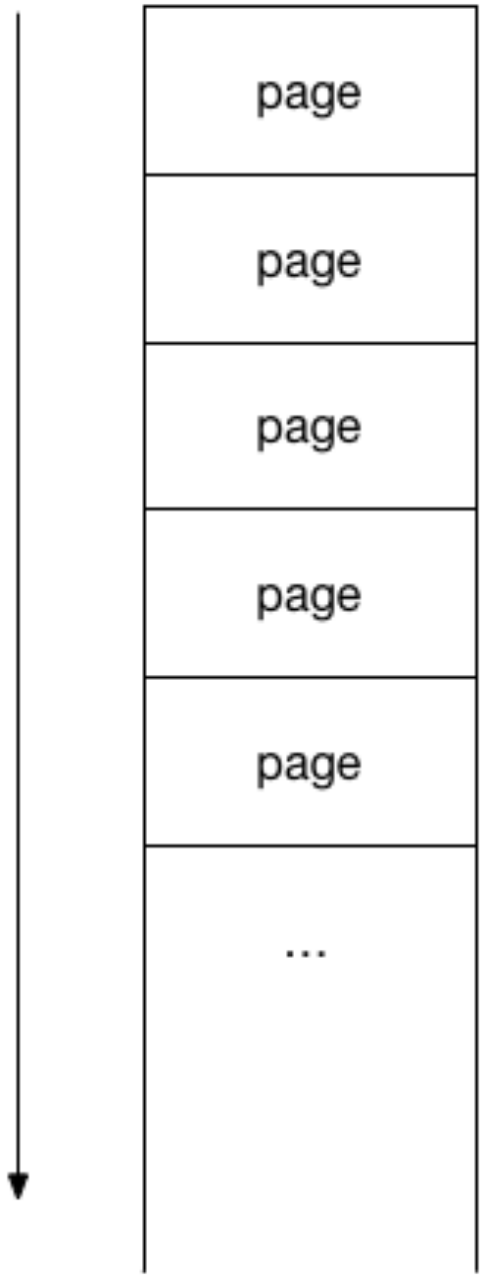
Scan methods

- Sequential scan
- Index scan
- Bitmap heap and index scan

11

```
EXPLAIN ANALYZE SELECT * FROM
conferences WHERE conference_id = 42;

QUERY PLAN
-----------------------------------------
Seq Scan on conferences
(cost=0.00..427.00 rows=1 width=39)
(actual time=0.031..7.886 rows=1
loops=1)
  Filter: (conference_id = 42)
  Rows Removed by Filter: 19999
Planning time: 0.073 ms
Execution time: 7.923 ms
```

Sequential scan



🔷 12

```
CREATE INDEX ON
conferences(conference_id);
```

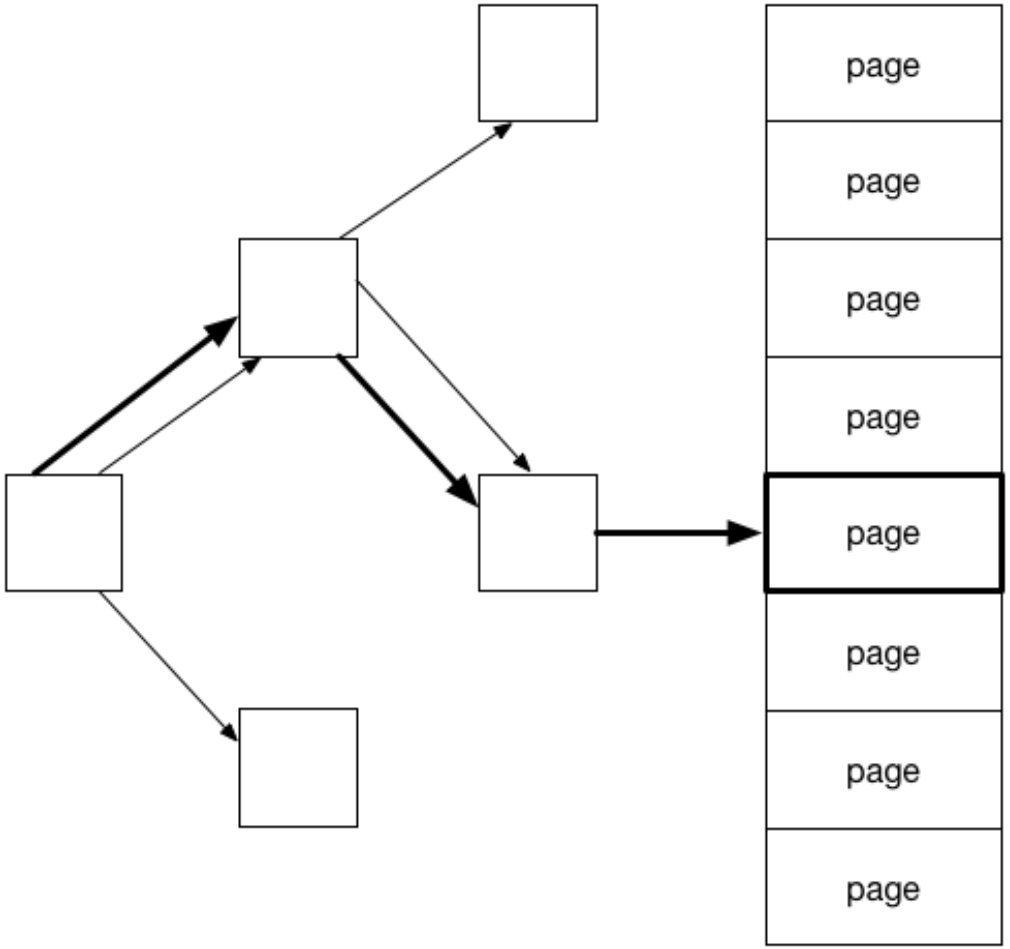
🔷 13

```
CREATE INDEX CONCURRENTLY ON
conferences(conference_id);
```

11

```
EXPLAIN ANALYZE SELECT * FROM
conferences WHERE conference_id = 42;

QUERY PLAN
------------------------------------
Index Scan using
conferences_conference_id_idx on
conferences
(cost=0.29..8.30 rows=1 width=39)
(actual time=0.029..0.030 rows=1
loops=1)
  Index Cond: (conference_id = 42)
Planning time: 0.391 ms
Execution time: 0.063 ms
(4 rows)
```

B-tree index scan



🔷 14

```
\timing
Timing is on.

EXPLAIN ANALYZE SELECT count(*) FROM
conferences WHERE total_attendees >
100;
Time: 9.996 ms

SELECT count(*) FROM conferences WHERE
total_attendees > 100;
Time: 7.625 ms
```

7

```
SELECT * FROM conferences
WHERE conference_id = 42;
```

	First run	Second run	Third run
Sequential scan	29.914 ms	7.195 ms	6.506 ms
Index scan	9.835 ms	0.594 ms	0.563 ms

🔷 15

```
CREATE INDEX ON
conferences(total_attendees);

SELECT * FROM conferences
WHERE total_attendees > 100;
```

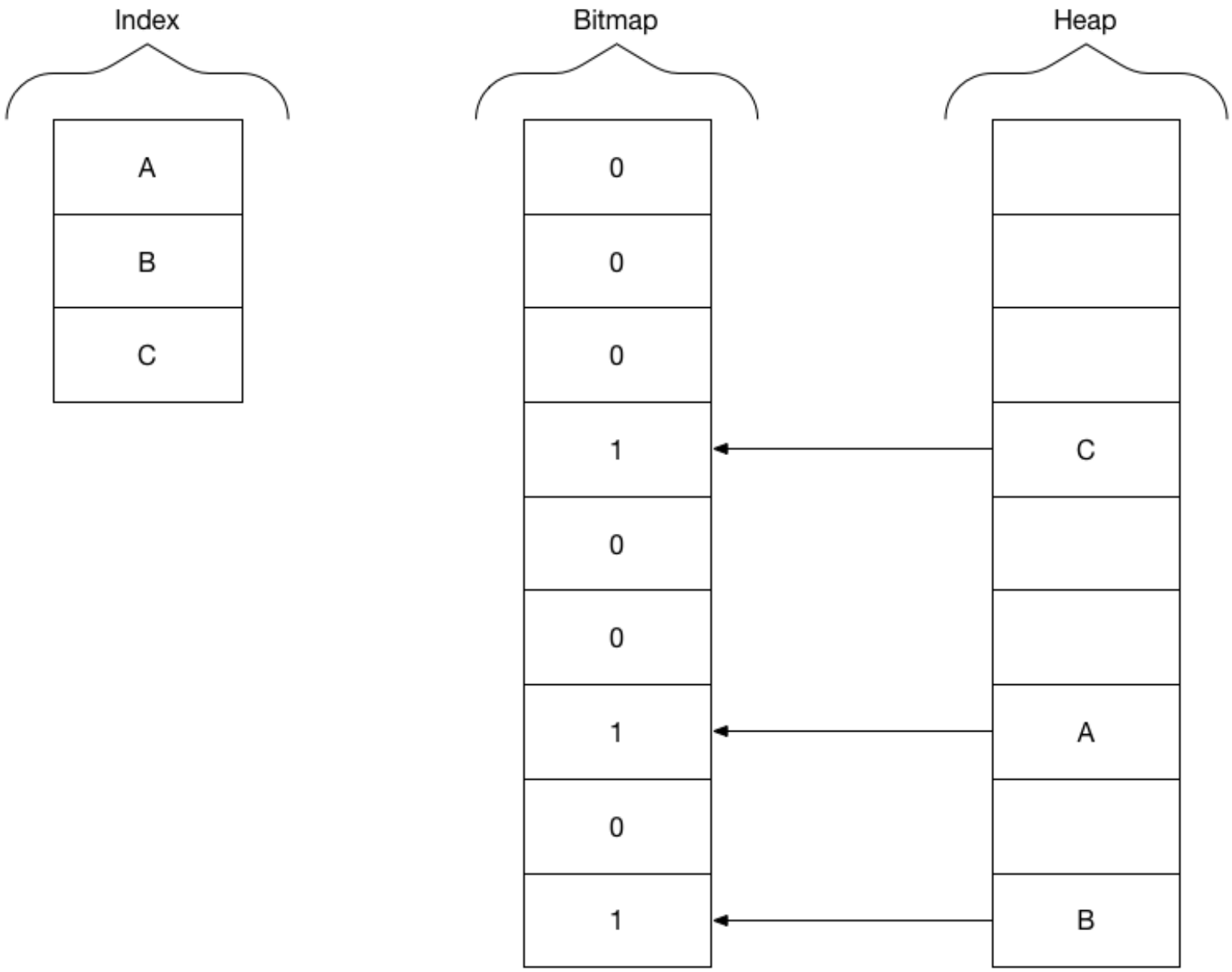
Number of conferences with over 100 attendees	15,970
Number of conferences with exactly 500 attendees	52
Total number of conferences	20,000

16

```
EXPLAIN ANALYZE SELECT * FROM
conferences WHERE total_attendees >
100;

QUERY PLAN
-----------------------------------------
Seq Scan on conferences
(cost=0.00..427.00 rows=15952 width=39)
(actual time=0.018..8.384 rows=15970
loops=1)
  Filter: (total_attendees > 100)
  Rows Removed by Filter: 4030
Planning time: 0.111 ms
Execution time: 9.681 ms
```

```
EXPLAIN ANALYZE SELECT * FROM conferences
WHERE total_attendees = 500;

QUERY PLAN
------------------------------------
Bitmap Heap Scan on conferences
(cost=4.69..118.99 rows=52 width=39)
(actual time=0.074..0.158 rows=52 loops=1)
  Recheck Cond: (total_attendees = 500)
  Heap Blocks: exact=45
  ->  Bitmap Index Scan on
conferences_total_attendees_idx
(cost=0.00..4.68 rows=52 width=0)
(actual time=0.059..0.059 rows=52 loops=1)
    Index Cond: (total_attendees = 500)
Planning time: 0.119 ms
Execution time: 0.198 ms
```

Bitmap scan



18

```
CREATE INDEX ON
conferences(expected_attendees);

SELECT * FROM conferences
WHERE total_attendees > 400
AND expected_attendees < 150;
```

21

```
SELECT count(*) FROM conferences
WHERE total_attendees > 400;
4002

SELECT count(*) FROM conferences
WHERE expected_attendees < 100;
3860

SELECT count(*) FROM conferences
WHERE expected_attendees < 150;
5819
```

```
EXPLAIN ANALYZE SELECT * FROM conferences
WHERE total_attendees > 400 and
expected_attendees < 150;
QUERY PLAN
-----------------------------------
Bitmap Heap Scan on conferences
(cost=78.68..315.87 rows=1174 width=39)
(actual time=0.889..2.780 rows=1193 loops=1)
Recheck Cond: (total_attendees > 400)
Filter: (expected_attendees < 150)
Rows Removed by Filter: 2809
Heap Blocks: exact=177
  ->  Bitmap Index Scan on
conferences_total_attendees_idx
(cost=0.00..78.38 rows=4013 width=0) (actual
time=0.845..0.845 rows=4002 loops=1)
Index Cond: (total_attendees > 400)
Planning time: 0.152 ms
Execution time: 2.872 ms
```

20

```
EXPLAIN ANALYZE SELECT * FROM conferences
WHERE total_attendees > 400 and
expected_attendees < 100;
QUERY PLAN
-----------------------------------------
Bitmap Heap Scan on conferences
(cost=77.60..312.85 rows=779 width=39)
(actual time=0.852..2.585 rows=826 loops=1)
Recheck Cond: (expected_attendees < 100)
Filter: (total_attendees > 400)
Rows Removed by Filter: 3034
Heap Blocks: exact=177
->  Bitmap Index Scan on
conferences_expected_attendees_idx
(cost=0.00..77.41 rows=3883 width=0) (actual
time=0.805..0.805 rows=3860 loops=1)
Index Cond: (expected_attendees < 100)
Planning time: 0.145 ms
Execution time: 2.682 ms
```

Scans

- **Sequential scan**
  - no index
  - there is an index but a very large proportion of the table is being returned
- **Index scan**
  - index and fetching a small number of tuples
- **Bitmap heap scan**
  - index and fetching a medium number of tuples
  - often used in multi-index searches

Query planner settings

- 22
  - `set enable_bitmapscan=off`
  - `set enable_indexscan=off`
  - `set enable_indexonlyscan=off`

Query planner cost settings

	What?	Default cost
`seq_page_cost`	Read single database page from disk	**1.0**
`random_page_cost`	Read when rows involved are scattered randomly across disks	**4.0**
`cpu_tuple_cost`	Process a row of data	**0.01**
`cpu_index_tuple_cost`	Process an index entry during an index scan	**0.005**
`cpu_operator_cost`	Process an operator or function	**0.025**

Joins

- Nested Loop Join
- Hash Join
- Merge Join

🔷 23

```
CREATE TABLE sponsors (name text,
conference_id int, employees int);

DO
$do$
BEGIN
FOR i IN 1..20000 LOOP
  INSERT INTO sponsors
  (name, conference_id, employees)
  SELECT 'sponsor ' || s,
  i,
  (random() * 500 + 1)::int
  FROM generate_series(1,10) s;
END LOOP;
END
$do$;
```

🔷 24

```
DROP INDEX conferences_conference_id_idx;
```

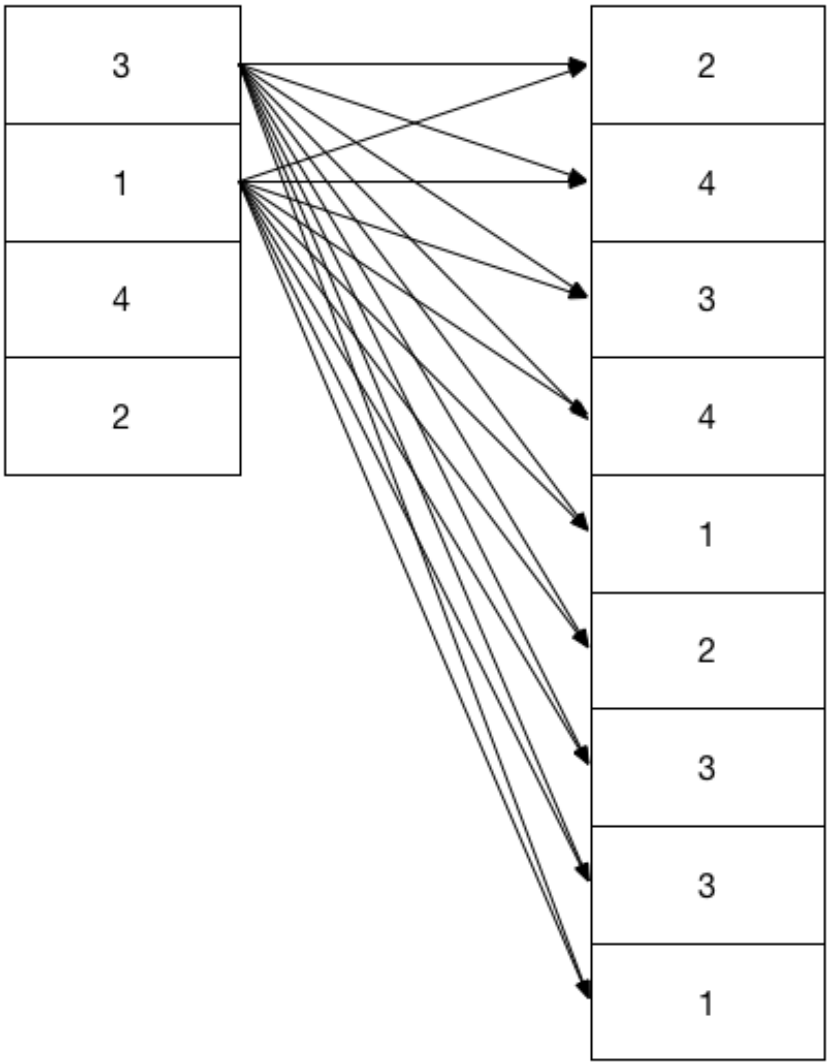
🔷 25

```
SELECT * FROM sponsors, conferences
WHERE sponsors.conference_id = 1
AND sponsors.conference_id =
conferences.conference_id;
```

26

```
EXPLAIN ANALYZE SELECT * FROM sponsors, conferences
WHERE sponsors.conference_id = 1
AND sponsors.conference_id =
conferences.conference_id;

QUERY PLAN
------------------------------------------------------------
Nested Loop  (cost=0.00..4009.25 rows=10 width=50) (actual
time=0.031..62.902 rows=10 loops=1)
->  Seq Scan on conferences  (cost=0.00..427.15 rows=1
width=39) (actual time=0.017..6.304 rows=1 loops=1)
Filter: (conference_id = 1)
Rows Removed by Filter: 20011
->  Seq Scan on sponsors  (cost=0.00..3582.00 rows=10
width=11) (actual time=0.011..56.593 rows=10 loops=1)
Filter: (conference_id = 1)
Rows Removed by Filter: 199990
Planning time: 0.206 ms
Execution time: 62.942 ms
```

Nested loop
with sequential scan



🔷 27

```
CREATE INDEX ON sponsors(conference_id);
CREATE INDEX ON conferences(conference_id);
```

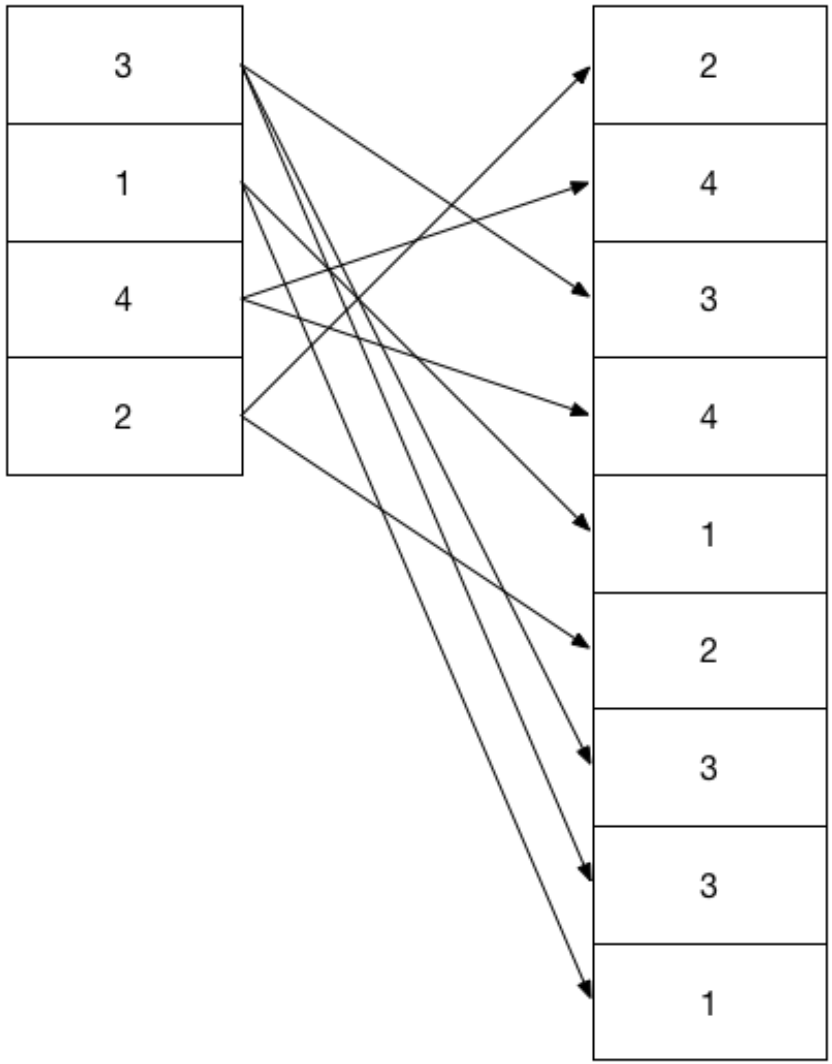
🔷 28

```
CREATE INDEX CONCURRENTLY ON
sponsors(conference_id);

CREATE INDEX CONCURRENTLY ON
conferences(conference_id);
```

```
EXPLAIN ANALYZE SELECT * from sponsors, conferences
WHERE sponsors.conference_id = 1
AND sponsors.conference_id =
conferences.conference_id;
```


```
QUERY PLAN
-----------------------------------------------------
-----------------------------------------------------
Nested Loop  (cost=0.71..17.00 rows=10 width=50) (actual
time=0.090..0.098 rows=10 loops=1)
->  Index Scan using conferences_conference_id_idx
on conferences  (cost=0.29..8.30 rows=1 width=39) (actual
time=0.023..0.024 rows=1 loops=1)
Index Cond: (conference_id = 1)
->  Index Scan using sponsors_conference_id_idx on
sponsors  (cost=0.42..8.59 rows=10 width=11) (actual
time=0.062..0.067 rows=10 loops=1)
Index Cond: (conference_id = 1)
Planning time: 0.455 ms
Execution time: 0.145 ms
```

Nested loop
with inner index scan



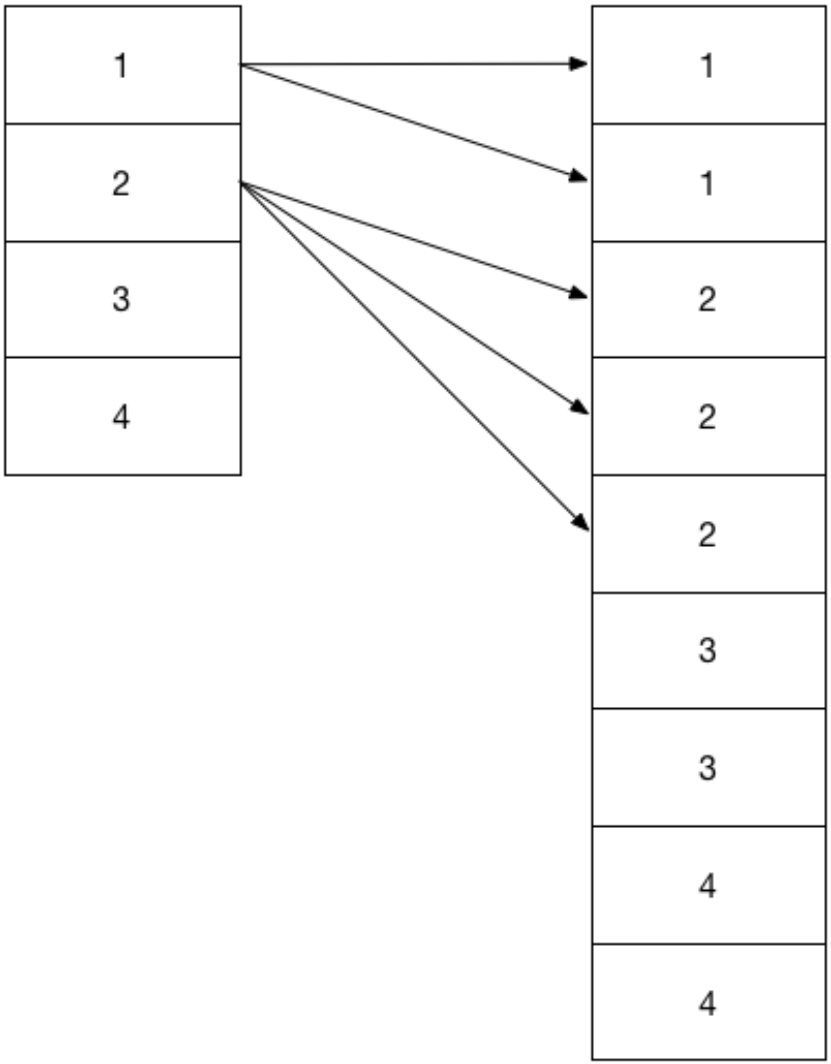
🔷 30

```
SELECT conferences.conference_id, avg(employees)
FROM conferences, sponsors
WHERE conferences.conference_id =
sponsors.conference_id
GROUP BY conferences.conference_id;
```

```
EXPLAIN ANALYZE SELECT conferences.conference_id,
avg(employees) FROM conferences, sponsors WHERE
conferences.conference_id = sponsors.conference_id GROUP
BY conferences.conference_id;
```


```
QUERY PLAN
-----------------------------------------------------
GroupAggregate  (cost=0.71..10982.49 rows=20012 width=36)
(actual time=0.065..305.677 rows=20000 loops=1)
Group Key: conferences.conference_id
->  Merge Join  (cost=0.71..9732.34 rows=200000 width=8)
(actual time=0.040..205.731 rows=200000 loops=1)
Merge Cond: (conferences.conference_id =
sponsors.conference_id)
->  Index Only Scan using conferences_conference_id_idx1
on conferences  (cost=0.29..708.47 rows=20012 width=4) (actual
time=0.021..12.036 rows=20001 loops=1)
Heap Fetches: 20001
->  Index Scan using sponsors_conference_id_idx on
sponsors  (cost=0.42..6481.42 rows=200000 width=8) (actual
time=0.014..92.679 rows=200000 loops=1)
```

Merge join



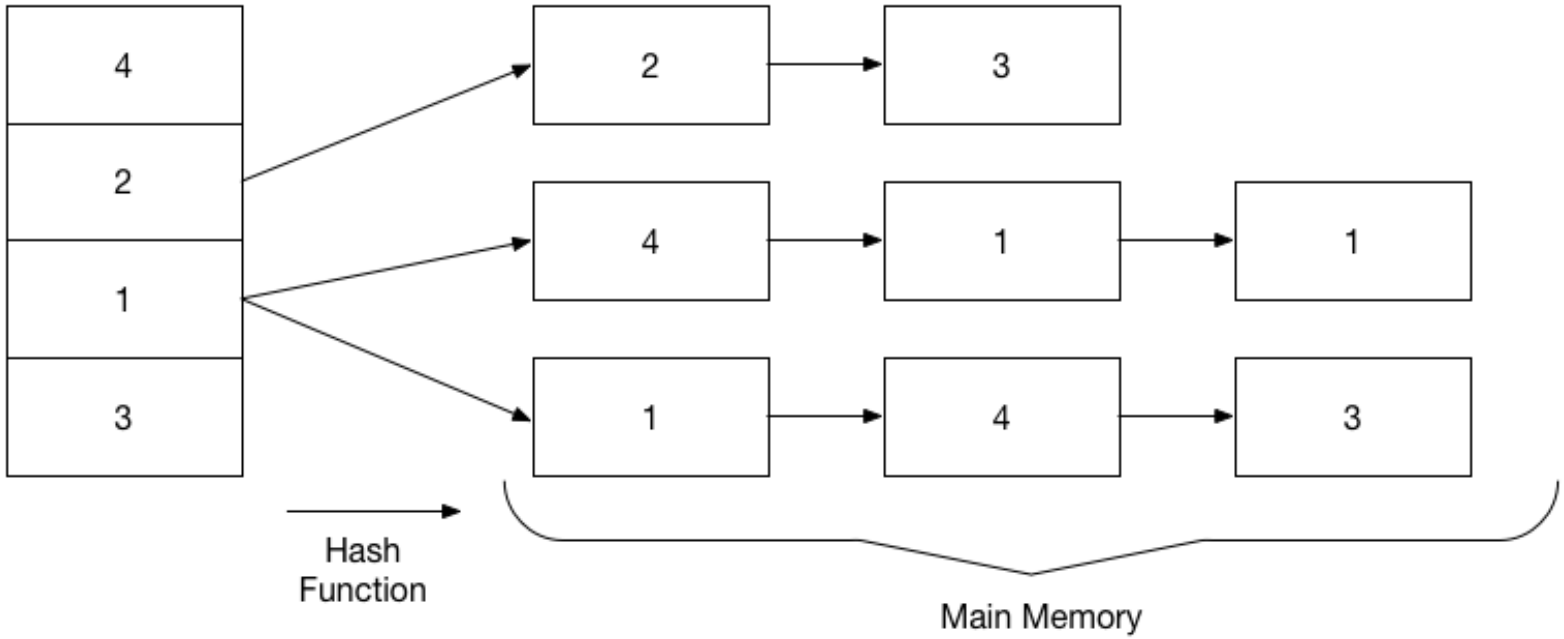
🔷 32

```
SELECT * FROM sponsors, conferences
WHERE sponsors.conference_id =
conferences.conference_id;
```

```
EXPLAIN ANALYZE SELECT * FROM sponsors, conferences
WHERE sponsors.conference_id =
conferences.conference_id;
```


```
QUERY PLAN
------------------------------------------------------------
Hash Join  (cost=784.27..8725.47 rows=199820 width=50)
(actual time=20.175..316.524 rows=200000 loops=1)
Hash Cond: (sponsors.conference_id =
conferences.conference_id)
->  Seq Scan on sponsors (cost=0.00..3082.00 rows=200000
width=11) (actual time=0.016..41.973 rows=200000 loops=1)
->  Hash  (cost=377.12..377.12 rows=20012 width=39) (actual
time=18.105..18.105 rows=20012 loops=1)
Buckets: 2048  Batches: 16  Memory Usage: 104kB
->  Seq Scan on conferences (cost=0.00..377.12 rows=20012
width=39) (actual time=0.009..5.532 rows=20012 loops=1)
Planning time: 0.534 ms
Execution time: 330.907 ms
```

Hash join



EXPLAIN: where to get started

- Look at longest execution times
- Look whether they had matching costs
- For each of the expensive parts of the query, can it be changed by modifying what indexes you have?
- Can it be made better by writing a different query/ changing your data structure?
- Can it be made better by forcing a different type of ordering/cost evaluation?

Joins

- The more joins, the more planning errors will be costly and join order will matter.
  - If you are certain of the order, consider setting **`join_collapse_limit`** to 1 for a query to force the order
- **`effective_cache_size`** is used to determined whether an index scan will fit in memory
  - it is an estimate of how much memory is available for disk caching by the operating system and within the database itself, after taking into account what's used by the OS itself
  - the lower it is, the less likely nested loops will be considered by the query optimizer

```
EXPLAIN ANALYZE SELECT description FROM conferences ORDER BY
name;

--with work_mem as 3MB

QUERY PLAN
-----------------------------------------------------
Sort  (cost=1805.77..1855.77 rows=20000 width=27)
Sort Key: name
Sort Method: quicksort  Memory: 2272kB
->  Seq Scan on conferences  (cost=0.00..377.00 rows=20000
width=27) (actual time=0.025..7.762 rows=20000 loops=1)

--with work_mem as 1MB

QUERY PLAN
-----------------------------------------------------
Sort  (cost=2285.27..2335.27 rows=20000 width=27)
Sort Key: name
Sort Method: external merge  Disk: 736kB
->  Seq Scan on conferences  (cost=0.00..377.00 rows=20000
width=27) (actual time=0.018..6.836 rows=20000 loops=1)
```

Don’t index everything!

- Adding an index is a trade-off between its benefits and the cost of maintaining it
- There are ways of determining statistically if an index is needed, via looking at some postgres views, or installing an extension such as pg_hypo.

\d pg_stat_user_tables;

relic

schema name

relname

seq_scan

seq_tup_read

idx_scan

idx_tup_fetch

n_tup_ins

n_tup_upd

n_tup_del

n_tup_hot_upd

n_live_tup

n_dead_tup

n_mod_since_analyze

last_vacuum

last_autovacuum

last_analyze

last_autoanalyze

…

Unused indexes

🔷 **34**

```
WITH table_scans as (
    SELECT relid,
        tables.idx_scan + tables.seq_scan as all_scans,
        (tables.n_tup_ins + tables.n_tup_upd +
            tables.n_tup_del) as writes,
         pg_relation_size(relid) as table_size
    FROM pg_stat_user_tables as tables
)
```

Unused indexes

🔷 **34**

```
WITH table_scans as (
    SELECT relid,
        tables.idx_scan + tables.seq_scan as all_scans,
        (tables.n_tup_ins + tables.n_tup_upd +
            tables.n_tup_del) as writes,
         pg_relation_size(relid) as table_size
    FROM pg_stat_user_tables as tables
)
```

Unused indexes

🔷 **34**

```
WITH table_scans as (
    SELECT relid,
        tables.idx_scan + tables.seq_scan as all_scans,
        (tables.n_tup_ins + tables.n_tup_upd +
            tables.n_tup_del) as writes,
         pg_relation_size(relid) as table_size
    FROM pg_stat_user_tables as tables
)
```

Unused indexes

🔷 **34**

```
WITH table_scans as (
    SELECT relid,
        tables.idx_scan + tables.seq_scan as all_scans,
        (tables.n_tup_ins + tables.n_tup_upd +
            tables.n_tup_del) as writes,
         pg_relation_size(relid) as table_size
    FROM pg_stat_user_tables as tables
)
```

Unused indexes

🔷 34

```
WITH indexes as (
    SELECT idx_stat.relid, idx_stat.indexrelid,
        idx_stat.schemaname, idx_stat.relname as tablename,
        idx_stat.indexrelname as indexname,
        idx_stat.idx_scan,
        pg_relation_size(idx_stat.indexrelid) as index_bytes,
        indexdef ~* 'USING btree' AS idx_is_btree
    FROM pg_stat_user_indexes as idx_stat
        JOIN pg_index
            USING (indexrelid)
        JOIN pg_indexes as indexes
            ON idx_stat.schemaname = indexes.schemaname
                AND idx_stat.relname = indexes.tablename
                AND idx_stat.indexrelname = indexes.indexname
    WHERE pg_index.indisunique = FALSE
)
```

Unused indexes

🔷 **34**

```
WITH indexes as (
    SELECT idx_stat.relid, idx_stat.indexrelid,
        idx_stat.schemaname, idx_stat.relname as tablename,
        idx_stat.indexrelname as indexname,
        idx_stat.idx_scan,
        pg_relation_size(idx_stat.indexrelid) as index_bytes,
        indexdef ~* 'USING btree' AS idx_is_btree
    FROM pg_stat_user_indexes as idx_stat
        JOIN pg_index
            USING (indexrelid)
        JOIN pg_indexes as indexes
            ON idx_stat.schemaname = indexes.schemaname
                AND idx_stat.relname = indexes.tablename
                AND idx_stat.indexrelname = indexes.indexname
    WHERE pg_index.indisunique = FALSE
)
```

Unused indexes

🔷 34

```
SELECT 'Never Used Indexes' as reason, *, 1 as grp
FROM index_ratios
WHERE
    idx_scan = 0
    and idx_is_btree
```

Unused indexes

🔷 34

```
SELECT 'Low Scans, High Writes' as reason, *, 2 as grp
FROM index_ratios
WHERE
    scans_per_write <= 1
    and index_scan_pct < 10
    and idx_scan > 0
    and writes > 100
    and idx_is_btree
```

Query plans: further optimizations

- **`DISTINCT`** is usually a bad idea
- using **`COPY`** instead of **`INSERT`**
- using expression indexes
- using partial indexes
- using multi-column indexes
- **`CLUSTER`**
- prepared statements

Query plans

- Use **`EXPLAIN,`** prioritize which part to fix first
- Add indexes if you need them
  - But check that they are being used and that they are helping
- Understand the cost of various join methods and rethink how to query data or indexing to minimize their cost, and adjust settings as necessary to end up with the optimal query plans
- Delete unused indexes, consider deleting underused indexes

Troubleshooting performance issues in PostgreSQL

- Query execution
- Query plans
- **At scale: high traffic, growing data**
- Other performance factors (PostgreSQL version, hardware) and how to benchmark them
- What to monitor

What problems?

- Increased traffic
- More types of queries
- Accelerated rate of writes
- Tables growing larger and larger

At scale

- Replication
- Managing connections
- Managing bloat
- Large tables

Replication

- Standby
  - standby server can read WAL from a WAL archive or directly from the master over a TCP connection (streaming replication)
  - set up followers where read-only queries can be run
  - use those followers to run other I/O heavy operations, such as base backups, or pg_dump

Replication

- Logical replication
  - Postgres 10 (or earlier with extensions like pg_logical)
  - set up followers that have a subset or all of your tables
  - these databases are not not read-only so you can setup different indexes than on their parents for analytics-specific queries

Logical replication (10)

```
CREATE PUBLICATION pub_conferences FOR TABLE conferences;

CREATE PUBLICATION pub_everything FOR ALL TABLES;

CREATE SUBSCRIPTION sub_conferences CONNECTION 'host=...
dbname=...' PUBLICATION pub_conferences;
```

At scale

- Replication
- **Managing bloat**
- Managing connections
- Large tables

Bloat

- Dead rows are generated by **`UPDATE`**s, **`DELETE`**s and rolling back transactions.
  - Your database will need periodic maintenance to clean out those dead rows.

Bloat

🔷 35

```
SELECT pg_stat_get_live_tuples(c.oid) AS n_live_tup,
pg_stat_get_dead_tuples(c.oid) AS n_dead_tup
FROM pg_class c WHERE relname='conferences';

n_live_tup | n_dead_tup
------------+------------
     20000 |          0

UPDATE conferences SET name = 'something';

n_live_tup | n_dead_tup
------------+------------
     20000 |      20000

-- wait for a bit or run
VACUUM conferences;
```

Fixing bloat issues

- PostgreSQL has an autovacuum process to manage bloat

Autovacuum

- automates the execution of **`VACUUM`** and **`ANALYZE`** commands
- checks for tables that have had a large number of inserted, updated or deleted tuples

Autovacuum settings

- when it is triggered
- throttling
- workers and memory

When autovacuum is triggered

- **`autovacuum_vacuum_threshold`**
- **`autovacuum_vacuum_scale_factor`**
- Autovacuum starts when:
  - **`pg_stat_all_tables.n_dead_tup > threshold + pg_class.reltuples * scale_factor`**

When autovacuum is triggered

- **`ALTER TABLE conferences SET (autovacuum_vacuum_scale_factor = 0.01);`**
- For large tables consider:
  - **`pg_stat_all_tables.n_dead_tup > threshold + pg_class.reltuples * scale_factor`**
  - decreasing the scale factor
  - solely relying on the threshold

Autovacuum settings

- when it is triggered
- **throttling**
- workers and memory

Autovacuum throttling

- Limits the amount of work that can be done in one go
- **`autovacuum_vacuum_cost_limit`**
- **`autovacuum_vacuum_cost_delay`**

Autovacuum settings

- when it is triggered
- throttling
- **workers and memory**

Autovacuum workers and memory

- **`autovacuum_max_workers`**
- **`maintenance_work_mem`** or **`autovacuum_work_mem`**

Autovacuum settings

- when it is triggered
- throttling
- workers and memory
- **choose values to balance bloat management and resource usage**

Fixing bloat issues

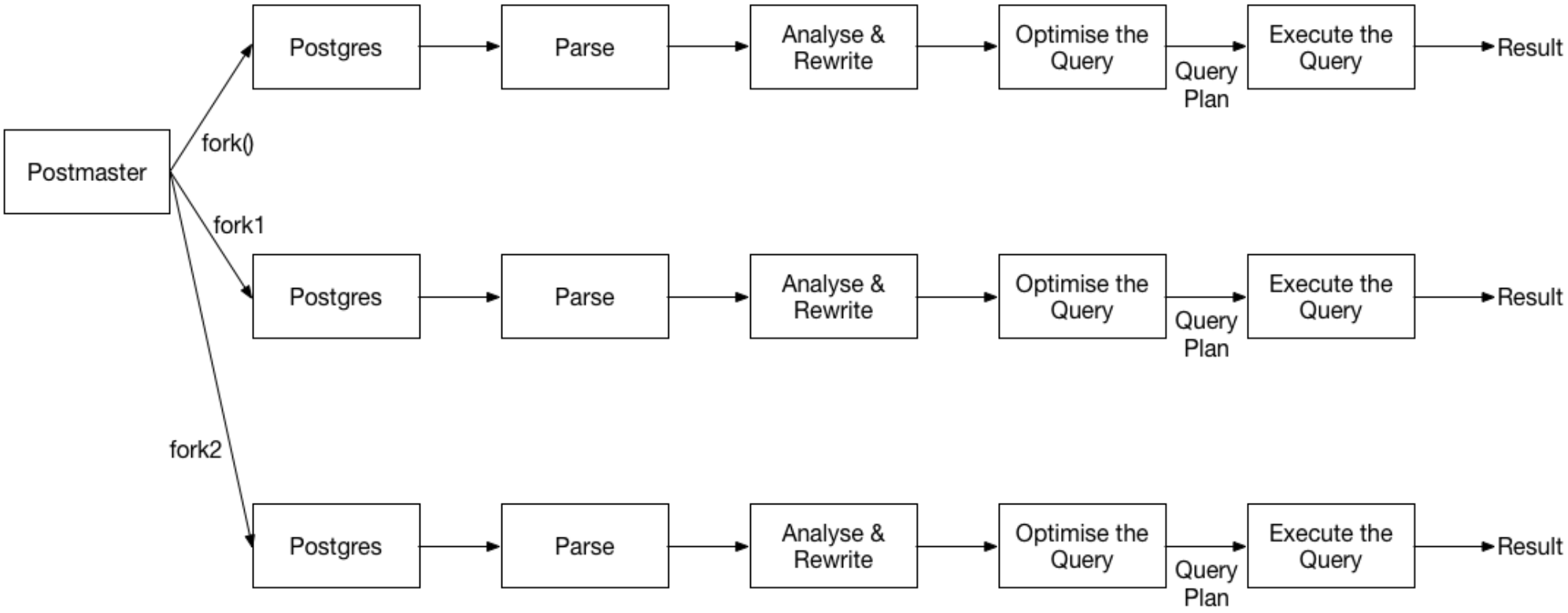
- PostgreSQL has an autovacuum process to manage bloat, but sometimes more aggressive methods such as **`VACUUM, VACUUM FULL`** will be needed.
  - **`VACUUM`** will not generally return the disk space back to the operating system, but it will make it usable for new rows. Run **`ANALYZE`** after vacuuming.
  - If disk space has to be returned to the operating system, **`VACUUM FULL`** may be needed.

VACUUM FULL vs VACUUM

- **`VACUUM FULL`**
  - access exclusive lock on the table
  - requires more disk space
  - only use when a significant amount of space needs to be reclaimed from within the table
- **`VACUUM`**
  - share update exclusive lock
  - queries can still **`INSERT`**, **`UPDATE`**, **`DELETE`** but can't do other operations like **`CREATE INDEX`**

At scale

- Replication
- Managing bloat
- **Managing connections**
- Large tables


Postmaster
fork()
fork1
fork2
Postgres
Parse
Analyse & Rewrite
Optimise the Query
Query Plan
Execute the Query
Result
Postgres
Parse
Analyse & Rewrite
Optimise the Query
Query Plan
Execute the Query
Result
Postgres
Parse
Analyse & Rewrite
Optimise the Query
Query Plan
Execute the Query
Result

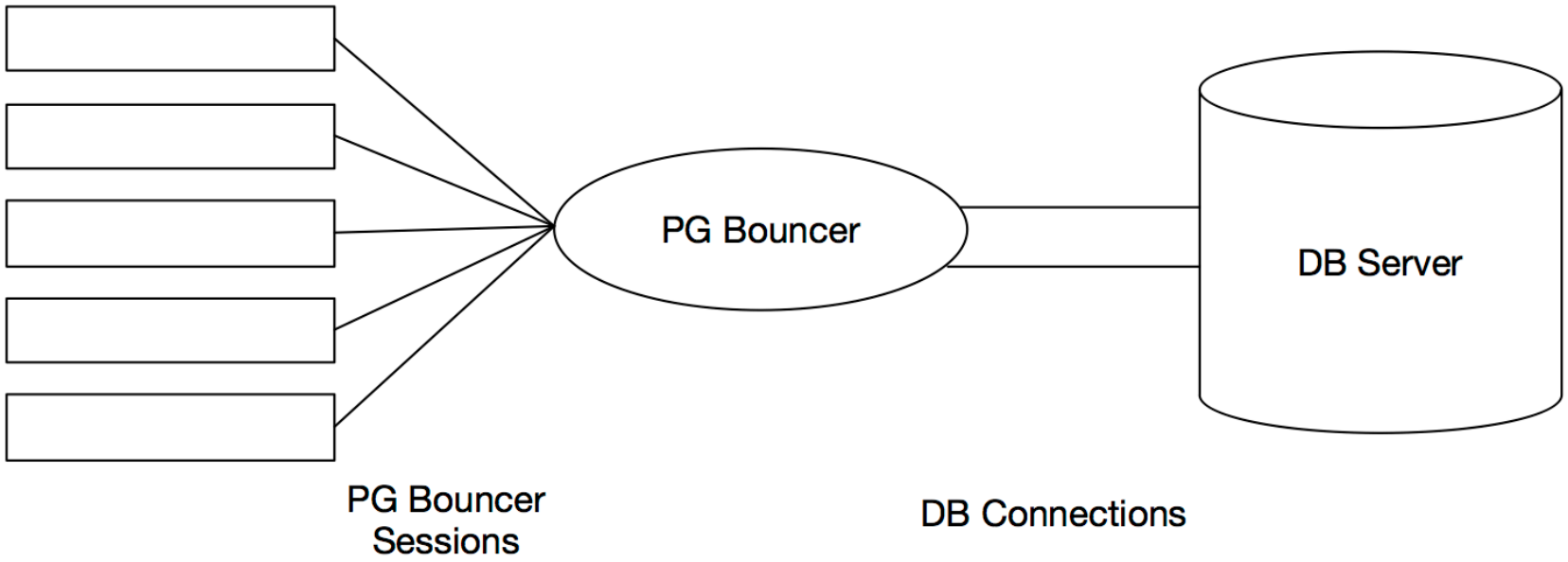
Managing connections

- Making a new connection in PostgreSQL can be slow. A whole new process needs to be forked.
- Each new connection takes more space in memory
  - **`work_mem`**
- Memory is also needed for
  - **`shared_buffers`**
  - **`maintenance_work_mem, autovacuum_max_workers autovacuum_work_mem`**

Managing connections

- Consider a connection pooler.
- Some application servers implement Database Connection Pools such as Apache Tomcat (DBCP).

pgBouncer



pgBouncer

- pgBouncer is one of the most common PostgreSQL connection poolers.
  - It relies on UNIX's libevent, like memcached.
  - Allows you to monitor the pool
  - Control connection limits per database/per user
  - Pooling per session, transaction, or statement

At scale

- Replication
- Managing bloat
- Managing connections
- **Large tables**

Large tables

- When a table grows beyond the size of PostgreSQL's physical memory
  - Indexes will grow too
  - Queries will slow down
- Table partitioning can help with those issues without ending changes to your application code

Tools for partitioning

- “Manual”
- pg_partman (time-based and serial-based table partition sets)
- Native partitioning in Postgres 10

36

```
UPDATE conferences
SET start_date = '1/1/2017'::date + ('1
day'::interval*floor(random()*180));

CREATE INDEX ON conferences(start_date);

CREATE TABLE conferences_2017_q1 (
CHECK (start_date >= DATE '2017-01-01'
AND start_date < DATE '2017-04-01'))
INHERITS (conferences);

CREATE TABLE conferences_2017_q2 (
CHECK (start_date >= DATE '2017-04-01'
AND start_date < DATE '2017-07-01'))
INHERITS (conferences);
```

37

```
CREATE OR REPLACE FUNCTION conferences_update_trigger()
RETURNS TRIGGER AS
$$
BEGIN
  IF (NEW.start_date >= DATE ('2017-01-01') AND
NEW.start_date < DATE('2017-04-01')) THEN
  INSERT INTO conferences_2017_q1 VALUES (NEW.*);
  ELSIF (NEW.start_date >= DATE ('2017-04-01') AND
NEW.start_date < DATE('2017-07-01')) THEN
  INSERT INTO conferences_2017_q2 VALUES (NEW.*);
  ELSE
RAISE EXCEPTION 'start_date date out of range';
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_conferences_trigger BEFORE UPDATE ON
conferences FOR EACH ROW EXECUTE PROCEDURE
conferences_update_trigger();
```

38

```
SELECT count(*) FROM conferences;
count
-------
20000

SELECT count(*) FROM conferences_2017_q1;
count
-------
0

UPDATE conferences set start_date=start_date;

SELECT count(*) FROM conferences_2017_q1;
count
-------
9942
```

```
EXPLAIN ANALYZE SELECT * FROM conferences WHERE start_date <
'02-11-2017' AND start_date > '01-11-2017';

QUERY PLAN
-----------------------------------------------------------
Append  (cost=74.89..731.66 rows=6670 width=43) (actual
time=1.219..6.881 rows=6588 loops=1)
->  Bitmap Heap Scan on conferences  (cost=74.89..489.53 rows=3376
width=43) (actual time=1.219..2.264 rows=3294 loops=1)
Recheck Cond: ((start_date < '2017-02-11'::date) AND (start_date >
'2017-01-11'::date))
Heap Blocks: exact=188
->  Bitmap Index Scan on conferences_start_date_idx
(cost=0.00..74.05 rows=3376 width=0) (actual time=1.177..1.177
rows=3294 loops=1)
Index Cond: ((start_date < '2017-02-11'::date) AND (start_date >
'2017-01-11'::date))
->  Seq Scan on conferences_2017_q1  (cost=0.00..242.13 rows=3294
width=43) (actual time=0.021..3.861 rows=3294 loops=1)
Filter: ((start_date < '2017-02-11'::date) AND (start_date >
'2017-01-11'::date))
Rows Removed by Filter: 6648
Planning time: 0.755 ms
Execution time: 7.349 ms
```

```
CREATE INDEX CONCURRENTLY ON
conferences_2017_q1(start_date);

CREATE INDEX CONCURRENTLY ON
conferences_2017_q2(start_date);
```

```
\d conferences

Column         |  Type   | Modifiers
---------------------+---------+-----------
name                 | text    |
description          | text    |
conference_id        | integer |
total_attendees      | integer | default 0
expected_attendees | integer | default 0
start_date           | date    |
Indexes:
"conferences_start_date_idx" btree (start_date) CLUSTER
Triggers:
update_conferences_trigger BEFORE UPDATE ON conferences FOR
EACH ROW EXECUTE PROCEDURE conferences_update_trigger()

CLUSTER conferences using conferences_start_date_idx;

ANALYZE conferences;
```

39

```
EXPLAIN ANALYZE SELECT * FROM conferences WHERE start_date <
'02-11-2017' AND start_date > '01-11-2017';

QUERY PLAN
-----------------------------------------------------------------
Append  (cost=0.29..664.44 rows=9964 width=43) (actual
time=0.054..5.387 rows=9882 loops=1)
->  Index Scan using conferences_start_date_idx on conferences
(cost=0.29..142.81 rows=3376 width=43) (actual time=0.054..1.145
rows=3294 loops=1)
Index Cond: ((start_date < '2017-02-11'::date) AND (start_date >
'2017-01-11'::date))
->  Bitmap Heap Scan on conferences_2017_q1  (cost=143.81..521.63
rows=6588 width=43) (actual time=1.379..3.066 rows=6588 loops=1)
Recheck Cond: ((start_date < '2017-02-11'::date) AND (start_date >
'2017-01-11'::date))
Heap Blocks: exact=187
->  Bitmap Index Scan on conferences_2017_q1_start_date_idx
(cost=0.00..142.17 rows=6588 width=0) (actual time=1.339..1.339
rows=6588 loops=1)
Index Cond: ((start_date < '2017-02-11'::date) AND (start_date >
'2017-01-11'::date))
Planning time: 0.407 ms
Execution time: 6.686 ms
```

```
EXPLAIN ANALYZE SELECT * FROM conferences WHERE total_attendees <
200;

QUERY PLAN
-----------------------------------------------------------
Append  (cost=0.00..1498.00 rows=23848 width=43) (actual
time=0.022..19.678 rows=23829 loops=1)
->  Seq Scan on conferences  (cost=0.00..437.00 rows=7967
width=43) (actual time=0.022..6.026 rows=7943 loops=1)
Filter: (total_attendees < 200)
Rows Removed by Filter: 12057
->  Seq Scan on conferences_2017_q1  (cost=0.00..527.55 rows=7999
width=43) (actual time=0.114..5.890 rows=7990 loops=1)
Filter: (total_attendees < 200)
Rows Removed by Filter: 11894
->  Seq Scan on conferences_2017_q2  (cost=0.00..533.45 rows=7882
width=43) (actual time=0.115..4.509 rows=7896 loops=1)
Filter: (total_attendees < 200)
Rows Removed by Filter: 12220
Planning time: 0.567 ms
Execution time: 21.178 ms
```

Tools for partitioning

- “Manual”
- pg_partman (time-based and serial-based table partition sets)
- Native partitioning in Postgres 10

At scale

- Replication
- Managing connections
- Managing bloat
- Large tables

Troubleshooting performance issues in PostgreSQL

- Query execution
- Query plans
- At scale: high traffic, growing data
- **Other performance factors (PostgreSQL version, hardware) and how to benchmark them**
- What to monitor

Other performance factors

- PostgreSQL version
- Hardware
- Benchmarking to check the changes made are correct

PostgreSQL versions impact on performance

- This graph comes from https://blog.pgaddict.com/posts/performance-since-postgresql-7-4-to-9-4-pgbench.

9.6 improvements

- Parallel query (sequentials scans, aggergations, joins on read-only queries)
- 9.6 sorts shared buffers first, instead of reading them randomly which was causing random IO
- Sort performance

10 improvements

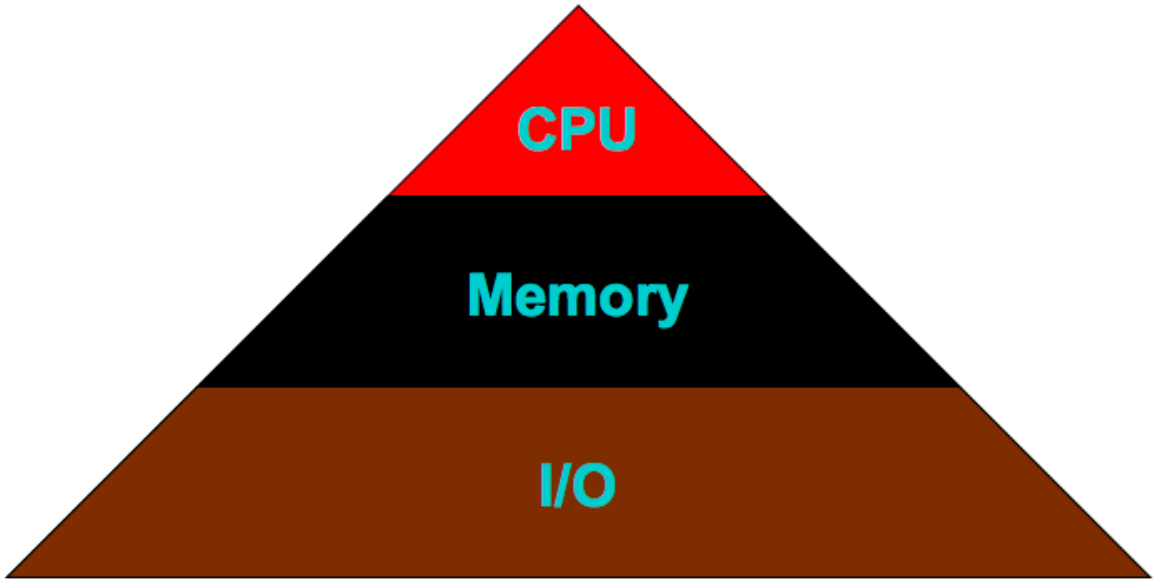
- INSERT performance
- multi-column statistics
- improved query parallelism

Other performance factors

- PostgreSQL version
- **Hardware**
- Benchmarking to check the changes made are correct

Hardware

- This excellent diagram comes from Bruce Momjian's database hardware selection guidelines, https://momjian.us/main/writings/pgsql/hw_selection.pdf.



Memory

- If your data is small enough to fit in memory, adding more won't help. Try better CPU instead.
- If the tables you are scanning cannot fit in memory, then faster disks will be more helpful than more RAM.

CPU

- Recent versions of Postgres are doing more and more in parallel where multiple cores can be used to your advantage.
  - 9.6 supports parallel querying

Other performance factors

- PostgreSQL version
- Hardware
- **Benchmarking to check the changes made are correct**

Benchmarking

- **`pgbench`** is a good place to get started for PostgreSQL benchmarking and checking your choice of hardware/settings/version is working well. It supports:
  - multiple query modes
  - adjustable number of clients
  - adding foreign key constraints
  - custom benchmark scenarios

43

psql

```
CREATE DATABASE tutorial_bench;
```

44

Terminal

```
pgbench -i -s 50 tutorial_bench
100000 of 5000000 tuples (2%) done (elapsed 0.12 s, remaining
6.05 s)
...
5000000 of 5000000 tuples (100%) done (elapsed 9.63 s, remaining
0.00 s)
vacuum...
set primary keys...
done.
```

🔷 45

psql tutorial_bench

```
\dt
List of relations
Schema |        Name        | Type  |      Owner
-------+--------------------+-------+-----------------
public | pgbench_accounts   | table | camillebaldock
public | pgbench_branches   | table | camillebaldock
public | pgbench_history    | table | camillebaldock
public | pgbench_tellers    | table | camillebaldock
```

```
SELECT pg_size_pretty(pg_database_size(pg_database.datname)) AS
size
FROM pg_database where datname = 'tutorial_bench';
size
--------
768 MB
```

🔷 46

Terminal

```
pgbench -c 10 -j 2 -t 10000 tutorial_bench

starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
latency average = 3.535 ms
tps = 2829.122821 (including connections establishing)
tps = 2829.406348 (excluding connections establishing)
```

🔷 46

Terminal

```
pgbench -c 10 -j 2 -t 10000 tutorial_bench

starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
latency average = 3.535 ms
tps = 2829.122821 (including connections establishing)
tps = 2829.406348 (excluding connections establishing)
```

🔷 46

Terminal

```
pgbench -c 10 -j 2 -t 10000 tutorial_bench

starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
latency average = 3.535 ms
tps = 2829.122821 (including connections establishing)
tps = 2829.406348 (excluding connections establishing)
```

```
BEGIN;

UPDATE pgbench_accounts SET abalance = abalance + :delta
WHERE aid = :aid;

SELECT abalance FROM pgbench_accounts WHERE aid = :aid;

UPDATE pgbench_tellers SET tbalance = tbalance + :delta
WHERE tid = :tid;

UPDATE pgbench_branches SET bbalance = bbalance + :delta
WHERE bid = :bid;

INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);

END;
```

pgbench

- Can be run with custom scripts
- Try monitoring your IO/memory/CPU use while running it

Other performance factors

- PostgreSQL version: you probably want the latest
- Hardware choices
- Benchmark, always benchmark!

Troubleshooting performance issues in PostgreSQL

- Query execution
- Query plans
- At scale: high traffic, growing data
- Other performance factors (PostgreSQL version, hardware) and how to benchmark them
- **What to monitor**

Monitoring

- Finding slow queries through pg_stat_statements
- Monitoring locks and blocked queries through pg_stat_activity and pg_locks
- Monitoring bloat, vacuuming and cache hit rates
- UNIX monitoring

```
CREATE EXTENSION pg_stat_statements;
shared_preload_libraries = 'pg_stat_statements'
```

query	shared_blks_hit
calls	shared_blks_read
total_time	shared_blks_dirtied
min_time	shared_blks_written
max_time	local_blks_hit
mean_time	local_blks_read
stddev_time	local_blks_dirtied
rows	local_blks_written
	blk_read_time
	blk_write_time

Slowest queries

```
SELECT
interval '1 millisecond' * total_time AS
total_exec_time,

to_char((total_time/sum(total_time) OVER()) * 100,
'FM90D0') || '%'  AS prop_exec_time,

to_char(calls, 'FM999G999G999G990') AS ncalls,
interval '1 millisecond' * (blk_read_time +
blk_write_time) AS sync_io_time,

query

FROM pg_stat_statements
ORDER BY total_time DESC LIMIT 10;
```

Slowest queries

```
SELECT
interval '1 millisecond' * total_time AS
total_exec_time,

to_char((total_time/sum(total_time) OVER()) * 100,
'FM90D0') || '%'  AS prop_exec_time,

to_char(calls, 'FM999G999G999G990') AS ncalls,
interval '1 millisecond' * (blk_read_time +
blk_write_time) AS sync_io_time,

query

FROM pg_stat_statements
ORDER BY total_time DESC LIMIT 10;
```

LOCK TYPE	USED BY
Access Share Lock	SELECT
Row Share Lock	SELECT FOR UPDATE
Row Exclusive Lock	INSERT, UPDATE, DELETE
Share Update Exclusive Lock	CREATE INDEX CONCURRENTLY, VACUUM,
Share Lock	CREATE INDEX
Share Row Exclusive Lock	ALTER TABLE
Exclusive Lock	REFRESH MATERIALIZED VIEW CONCURRENTLY
Access Exclusive Lock	DROP TABLE, REINDEX, CLUSTER…

\d pg_stat_activity;

datid

datname

pid

usesysid

usename

application_name

client_addr

client_hostname

client_port

backend_start

xact_start

query_start

state_change

wait_event_type

wait_event

state

backend_xid

backend_xmin

query

Locks

```
SELECT
    pg_stat_activity.pid,
    pg_class.relname,
    pg_locks.transactionid,
    pg_locks.granted,
    pg_stat_activity.query,
    age(now(),pg_stat_activity.query_start) AS "age"
FROM pg_stat_activity,pg_locks
LEFT OUTER JOIN pg_class
    ON (pg_locks.relation = pg_class.oid)
WHERE pg_locks.pid = pg_stat_activity.pid
-- AND pg_locks.mode = 'ExclusiveLock'
AND pg_stat_activity.pid <> pg_backend_pid() order by
query_start;
```

Locks

```
SELECT
    pg_stat_activity.pid,
    pg_class.relname,
    pg_locks.transactionid,
    pg_locks.granted,
    pg_stat_activity.query,
    age(now(),pg_stat_activity.query_start) AS "age"
FROM pg_stat_activity,pg_locks
LEFT OUTER JOIN pg_class
    ON (pg_locks.relation = pg_class.oid)
WHERE pg_locks.pid = pg_stat_activity.pid
-- AND pg_locks.mode = 'ExclusiveLock'
AND pg_stat_activity.pid <> pg_backend_pid() order by
query_start;
```

Blocking queries

```
SELECT bl.pid AS blocked_pid,
    ka.query AS blocking_statement,
    now() - ka.query_start AS blocking_duration,
    kl.pid AS blocking_pid,
    a.query AS blocked_statement,
    now() - a.query_start AS blocked_duration
FROM pg_catalog.pg_locks bl
JOIN pg_catalog.pg_stat_activity a
    ON bl.pid = a.pid
JOIN pg_catalog.pg_locks kl
    JOIN pg_catalog.pg_stat_activity ka
        ON kl.pid = ka.pid
ON bl.transactionid = kl.transactionid AND bl.pid != kl.pid
WHERE NOT bl.granted
```

Blocking queries

```
SELECT bl.pid AS blocked_pid,
    ka.query AS blocking_statement,
    now() - ka.query_start AS blocking_duration,
    kl.pid AS blocking_pid,
    a.query AS blocked_statement,
    now() - a.query_start AS blocked_duration
FROM pg_catalog.pg_locks bl
JOIN pg_catalog.pg_stat_activity a
    ON bl.pid = a.pid
JOIN pg_catalog.pg_locks kl
    JOIN pg_catalog.pg_stat_activity ka
        ON kl.pid = ka.pid
ON bl.transactionid = kl.transactionid AND bl.pid != kl.pid
WHERE NOT bl.granted
```

Index and table hit rates

```
SELECT
  'index hit rate' AS name,
  (sum(idx_blks_hit)) / nullif(sum(idx_blks_hit +
idx_blks_read),0) AS ratio
FROM pg_statio_user_indexes

UNION ALL

SELECT
 'table hit rate' AS name,
  sum(heap_blks_hit) / nullif(sum(heap_blks_hit) +
sum(heap_blks_read),0) AS ratio
FROM pg_statio_user_tables;
```

Index and table hit rates

```
SELECT
  'index hit rate' AS name,
  (sum(idx_blks_hit)) / nullif(sum(idx_blks_hit +
idx_blks_read),0) AS ratio
FROM pg_statio_user_indexes

UNION ALL

SELECT
 'table hit rate' AS name,
  sum(heap_blks_hit) / nullif(sum(heap_blks_hit) +
sum(heap_blks_read),0) AS ratio
FROM pg_statio_user_tables;
```

Index and table hit rates

```
SELECT
  'index hit rate' AS name,
  (sum(idx_blks_hit)) / nullif(sum(idx_blks_hit +
idx_blks_read),0) AS ratio
FROM pg_statio_user_indexes

UNION ALL

SELECT
 'table hit rate' AS name,
  sum(heap_blks_hit) / nullif(sum(heap_blks_hit) +
sum(heap_blks_read),0) AS ratio
FROM pg_statio_user_tables;
```

Vacuuming

- Keep track of how efficiently your autovacuum processes are cleaning up bloat
    - When last autovacuum/last vacuum were run on a table, and compare against proportion of dead rows in a table

Vacuum stats

```
SELECT
  pg_class.relname,
  psut.last_vacuum AS last_vacuum,
  psut.last_autovacuum AS last_autovacuum,
  to_char(pg_class.reltuples, '9G999G999G999') AS rowcount,
  to_char(psut.n_dead_tup, '9G999G999G999') AS
dead_rowcount,
FROM
  pg_stat_user_tables psut INNER JOIN pg_class ON psut.reli
= pg_class.oid
    INNER JOIN vacuum_settings ON pg_class.oid =
vacuum_settings.oid
ORDER BY 1
```

Vacuum stats

```
SELECT
  pg_class.relname,
  psut.last_vacuum AS last_vacuum,
  psut.last_autovacuum AS last_autovacuum,
  to_char(pg_class.reltuples, '9G999G999G999') AS rowcount,
  to_char(psut.n_dead_tup, '9G999G999G999') AS
dead_rowcount,
FROM
  pg_stat_user_tables psut INNER JOIN pg_class ON psut.reli
= pg_class.oid
    INNER JOIN vacuum_settings ON pg_class.oid =
vacuum_settings.oid
ORDER BY 1
```

Vacuuming

- Keep track of how efficiently your autovacuum processes are cleaning up bloat
  - When last autovacuum/last vacuum were run on a table, and compare against proportion of dead rows in a table
- **Autovacuums to prevent transaction ID wraparounds are particularly important.**

Table transaction id age

- 47

```
select relname, age(relfrozenxid) from pg_class where
relname = 'conferences';
   relname    |  age
--------------+--------
 conferences | 200288

select setting::numeric as max_age from pg_settings where
name = 'autovacuum_freeze_max_age';
  max_age
-----------
 200000000
```

Table transaction id age

```
 WARNING:  database "db123456" must be vacuumed within
143001235 transactions
 HINT:  To avoid a database shutdown, execute a
database-wide VACUUM in "db123456".
```

Vacuuming

- Keep track of how efficiently your autovacuum processes are cleaning up bloat
    - When last autovacuum/last vacuum were run on a table, and compare against proportion of dead rows in a table
- Autovacuums to prevent transaction ID wraparounds are particularly important.
- **Additionally, auto vacuum runs AUTOANALYZE which helps gather accurate statistics for your query plans**
    - Query plans will only be as good as PostgreSQL's statistics on your data

Other monitoring

- Track CPU and disk statistics on your server.
- Keep track of database and index bloat.
- Monitor number of connections to your database server.
- Monitor for service interruptions

Monitoring

- Finding slow queries through pg_stat_statements
- Monitoring locks and blocked queries through pg_stat_activity and pg_locks
- Monitoring bloat, vacuuming and cache hit rates
- UNIX monitoring

Troubleshooting performance issues in PostgreSQL

- Query plans
- Query execution
- At scale
- Other performance factors (PostgreSQL version, hardware) and how to benchmark them
- What to monitor

Credits

- PostgreSQL documentation (https://www.postgresql.org/docs/9.6/static/) and wiki (https://wiki.postgresql.org/wiki/Main_Page)
- Help scripts from https://github.com/pgexperts/pgx_scripts and https://github.com/heroku/heroku-pg-extras
- Blog post by Keith Fiske on shared_buffers: https://www.keithf4.com/a-large-database-does-not-mean-large-shared_buffers/
- Bruce Momjian's database hardware selection guidelines, https://momjian.us/main/writings/pgsql/hw_selection.pdf.
- https://blog.pgaddict.com/posts/performance-since-postgresql-7-4-to-9-4-pgbench.

Thank you!

Please remember to complete your tutorial evaluation

https://camillebaldock.com/postgres
camille@camillebaldock.com
@camille_