PostgreSQL tuning for Oracle DBAs

About me

Hervé Schweitzer

CTO

Principal consultant

+41 79 963 43 67

herve.schweitzer[at]dbi-services.com

ORACLE

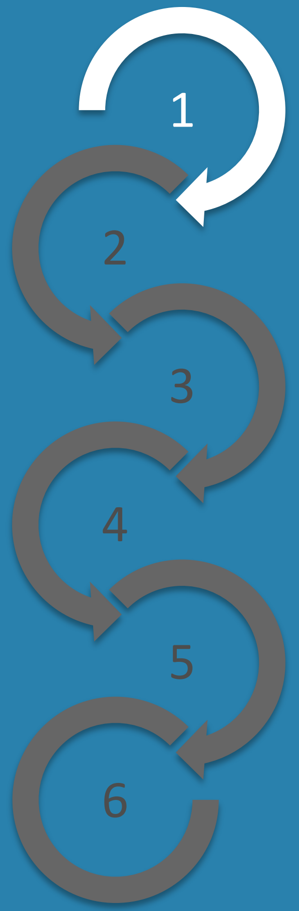
Certified Master

Oracle Database 11*g* Administrator

Agenda

1. My story
2. Some tips
3. Database optimizer
4. Object statistics
5. Execution plan
6. Conclusion

My story



My story

1997 – 1999	Linux Admin/Adabas DBA
1999 – 2003	Oracle DBA (Mainly Database performance Tuning)
2003 – 2010	Oracle Senior Consultant (HA-Tuning)
2010 – 2018	CTO – Oracle Consultant (HA-Tuning-GoldenGate) – Oracle OCM
2018 – Today	CTO – Oracle Consultant / PostgreSQL performance Tuning

Why the switch ?

> The PostgreSQL database is part of our daily business today

> Both RDBMS have many similarities (Linux based, cmdline with scripts)

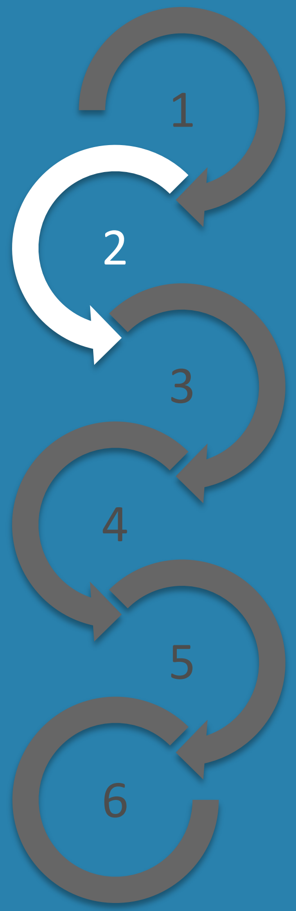
> Beta and development releases are available without any restriction to test future features

> Can be directly implemented at customers without any license issues

> You can implement what you recently learned ☺

Some tips

- Prompt
- MacOS user



1

2

3

4

5

6

Some tips

Prompt

Default PostgreSQL prompt

> Is terminating with "**#**"

```
postgres@dbi-pg :/home/postgres/ [PG11] psql -U postgres postgres
postgres=#
```

> Hashtag ”**#**" prompt can be confuse, because is also the default Linux ROOT prompt

```
[root@dbi-pg ~]#
```

> Therefore I decided to change it, to an Oracle like prompt to begin with Postgres ☺

```
postgres@dbi-pg :/home/postgres/ [PG11] psql -U postgres postgres
postgres PSQL> \c test
You are now connected to database "test" as user "postgres".
test PSQL>
test PSQL>  first  line of multiline code
(test PSQL> second line of multiline code
(test PSQL> ;
```

Some tips

Prompt

How to change the prompt of the Postgres Linux user

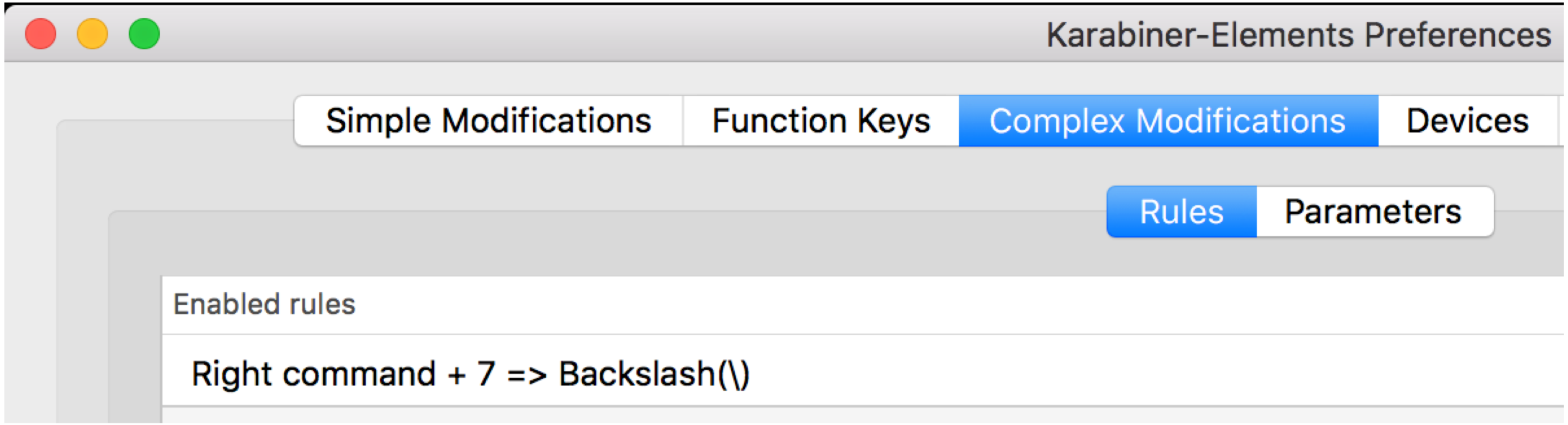
- PROMPT1
  - Single line code ended by ";"
- PROMPT2
  - Multi line code ended by ";"

```
postgres@dbi-pg :/home/postgres/ [PG111] cat .psqlrc
. . .
\set PROMPT1 '%/ PSQL>'
\set PROMPT2 '(%/ PSQL>'
```

Some tips
MacOS user

Backslash on MacOS Without always entering a 3 key combination : Alt + Maj + /

> Install Karabiner-Elements and configure another key combination



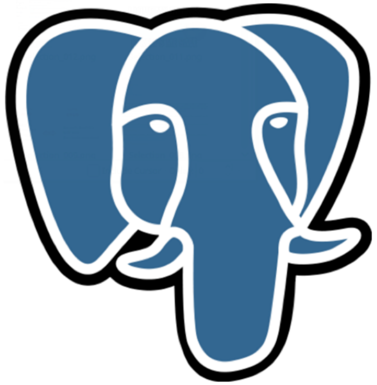
Enabled rules

Right command + 7 => Backslash(\)

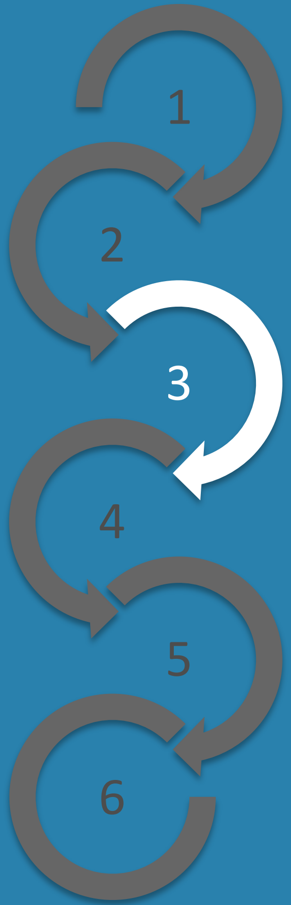
Why this combination

> Because slash is "Shift + 7"

Database optimizer



- > Oracle vs PostgreSQL terminologies
- > Optimizer flow
- > Parsing
- > Planning
- > Executions
- > Optimizer parameter
- > The cost model



Database optimizer

Oracle vs PostgreSQL terminologies

Optimizer

- Transforms the statement
- Generate different execution plans
- Evaluates costs for all operation to get costs for several execution plans
- Choose execution plan with the best (lowest) cost

Oracle and PostgreSQL optimizer are working the same way

Object Statistics

- Required for the optimizer to generate the best access plan with the lowest cost
- Object statistics collect different information
- Oracle and PostgreSQL collect also histograms to identify the content of one columns (e.g. skewed data)

Oracle and PostgreSQL Statistics are working the same way

Database optimizer

Oracle vs PostgreSQL terminologies

Buffer Cache

- Oracle **buffer cache**
  - All data blocks are saved into the database buffer cache

With Oracle the memory will be mainly managed from the database

- PostgreSQL **Shared buffer cache**
  - Less blocks are cached, all other data are cache on the OS level (filesystem cache)

With PostgreSQL the memory will be mainly managed from the OS

Database optimizer
Oracle vs PostgreSQL terminologies

Shared Pool

- Oracle Shared Pool
  - All dictionary information, executions plans, running information will be cached there

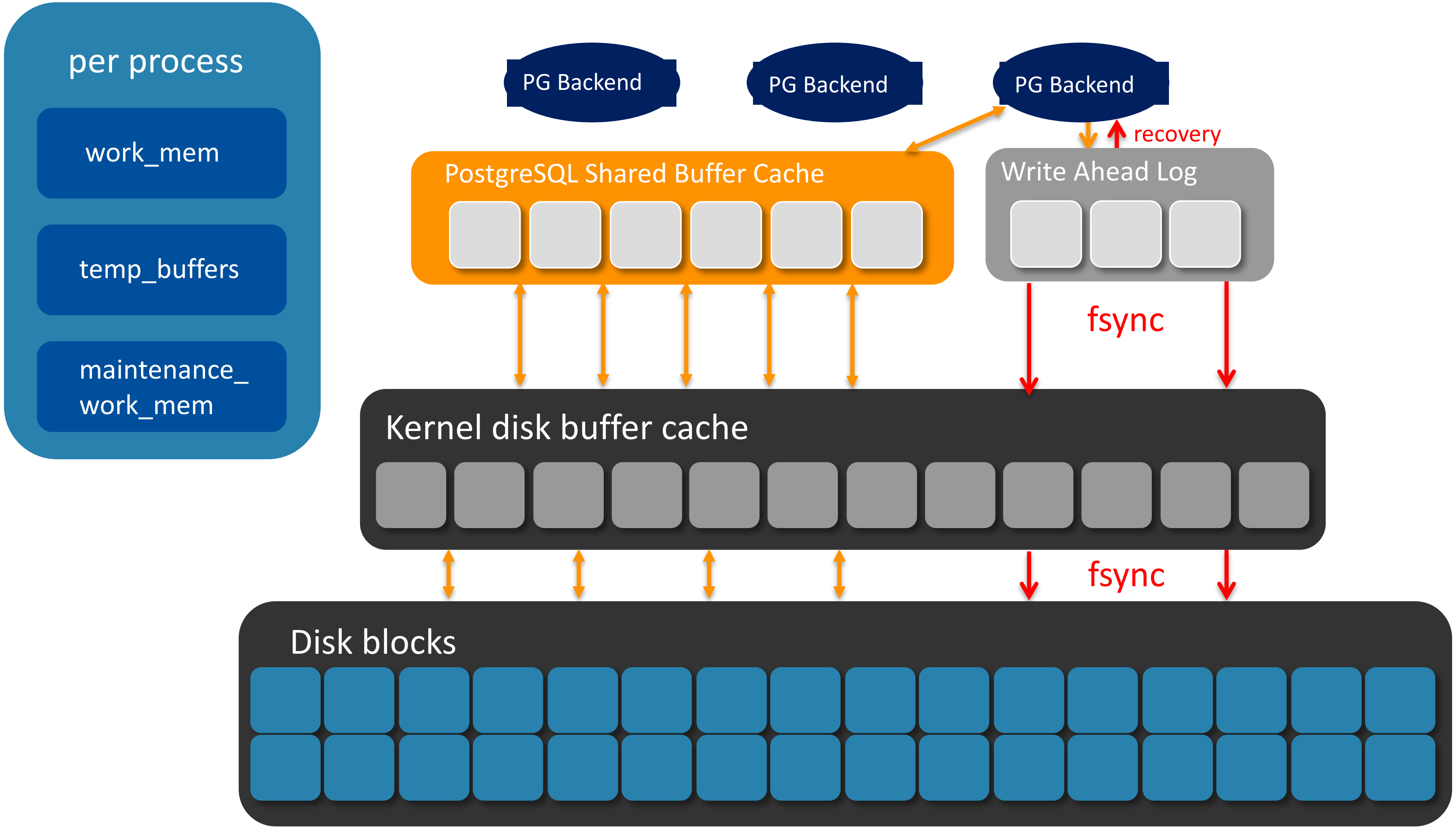
Oracle shared memory is available for existing and new sessions

- PostgreSQL does not have any Shared Pool for the moment
  - Session information is only cached in the session it self, nothing is shared cross-session

No Shared Pool exist for PostgreSQL

Database optimizer

Oracle vs PostgreSQL terminologies



Database optimizer

Oracle vs PostgreSQL terminologies

Parsing `(log_parser_stats)`

> Check the syntax and semantic

> Check access rights

> PostgreSQL also rewrite the SQL and format it into a raw tree format

> With a PREPARE statement this step occurs once

PostgreSQL does a little bit more during parsing time

Optimizing/Planning `(log_planner_stats)`

> Step where the best plan will be generated based on the object statistics

> For Oracle, the rewrite of the SQL is done here

> This step is the **Hard Parsing time for Oracle**, what not always occurs if available into the SharedPool

> With a **Postgres PREPARE** statement after 5 executions **it will sometimes also bypassed**

Oracle does not always need this step(HardParse), if the cursor is still available on the SharedPool

Database optimizer

Oracle vs PostgreSQL terminologies

Executing `(log_executor_stats)`

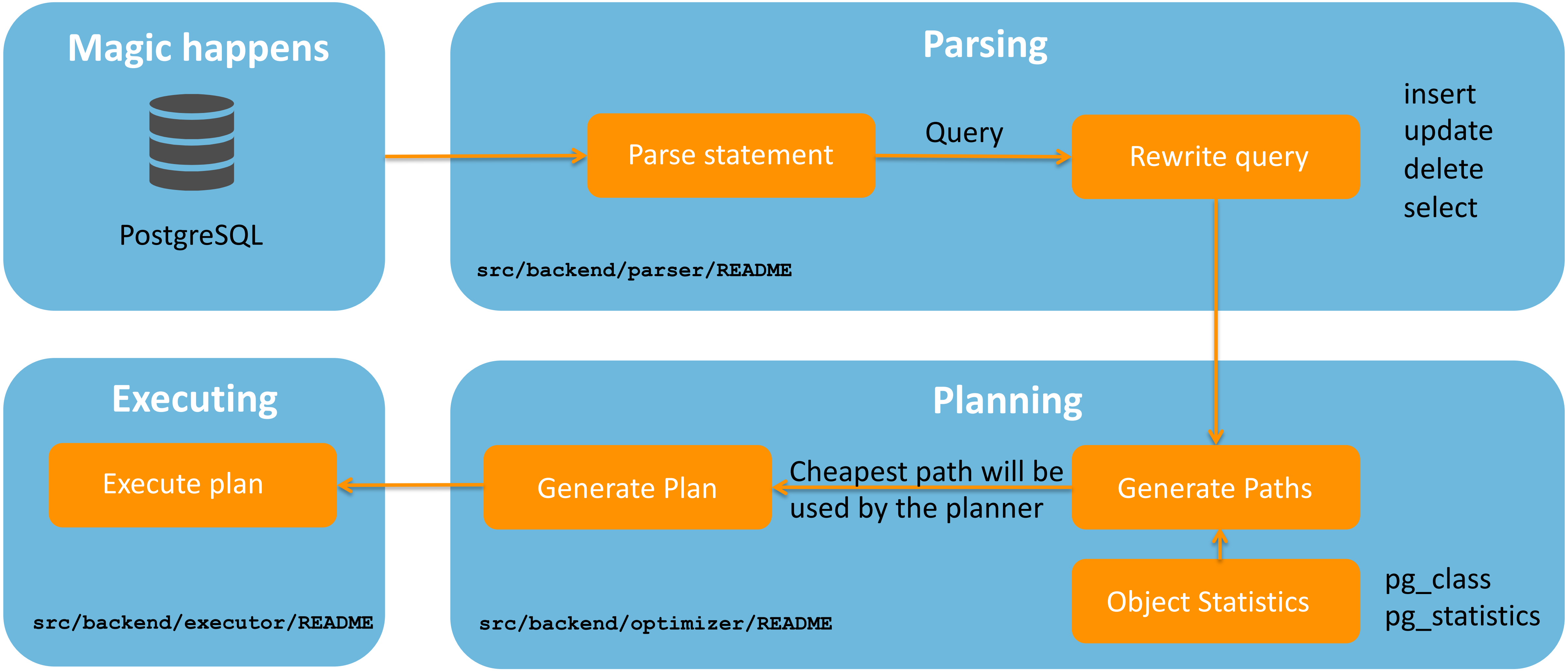
> Executions of the SQL based of the execution plan generated

> During execution the data will be fetched back to the client

Oracle and PostgreSQL executions are working the same way

Database optimizer

Optimizer flow



Magic happens

PostgreSQL

Parsing

Parse statement

Query

Rewrite query

insert
update
delete
select

src/backend/parser/README

Planning

Generate Paths

Object Statistics

pg_class
pg_statistics

Cheapest path will be used by the planner

Generate Plan

src/backend/optimizer/README

Executing

Execute plan

src/backend/executor/README

Database optimizer
Parsing

Display the parsing time of an SQL statement

> System level

```
postgres PSQL> alter system set log_parser_stats=true;
postgres PSQL> select pg_reload_conf();
```

> Session level

```
postgres PSQL> set log_parser_stats=true;
postgres PSQL> select 1;
```

> User level

```
postgres PSQL> alter user HR set log_parser_stats=true;
```

> Output into logfile postgresql.log

```
2018-09-24 22:20:40.887 CEST - 61 - 15900 - [local] - postgres@postgres LOG:  PARSER STATISTICS
! 0.000004 s user, 0.000019 s system, 0.000021 s elapsed

2018-09-24 22:20:40.887 CEST - 64 - 15900 - [local] - postgres@postgres LOG:  PARSE ANALYSIS STATISTICS
! 0.000003 s user, 0.000013 s system, 0.000016 s elapsed

2018-09-24 22:20:40.887 CEST - 67 - 15900 - [local] - postgres@postgres LOG:  REWRITER STATISTICS
! 0.000000 s user, 0.000002 s system, 0.000002 s elapsed
```

Database optimizer

Planning

Display the planner time of an SQL statement

> System level

```
postgres PSQL> alter system set log_planner_stats=true;
postgres PSQL> select pg_reload_conf();
postgres PSQL> select 1;
```

> Session level

```
postgres PSQL> set log_planner_stats=true;
postgres PSQL> select 1;
```

> User level

```
postgres PSQL> alter user HR set log_planner_stats=true;
```

> Output into logfile postgresql.log

```
2018-09-24 22:33:57.789 CEST - 2 - 16055 - [local] - postgres@postgres LOG:  PLANNER STATISTICS
! 0.000018 s user, 0.000007 s system, 0.000025 s elapsed
2018-09-24 22:33:57.789 CEST - 4 - 16055 - [local] - postgres@postgres STATEMENT:  select 1;
```

Database optimizer

Executions

Display the executor time of an SQL statement

> System level

```
postgres PSQL> alter system set log_executor_stats=true;
postgres PSQL> select pg_reload_conf();
postgres PSQL> select 1;
```

> Session level

```
postgres PSQL> set log_executor_stats=true;
postgres PSQL> select 1;
```

> User level

```
postgres PSQL> alter user HR set log_executor_stats=true;
```

> Output into logfile postgresql.log

```
2018-01-04 12:02:11.202 CET [7832] STATEMENT:  select 1;
2018-01-04 12:02:11.220 CET [2119] LOG:  EXECUTOR STATISTICS
2018-01-04 12:02:11.220 CET [2119] DETAIL:  ! system usage stats:
    !    0.000025 s user, 0.000000 s system, 0.000024 s elapsed
```

Database optimizer
Optimizer parameter

There are several parameters to control the optimizer's choice to access the data

```
postgres PSQL> show enable_[TAB_TAB]
enable_bitmapscan     enable_hashagg        enable_indexonlyscan  enable_material
enable_nestloop       enable_sort           enable_gathermerge    enable_hashjoin
enable_indexscan      enable_mergejoin      enable_seqscan        enable_tidscan
postgres PSQL> set enable_hashagg=off;
```

It is not advisable to change these optimizer parameters

- They fake the optimizer estimations
  - In fact they massively increase the cost, when turned off
- They are there only for exceptions (bugs)
- When it is really required
  - Set a parameter on the session level?
  - Set a parameter globally?

Database optimizer

The cost model

There are several parameters that control cost calculations

Parameter	Description	Default Value
seq_page_cost	The cost of one (sequential) page fetch from disk	1
random_page_cost	The cost of one random page fetch from disk	4
cpu_tuple_cost	The cost of processing each row	0.01
cpu_index_tuple_cost	The cost of processing each index entry	0.005
cpu_operator_cost	The cost of processing each operator or function	0.0025

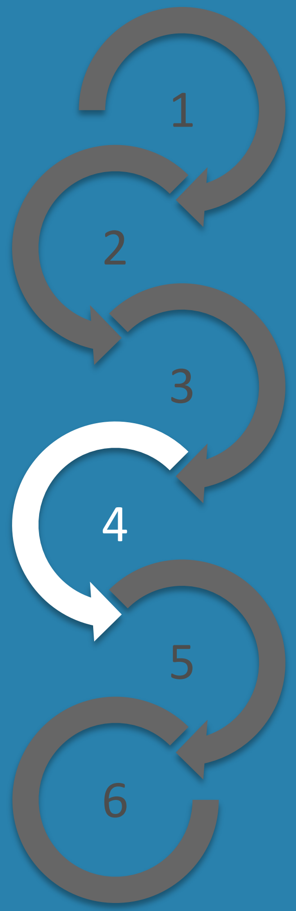
> Everything is relative to seq_page_cost

> When you are on SSDs: Is the cost of a random scan still 4 times as expensive as a sequential scan?

> Try to tune the random_page_cost parameter

Object statistics

- Statistics overview
- pg_class
- pg_stats
- Gathering object statistics



Object statistics
Statistics overview

Used to provide statical information about the data in a relation

- Numbers of rows
- Numbers of blocks
- Numbers of distinct values/nulls for a column
- The average rows width
- The most common values and their frequency
- Histogram bounds

Use catalog tables and views to get object statistics

- pg_class
- pg_stats

Object statistics

pg_class

To check statistics on the table level

```
pgbench PSQL> select relname,relpages,reltuples::int, reltuples/relpages avgtuple
      from pg_class
      where relname = 'pgbench_accounts';

┌──────────────────┬──────────┬───────────┬──────────────────┐
│     relname      │ relpages │ reltuples │     avgtuple     │
├──────────────────┼──────────┼───────────┼──────────────────┤
│ pgbench_accounts │   163935 │  10000000 │ 60.9997865007472 │
└──────────────────┴──────────┴───────────┴──────────────────┘
```

- > relpages : Number of 8K block
- > reltuples : Number of rows
- > avgtupl : Number of rows per block

Object statistics
pg_stats

To check statistics on a column level

```
pgbench PSQL> select tablename,attname,null_frac,avg_width,n_distinct
              from pg_stats
              where tablename = 'pgbench_accounts';

┌──────────────────┬──────────┬───────────┬───────────┬────────────┐
│    tablename     │ attname  │ null_frac │ avg_width │ n_distinct │
├──────────────────┼──────────┼───────────┼───────────┼────────────┤
│ pgbench_accounts │ aid      │         0 │         4 │         -1 │
│ pgbench_accounts │ bid      │         0 │         4 │        100 │
│ pgbench_accounts │ abalance │         0 │         4 │          1 │
│ pgbench_accounts │ filler   │         0 │        85 │          1 │
└──────────────────┴──────────┴───────────┴───────────┴────────────┘
```

> attname : Column name
> null_frac : Fraction of column entries that are null
> avg_width : Average width in bytes of column's entries
> n_distinct : Number of distinct values (but negative values can exist, Ex. -1 indicates a unique column)

Object statistics

pg_stats - Histograms

Most common values and their frequency per column

```
pgbench PSQL> select most_common_vals,most_common_freqs
              from pg_stats where tablename = 'pgbench_accounts' and attname = 'bid';

-[ RECORD 1 ]--
most_common_vals  | {21,68,88,25,14,53,58,61,7,44,35…
most_common_freqs | {0.0119667,0.0117667,0.0115,0.0114333,0.0114,0.0113333…
```

> The value of 21 has a frequency of 0.0119667

> The value of 68 has a frequency of 0.0117667

> The formula to calculate the frequency: count(value)/total rows

Object statistics

pg_stats - Histograms

histogram_bounds

```
pgbench PSQL> select histogram_bounds
              from pg_stats where tablename = 'pgbench_accounts' and attname = 'aid';

-[ RECORD 1 ]--
histogram_bounds | {12,103238,213931,305537,410681,503952,610274,703390,801506,918762 …
```

- These are groups of approximately the same number of values
  - 103238-12 = 103226
  - 213931-103238 = 110693
  - 305537-213931 = 91606
  - ...
- The values in most_common_vals, if present, are omitted from this histogram calculation
- When the column data type does not have a "<" operator this column is null

Object statistics

pg_stats - Example

histogram_bounds - example

```
pgbench PSQL> select a,count(*) from t1 group by a order by count(*);
  a   | count
------+-------
    1 |     1
    2 |     1
    3 |  1000
    4 |  2000
    5 |  2000

pgbench PSQL> select histogram_bounds from pg_stats where tablename = 't1';
 histogram_bounds
------------------
 {1,2}

pgbench PSQL> select most_common_vals,most_common_freqs from pg_stats where tablename = 't1';
 most_common_vals |      most_common_freqs
------------------+---------------------------
 {4,5,3}          | {0.39984,0.39984,0.19992}
```

Object statistics

Gathering object statistics

The formula when autovacuum kicks in to gather statistics

```
vacuum threshold =   autovacuum analyze threshold
                   + autovacuum analyze scale factor
                   * pg_class.reltuples
```

The default configuration

```
pgbench PSQL> select name,setting from pg_settings
              where name in ('autovacuum_analyze_threshold'
                            ,'autovacuum_analyze_scale_factor');

              name               | setting
---------------------------------+---------
 autovacuum_analyze_scale_factor | 0.1
 autovacuum_analyze_threshold    | 50
(2 rows)
```

Object statistics

Gathering object statistics

Checking for the last (auto)analyze and (auto)vacuum

```
pgbench PSQL> \x
Expanded display is on.

pgbench PSQL>  select last_vacuum,last_autovacuum,last_analyze,last_autoanalyze
               from pg_stat_all_tables where relname = 'pgbench_accounts';

-[ RECORD 1 ]----+------------------------------
last_vacuum      | 2019-03-14 01:13:41.070397+00
last_autovacuum  | NULL
last_analyze     | 2019-03-14 01:13:45.482932+00
last_autoanalyze | NULL
```

Object statistics

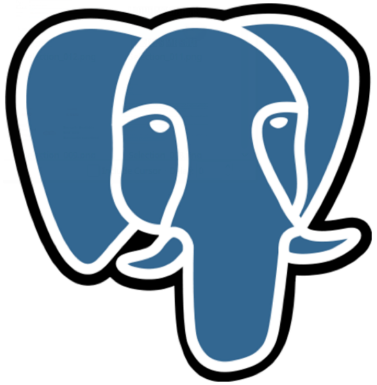
Gathering object statistics

To manually gather statistics

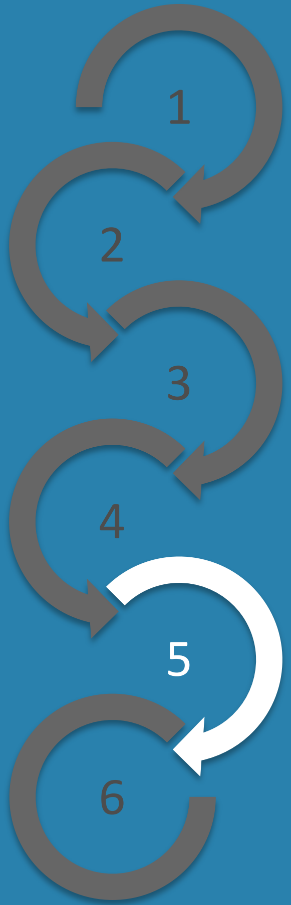
> Analyze can operate on the table or on the column level

```
pgbench PSQL> analyze pgbench_accounts;
ANALYZE
pgbench PSQL> analyze pgbench_accounts(aid);
ANALYZE
pgbench PSQL> \h analyze
Command:     ANALYZE
Description: collect statistics about a database
Syntax:
ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

Execution plan



- EXPLAIN command
- PREPARE statements
- Skewed data distribution



Execution plan

EXPLAIN command

EXPLAIN is the tool to display execution plan and various statistics

> explain is ready to use by default Inside psql

```
pgbench PSQL> \h explain  -- help page of all explain commands
pgbench PSQL> explain select * from t1 where a=1;

                              QUERY PLAN
----------------------------------------------------------------------
 Index Only Scan using index1 on t1  (cost=0.28..8.30 rows=1 width=4)
   Index Cond: (a = 1)
```

> explain with the analyze parameter will execute the statement

```
pgbench PSQL> explain analyze select * from t1 where a=1;

                              QUERY PLAN
------------------------------------------------------------------------------------------------------
 Index Only Scan using index1 on t1  (cost=0.28..8.30 rows=1 width=4) (rows=1)
   Index Cond: (a = 1)
   Heap Fetches: 1
 Planning Time: 0.092 ms
 Execution Time: 0.123 ms
```

Execution plan

PREPARE command

The PREPARE command allows the usage of bind variables

When the same statement is executed over and over again

> Prepare the statement so it is parsed, analyzed, and rewritten only once

> Execution of a prepared statement only requires planning and execution

> Prepared statements only live in the session and are gone once the session ends

How to prepare a statement

```
Pgbench PSQL> prepare my_stmt as select * from t1 where a = $1;
Pgbench PSQL> execute my_stmt ('1');
```

To remove a prepared statement

```
Pgbench PSQL> deallocate my_stmt;
```

Execution plan

Skewed data distribution

When data is unregularly distributed

```
pgbench PSQL> select a,count(*) from skewed_data group by a;
 a  |  count
----+---------
  1 |   99990
  2 |       1
  3 |       1
  4 |       1
  5 |       1
  6 |       1
  7 |       1
  8 |       1
  9 |       1
 10 |       1
```

The optimizer should

> For a=1 do a Seq Scan on table skewed_data (FULL TABLE SCAN)

> For all other values of a, it should use an index scan on (a)

Execution plan
Skewed data distribution

With litterals it works perfectly because planning time will be executed for each values

```
pgbench PSQL> explain select * from skewed_data where a = 1;

                             QUERY PLAN
---------------------------------------------------------------------
 Seq Scan on skewed_data  (cost=0.00..29167.00 rows=99992 width=37)
   Filter: (a = 1)
```

```
pgbench PSQL> explain select * from skewed_data where a = 2;

                                QUERY PLAN
--------------------------------------------------------------------------
 Index Only Scan using i1 on skewed_data  (cost=0.42..4.44 rows=1 width=4)
   Index Cond: (a = 2)
```

Execution plan
Skewed data distribution

What about prepared statements, the same :-)

```
pgbench PSQL> prepare my_stmt as select * from skewed_data where a = $1;
```

```
pgbench PSQL> explain analyze execute my_stmt ('1');

                QUERY PLAN
--------------------------------------------------------------------------------------------------
Seq Scan on skewed_data  (cost=0.00..29167.00 rows=99990 width=6)  (rows=99991)
   Filter: (a = 1)
   Rows Removed by Filter: 9
 Planning time: 0.135 ms
 Execution time: 217.040 ms
```

```
pgbench PSQL> explain analyze execute my_stmt ('2');

                QUERY PLAN
--------------------------------------------------------------------------------------------------
Index Only Scan using i1 on skewed_data  (cost=0.42..8.48 rows=3 width=6)  (rows=1)
   Index Cond: (a = 2)
 Planning time: 0.155 ms
 Execution time: 0.083 ms
```

Execution plan

Skewed data distribution

Take care : Generic plans with prepared statements

- Usually a prepared statement is re-planned with every execution
- But after 5 executions when the costs (including planning overhead) is more expensive than a generic plan
  - A generic plan will be used

```
pgbench PSQL> explain analyze execute my_stmt ('1');    -- repeat that 5 times more --

     QUERY PLAN
----------------------------------------------------------------------------------------------
 Seq Scan on skewed_data  (cost=0.00..29167.00        rows=99990 width=37)
                   (actual time=0.014..231.884        rows=99991 loops=1)
   Filter: (a = 1)
```

```
pgbench PSQL> explain analyze execute my_stmt ('1');

      QUERY PLAN
------------------------------------------------------------------------------------------
 Index Scan using i1 on skewed_data  (cost=0.42..11300.93 rows=33333 width=37)
                              (actual time=0.115..355.414 rows=99991 loops=1)
   Index Cond: (a = $1)
```

Execution plan
Skewed data distribution

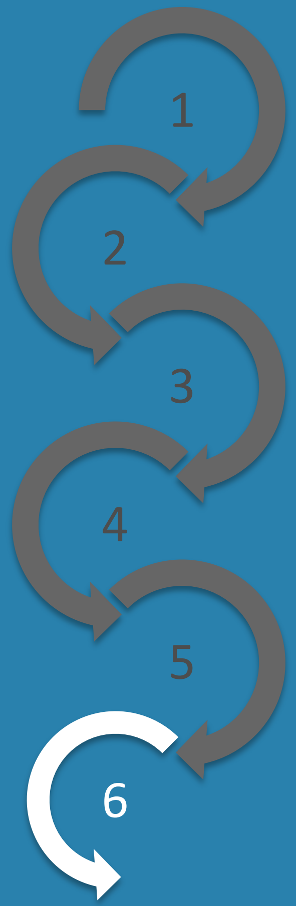
Generic plans

- From now on only the generic plan will be used for the lifetime of the prepared statement
- You can see if a generic or custom plan is used in the explain output
  - $x means this is a generic plan
  - A custom plan will show the actual value(s)

Solution about our issue?

- Don't use PREPARE statement

Conclusion



Conclusion

The most important with Performance Tuning

> To be able to exactly locate the problem

You don’t have to

> Create all kind of table and index types, define each instance parameter

But you have to know

> The available table/index types and how they access data
> What can be configured at instance/session/query level

Your knowledge is strengthened by

> Documentation
> Tests on small testcases
> Experience (but experience is linked to one version and one application)

Conclusion

Which is the main missing performance feature with PostgreSQL ?

It has no Shared Pool, Session information is only cached in the session it self

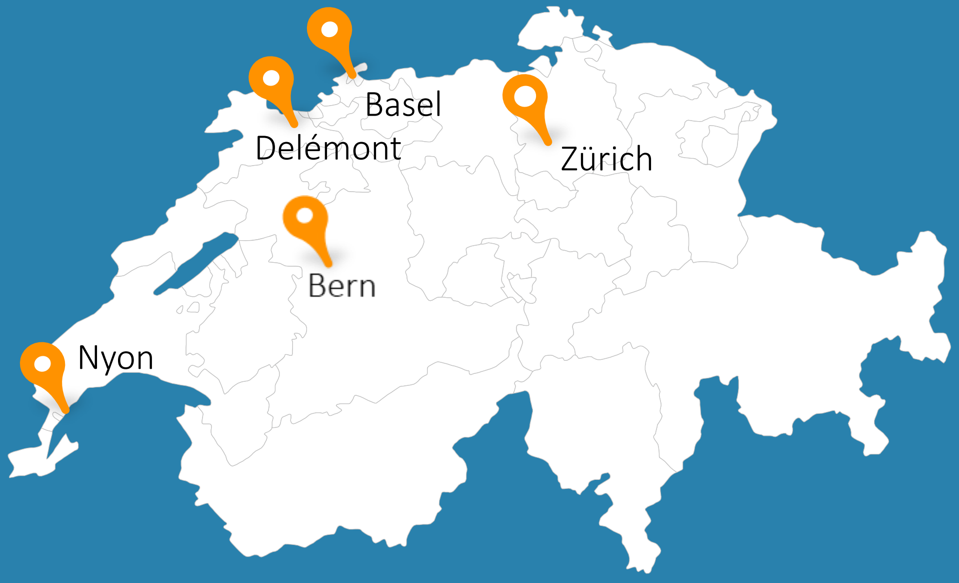
But using a connection pool, with the only required parallel sessions
The chance to have it cached will be high

dbi InSite PostgreSQL Performance Tuning workshop

> 22-23 May in Basel
> 17-18 December in Zürich

Any questions?

Please do ask!



We would love to boost your IT-Infrastructure

How about you?