ONLINE , AROUND-THE-WORLD OPEN SOURCE DATABASE CONFERENCE

PERCONA LIVEONLINE

MAY 12 - 13th

2021

Tuning PostgreSQL for High Performance and Optimization

PostgreSQL High Performance

Ibrar Ahmed

Who am I?

IBRAR AHMED

Senior Software Architect

Percona LLC

@ibrar_ahmad

https://www.facebook.com/ibrar.ahmed

https://www.linkedin.com/in/ibrarahmed74/

https://pgelephant.com/

Software Career

- Software industries since 1998.

PostgreSQL Career

- Working on PostgreSQL Since 2006.
- EnterpriseDB (Associate Software Architect core Database Engine) 2006-2009
- EnterpriseDB (Software Architect core Database Engine) 2011 - 2016
- EnterpriseDB (Senior Software Architect core Database Engine) 2016 – 2018
- Percona (Senior Software Architect core Database Engine) 2018 – Present

PostgreSQL Books

- PostgreSQL Developer's Guide
- PostgreSQL 9.6 High Performance

AGENDA

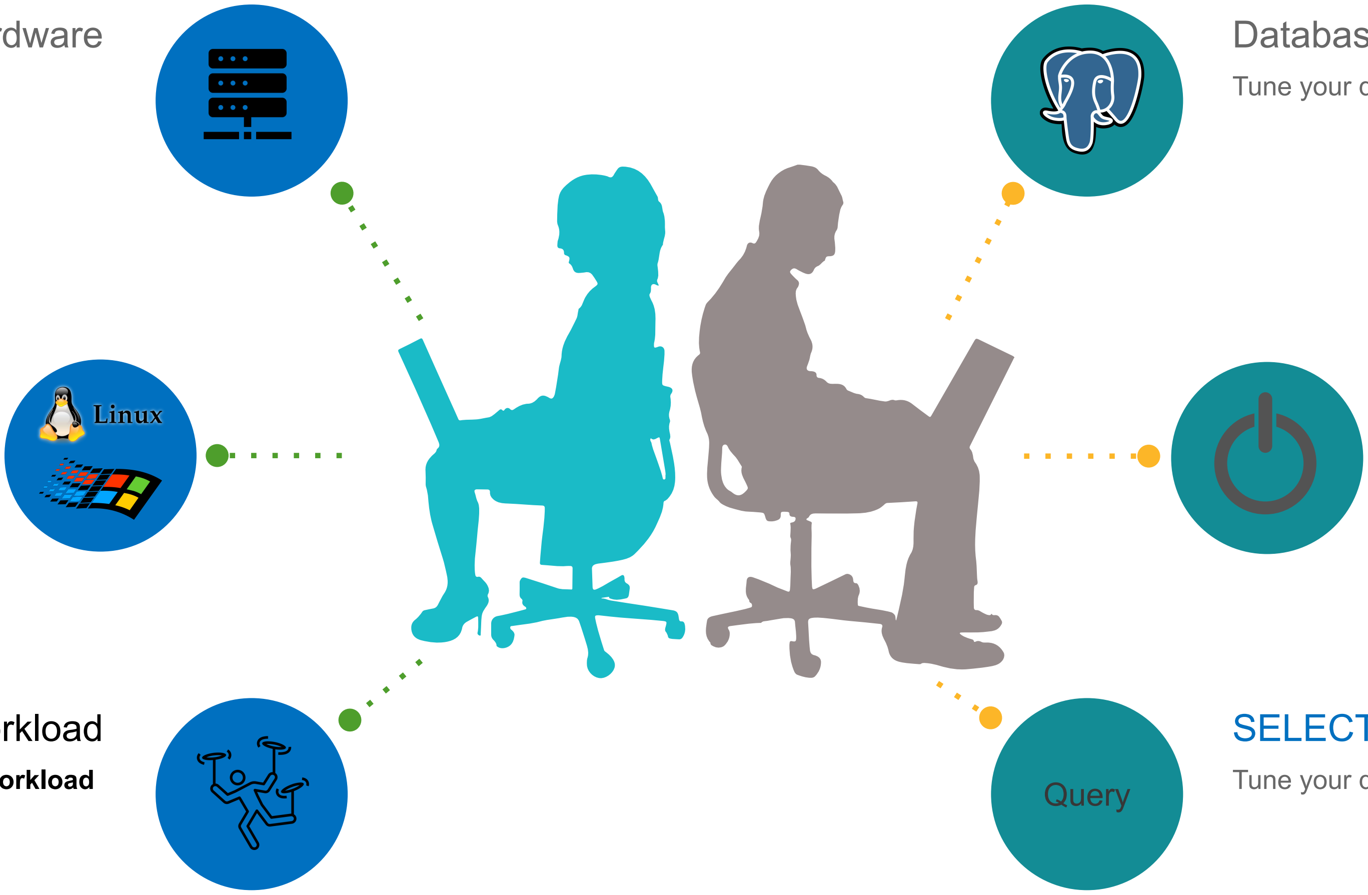
Timeline

5 Minutes	Database Performance
10 Minutes	PostgreSQL Modules
15 Minutes	PostgreSQL Performance Tuning
10 Minutes	Linux Kernel Tuning
5 Minutes	Parallel Query Optimization
5 Minutes	Tips
5 Minutes	Question Answers

1. Database Performance

Overall Database System Performance

Database Performance



Hardware

Choose better **Hardware**

Operating System

Operating System Selection and **Tuning**

Workload

Choose **Database** Depend based on **Workload**

Database

Tune your database (PostgreSQL)

Application

Tune client application

SELECT * FROM foo

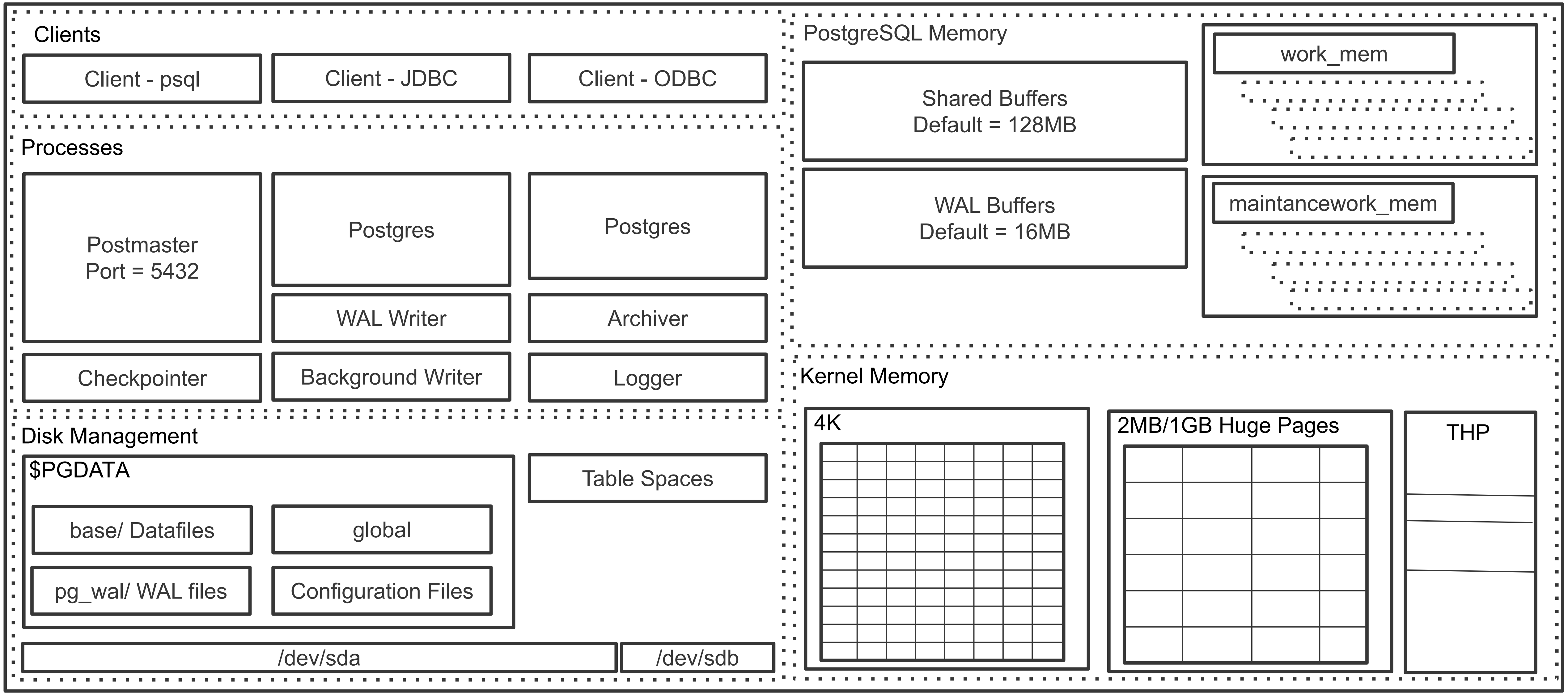
Tune your query properly.

2.

PostgreSQL Modules

PostgreSQL

PostgreSQL Modules



3. PostgreSQL Performance Tuning

PostgreSQL

Tuning Parameters

- shared_buffers
- wal_buffers
- effective_cache_size
- maintenance_work_mem
- synchronous_commit
- checkpoint_timeout
- checkpoint_completion_target
- temp_buffers
- huge_pages

PostgreSQL Tuning / shared_buffers

- PostgreSQL uses its own buffer along with kernel buffered I/O.
- PostgreSQL does not change the information on disk directly then how?
- Writes the data to shared buffer cache.
- The backend process writes that these blocks kernel buffer.

```
postgresql=# SHOW shared_buffers;
shared_buffers
----------------
128MB
(1 row)
```

The proper size for the POSTGRESQL shared buffer cache is the largest useful size that does not adversely affect other activity.

—Bruce Momjian

PostgreSQL Tuning / `max_connections`

- Maximum connection for PostgreSQL
- Default is 100

```
postgres=# SELECT pg_backend_pid();
 pg_backend_pid
----------------
           3214
(1 row)

postgres=# SELECT pg_backend_pid();
 pg_backend_pid
----------------
           3590
(1 row)


postgres=# SELECT pg_backend_pid();
 pg_backend_pid
----------------
           3616
(1 row)
```

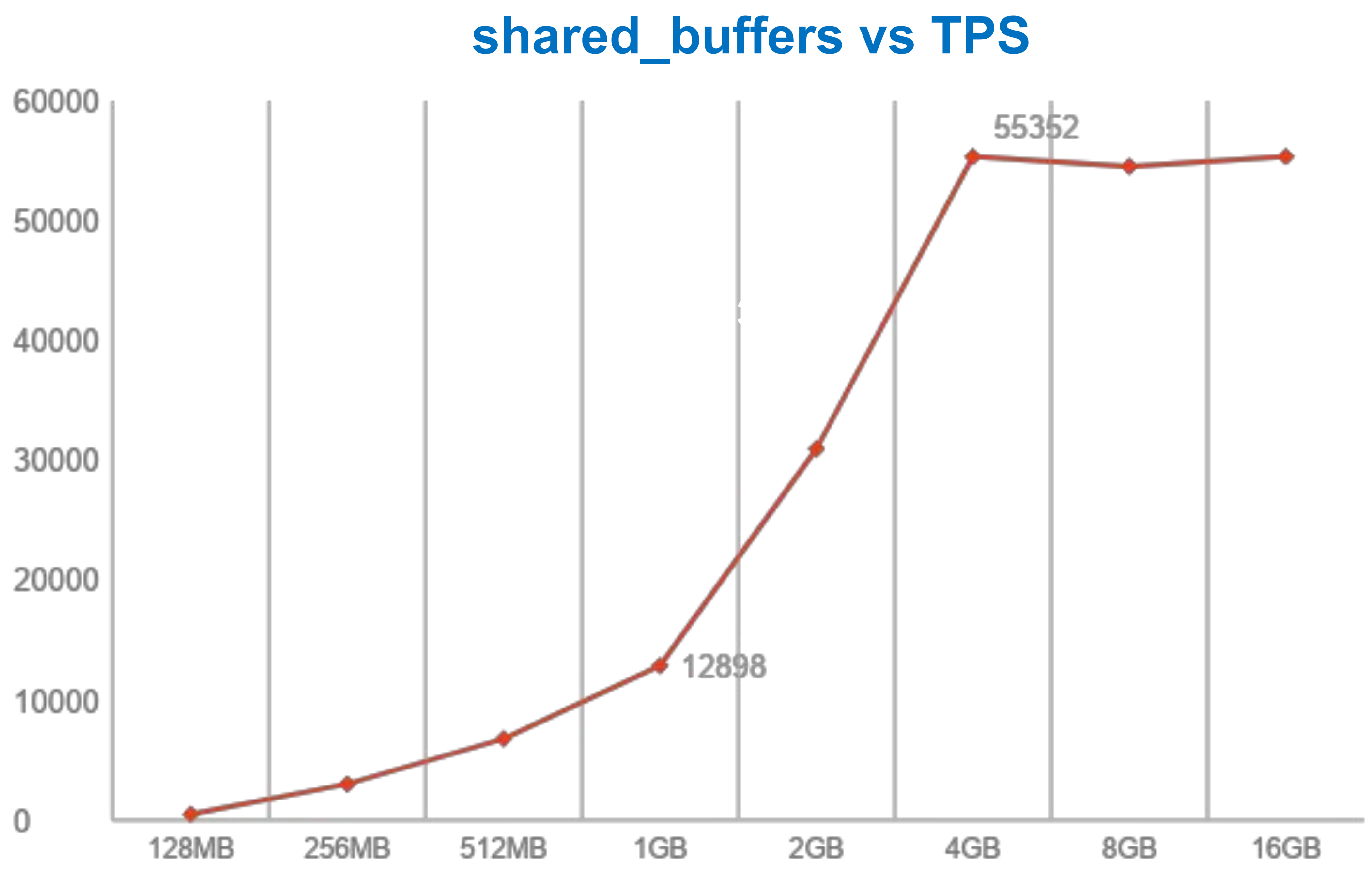
```
ps aux | grep postgres | grep idle
vagrant   3214  0.0  1.2 194060 12948 ?        Ss   15:09   0:00
postgres: vagrant postgres [local] idle

vagrant   3590  0.0  1.2 193840 12936 ?        Ss   15:11   0:00
postgres: vagrant postgres [local] idle

vagrant   3616  0.0  1.2 193840 13072 ?        Ss   15:11   0:00
postgres: vagrant postgres [local] idle
```

If you want too many connections, try using pooler (pgbouncer)

PostgreSQL Tuning / shared_buffers



Shared_buffer vs TPS

Shared_buffer	TPS
128 Megabyte	**500**
128 Megabyte	**3023**
128 Megabyte	**6801**
1 Gigabyte	**12898**
2 Gigabyte	**30987**
4 Gigabyte	**54536**
8 Gigabyte	**55352**
16 Gigabyte	**55364**

PostgreSQL Tuning / wal_buffer

- Do you have Transaction? Obviously ✓
- WAL – (Write Ahead LOG) Log your transactions
- Size of WAL files 16MB with 8K Block size (can be changed at compile time)
- PostgreSQL writes WAL into the buffers(*wal_buffer*) and then these buffers are flushed to disk.

Bigger value for wal_buffer in case of lot of concurrent connection gives better performance.

PostgreSQL Tuning effective_cache_size

- This used by the optimizer to estimate the size of the kernel's disk buffer cache.
- The effective_cache_size provides an estimate of the memory available for disk caching.
- It is just a guideline, not the exact allocated memory or cache size.

PostgreSQL Tuning / work_mem

- This configuration is used for complex sorting.
- It allows PostgreSQL to do larger in-memory sorts.
- Each value is per session based, that means if you set that value to 10MB and 10 users issue sort queries then 100MB will be allocated.
- In case of merge sort, if x number of tables are involved in the sort then **`x * work_mem`** will be used.
- It will allocate when required.
- Line in EXPLAIN ANALYZE `“Sort Method: external merge  Disk: 70208kB”`

PostgreSQL Tuning / work_mem

```
postgres=# SET work_mem ='2MB';
postgres=# EXPLAIN ANALYZE SELECT * FROM foo ORDER BY id;
                                                              QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Gather Merge  (cost=848382.53..1917901.57 rows=9166666 width=9) (actual time=5646.575..12567.495 rows=11000000 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Sort  (cost=847382.51..858840.84 rows=4583333 width=9) (actual time=5568.049..7110.789 rows=3666667 loops=3)
         Sort Key: id
         Sort Method: external merge  Disk: 74304kB
         Worker 0:  Sort Method: external merge  Disk: 70208kB
         Worker 1:  Sort Method: external merge  Disk: 70208kB
         ->  Parallel Seq Scan on foo  (cost=0.00..105293.33 rows=4583333 width=9) (actual time=0.018..985.524 rows=3666667 loops=3)
 Planning Time: 0.055 ms
 Execution Time: 13724.353 ms
(11 rows)
```

```
postgres=# SET work_mem ='1GB';
postgres=# EXPLAIN ANALYZE SELECT * FROM foo ORDER BY id;
                                                                         QUERY PLAN
        -------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=1455965.01..1483465.01 rows=11000000 width=9) (actual time=5346.423..6554.609 rows=11000000 loops=1)
   Sort Key: id
   Sort Method: quicksort  Memory: 916136kB
   ->  Seq Scan on foo  (cost=0.00..169460.00 rows=11000000 width=9) (actual time=0.011..1794.912 rows=11000000 loops=1)
 Planning Time: 0.049 ms
 Execution Time: 7756.950 ms
(6 rows)


Time: 7757.595 ms (00:07.758)
```

maintenance_work_mem

- maintmaintenance_work_mem is a memory setting used for maintenance tasks.
- The default value is 64MB.
- Setting a large value helps in tasks like VACUUM, RESTORE, CREATE INDEX, ADD FOREIGN KEY and ALTER TABLE.

maintenance_work_mem

```
postgres=# CHECKPOINT;
postgres=# SET maintenance_work_mem='10MB';
postgres=# SHOW maintenance_work_mem;
 maintenance_work_mem
----------------------
 10MB
(1 row)
postgres=# CREATE INDEX idx_foo ON foo(id);
Time: 12374.931 ms (00:12.375)
```

```
postgres=# CHECKPOINT;
postgres=# SET maintenance_work_mem='1GB';
postgres=# SHOW maintenance_work_mem;
 maintenance_work_mem
----------------------
 1GB
(1 row)
postgres=# CREATE INDEX idx_foo ON foo(id);
Time: 9550.766 ms (00:09.551)
```

synchronous_commit

- This is used to enforce that commit will wait for WAL to be written on disk before returning a success status to the client.
- This is a trade-off between performance and reliability.
- Increasing reliability decreases performance and vice versa.

Synchronous commit doesn't introduce the risk of *corruption*, which is really bad, just some risk of data *loss*.

https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server

checkpoint_timeout

- PostgreSQL writes changes into WAL. The checkpoint process flushes the data into the data files.
- More checkpoints have a negative impact on performance.
- Frequent checkpoint reduces the recovery time

4.

Linux Tuning for PostgreSQL

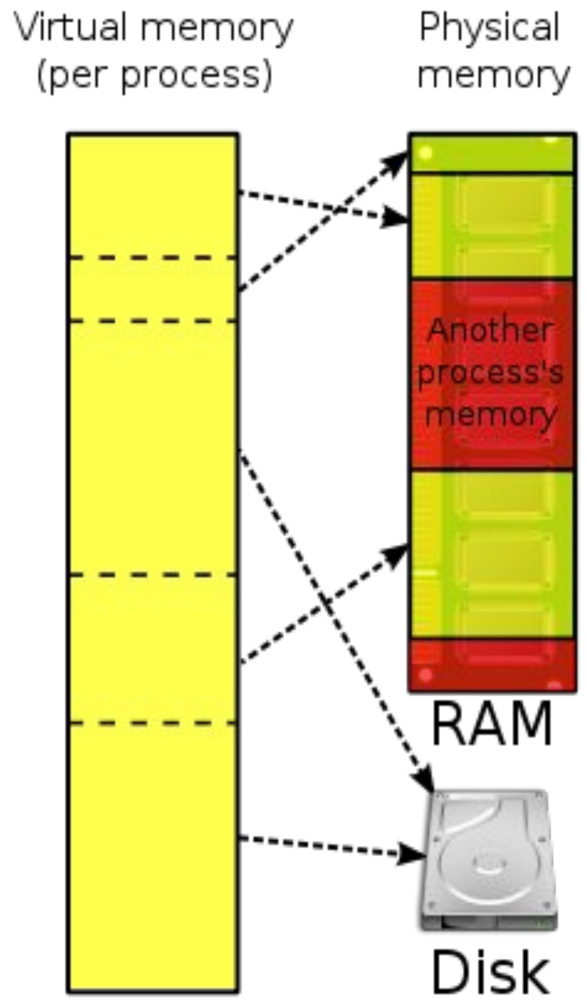
Ubuntu

Input Output Handling

- Direct IO, Buffered IO and Double buffering
- PostgreSQL believes that the Operating system (Kernel) knows much better about storage and IO scheduling.
- PostgreSQL has its own buffering; and also needs the pages cache. Double Buffering
- It Increase the use of memory.
- And different kernel and setting behave differently.

Virtual Memory

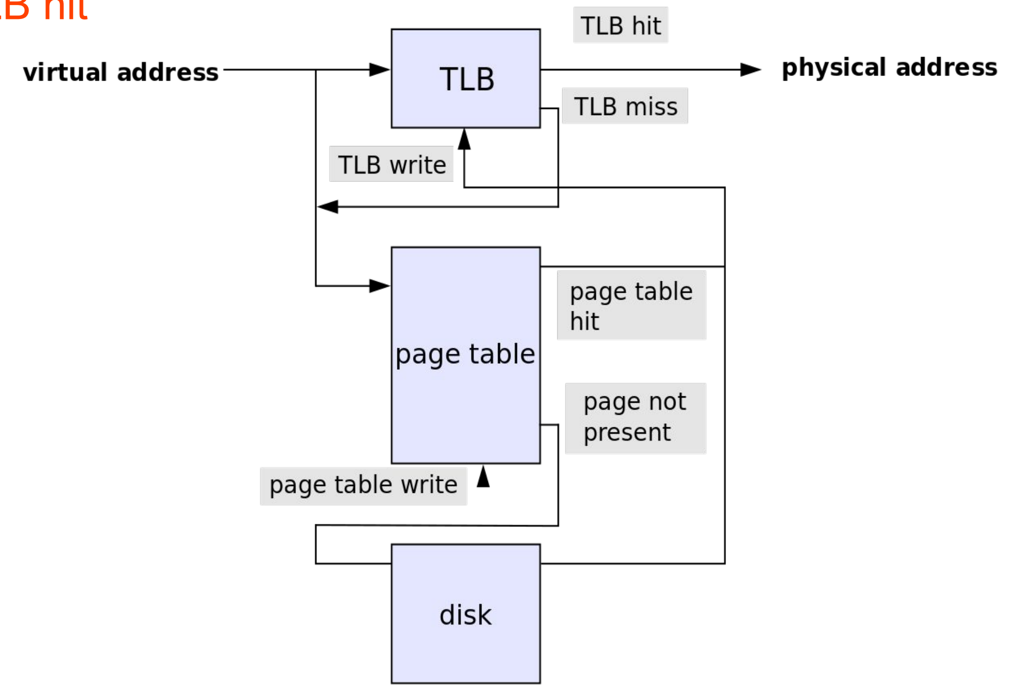
- Every process is given the impression that it is working with large, contiguous sections of memory
- Each process runs in its own dedicated address space
- Pages Table are used to translate the virtual addresses seen by the application into Physical Address



https://en.wikipedia.org/wiki/Virtual_memory
https://en.wikipedia.org/wiki/Page_table

Translation Lookaside Buffer (TLB)

- Translation Lookaside Buffer is a memory cache
- It reduce the time to access a user memory location
- If a match is found the physical address of the page is returned → TLB hit
- If match not found scan the page table (*walk*)
-    looking for the address mapping (entry) → TLB miss



Small page size bigger TLB size → expensive

Memory Pages

- Different OS has different size of Pages
- Linux has a concept of Classic Huge Pages and Transparent Huge Pages.
- BSD has Super Pages.
- Windows has Large Pages.

Linux Page sizes

- Linux Page size is 4K
- Many modern processors support other page sizes

If we consider a server with 256G of RAM:

4K	67108864
2M	131072
1G	256

large/huge pages

Classic Huge Pages

```
# cat /proc/meminfo
MemTotal:       264041660 kB
...
Hugepagesize:        2048 kB
DirectMap4k:       128116 kB
DirectMap2M:      3956736 kB
DirectMap1G:    266338304 kB

sysctl -w vm.nr_hugepages=256
```

Classic Huge Pages

```
# vi /etc/default/grub
  GRUB_CMDLINE_LINUX_DEFAULT="hugepagesz=1GB default_hugepagesz=1G”
# update-grub
      Generating grub configuration file ...
      Found linux image: /boot/vmlinuz-4.4.0-75-generic
      Found initrd image: /boot/initrd.img-4.4.0-75-generic
      Found memtest86+ image: /memtest86+.elf
      Found memtest86+ image: /memtest86+.bin
      Done
# shutdown -r now
```

Classic Huge Pages

```
# vim /etc/postgresql/10/main/postgresql.conf


huge_pages=ON # default is try


# service postgresql restart
```

Transparent Huge pages

- The kernel works in the background (khugepaged) trying to:
  - "create" huge pages.
  - Find enough contiguous blocks of memory
  - Convert them into a huge page
- Transparently allocate them to processes when there is a "fit"

Disabling Transparent Huge pages

```
# cat /proc/meminfo | grep AnonHuge

AnonHugePages:       2048 kB


# ps aux | grep huge

root        42  0.0  0.0      0     0 ?        SN   Jan17   0:00 [khugepaged]
```

To disable it:

- at runtime:

```
  # echo never > /sys/kernel/mm/transparent_hugepage/enabled

  # echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

- at boot time:

```
  GRUB_CMDLINE_LINUX_DEFAULT="(...) transparent_hugepage=never"
```

vm.swappiness

- This is another kernel parameter that can affect the performance of the database.
- Used to control the swappiness (swapping pages to swap memory into RAM) behavior on a Linux system.
- The parameter can take anything from “0” to “100”.
- The default value is 60.
- Higher value means more aggressively swap.

```
sudo sh -c 'echo 1 > /proc/sys/vm/swappiness'

sudo sh -c 'echo "vm.swappiness = 1" >> /etc/sysctl.conf'
```

vm.overcommit_memory and vm.overcommit_ratio

- Applications acquire memory and free that memory when it is no longer needed.
- But in some cases, an application acquires too much memory and does not release it. This can invoke the OOM killer.
- This is used to control the memory over-commit.
- It has three options
  - 0 - Heuristic overcommits, Do it intelligently (default); based kernel heuristics
  - 1 - Allow overcommit anyway
  - 2 - Don’t over commit beyond the overcommit ratio.

```
sudo sh -c  ‘echo 2 > /proc/sys/vm/overcommit_memory’
sudo sh -c ‘echo 50 > /proc/sys/vm/overcommit_ratio’
```

vm.dirty_background_ratio and vm.dirty_background_bytes

- The vm.dirty_background_ratio is the percentage of memory filled with dirty pages that need to be flushed to disk.
- Flushing is done in the background.
- The value of this parameter ranges from 0 to 100;

vm.dirty_ratio / vm.dirty_bytes

- The vm.dirty_background_ratio is the percentage of memory filled with dirty pages that need to be flushed to disk.
- Flushing is done in the foreground.
- The value of this parameter ranges from 0 to 100;

5.

Parallel Query Optimizations

Parallel Query

Parallel Query

- How Parallel Query Works?
    - Actual working mechanism of parallel queries.
- When Can Parallel Query Be Used?
    - When parallel query can be used, and when it can not.
- Parallel Plans
    - Different parallel plans available.
- Parallel Safety
    - Parallel query operations are parallel safe, parallel restricted, or parallel unsafe.

https://www.postgresql.org/docs/12/parallel-plans.html

How Parallel Query Works - Configuration

- **max_worker_processes** The default is 8

  Sets the maximum number of background processes that the system can support. This parameter can only be set at server start..

- **max_parallel_workers_per_gather** *(integer)* The default value is 2

  Sets the maximum number of workers that can be started by a single Gather or Gather Merge node.

- **max_parallel_workers** *(integer)* The default value is 8

  Sets the maximum number of workers that the system can support for parallel queries.

- **dynamic_shared_memory_type** *(enum)*

  Parallel query requires dynamic shared memory in order to pass data between cooperating processes, so there should not be set to "none"

  - Posix
  - sysv
  - windows
  - mmap

- **parallel_workers** *(integer)* Sets the number of workers that should be used to assist a parallel scan of this table

postgresql.conf is the place where you can set these.
Some of the setting can be set on psql useng set command.

How Parallel Query Works - Configuration

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM foo;
 Finalize Aggregate
   ->  Gather
         Workers Planned: 2
         Workers Launched: 0
         ->  Partial Aggregate
               ->  Parallel Seq Scan on foo
```

Check
1 - max_worker_processes
2 - max_parallel_workers

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM foo;
Finalize Aggregate
  -> Gather
     Workers Planned: 1
     Workers Launched: 1
     -> Partial Aggregate
        -> Parallel Seq Scan on foo
```

Check
1 - max_parallel_workers_per_gather
2 - parallel_workers

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM foo;
Finalize Aggregate
  -> Gather
     Workers Planned: 1
     Workers Launched: 1
     -> Partial Aggregate
        -> Parallel Seq Scan on foo
```

ALTER TABLE foo SET (parallel_workers = 2);

```
ALTER TABLE foo SET (parallel_workers = 2);
EXPLAIN ANALYZE SELECT COUNT(*) FROM foo;
Finalize Aggregate
  -> Gather
     Workers Planned: 2
     Workers Launched: 2
     -> Partial Aggregate
        -> Parallel Seq Scan on foo
```

These are quick tips, there may be some other reasons.

How Parallel Query Works - Example

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                              QUERY PLAN
---------------------------------------------------------------------------------------------------
Gather (cost=1000.00..217018.43 rows=1 width=97)
    Workers Planned: 2
    -> Parallel Seq Scan on pgbench_accounts (cost=0.00..216018.33 rows=1 width=97)
Filter: (filler ~~ '%x%'::text) (4 rows)
```

If the Gather or Gather Merge node is at the very top of the plan tree, then the entire query will execute in parallel, otherwise only the portion of the plan below it will run in parallel

When Can Parallel Query Be Used?

- Number of backgrounds < max_worker_processes. ✓
- Background workers for purposes of parallel query < max_parallel_workers. ✓
- The client sends an execute message with a non-zero fetch count. ✗
- When a prepared statement is executed using a CREATE TABLE .. AS EXECUTE .. statement. ✗
- When the transaction isolation level is serializable*. ✗
- The query uses any function marked PARALLEL UNSAFE. ✗
- The query is running inside of another query that is already parallel. ✗
- The query might be suspended during execution. ✗
  - PL/pgSQL Loops of the form FOR x IN query LOOP .. END LOOP
  - DECLARE CURSOR
- The query writes any data or locks any database rows. If a query contains a data-modifying operation (top level/within a CTE)*. ✗

Default value of max_worker_processes and max_parallel_workers are 8.
* Limitation of current implementation.

When Can Parallel Query Be Used? 2/2

- Windowed functions and ordered-set aggregate functions are non-parallel*. ✗
- Support for FULL OUTER JOIN ✗

** This is a limitation of the current implementation.*

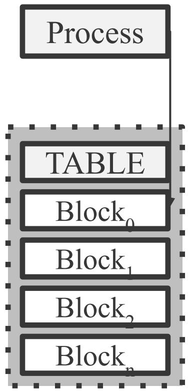
Parallel Plans

- Parallel Scans
  - Parallel Sequential Scan
  - Parallel Bitmap Heap Scan
  - Parallel Index Scan
  - Parallel Index Only Scan
- Parallel Joins
- Parallel Aggregation
- Parallel Append

Parallel Sequential Scan

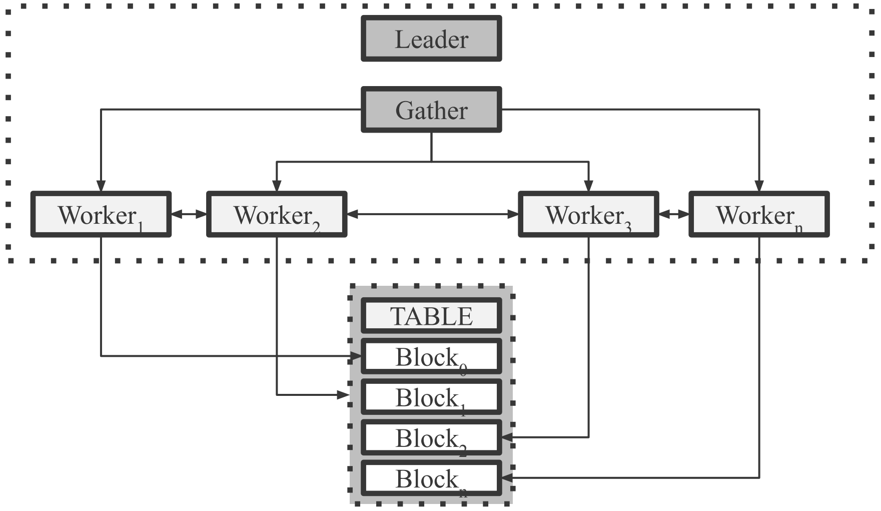
```
CREATE TABLE foo AS SELECT id AS id, 'name'::TEXT||id::TEXT AS name FROM
generate_series(1,10000000) AS id;
EXPLAIN ANALYZE SELECT * FROM foo WHERE id %2 = 10;
                              QUERY PLAN
----------------------------------------------------------------------------
----Seq Scan on foo  (cost=0.00..204052.90 rows=49999 width=15)
Planning Time: 0.063 ms
Execution Time: 1657.964 ms
```

Planning time ?



```
EXPLAIN ANALYZE SELECT * FROM foo WHERE id %2 = 10;
                                        QUERY PLAN
--------------------------------------------------------------------------------
Gather
Workers Planned: 4
Workers Launched: 4
->  Parallel Seq Scan on foo  (cost=0.00..91554.48 rows=12500 width=15)
         Filter: ((id % 2) = 10)
 Planning Time: 0.074 ms
 Execution Time: 469.946 ms
```

Reduce query execution time by using machines multiple CPUs



No benefit if table size is < min_parallel_table_scan_size

Parallel Bitmap Heap Scan

- One process is chosen as the **leader**.
- Leader performs a scan of one or more indexes and builds a bitmap indicating which table blocks need to be visited.
- These blocks are then divided among the cooperating processes as in a parallel sequential scan.

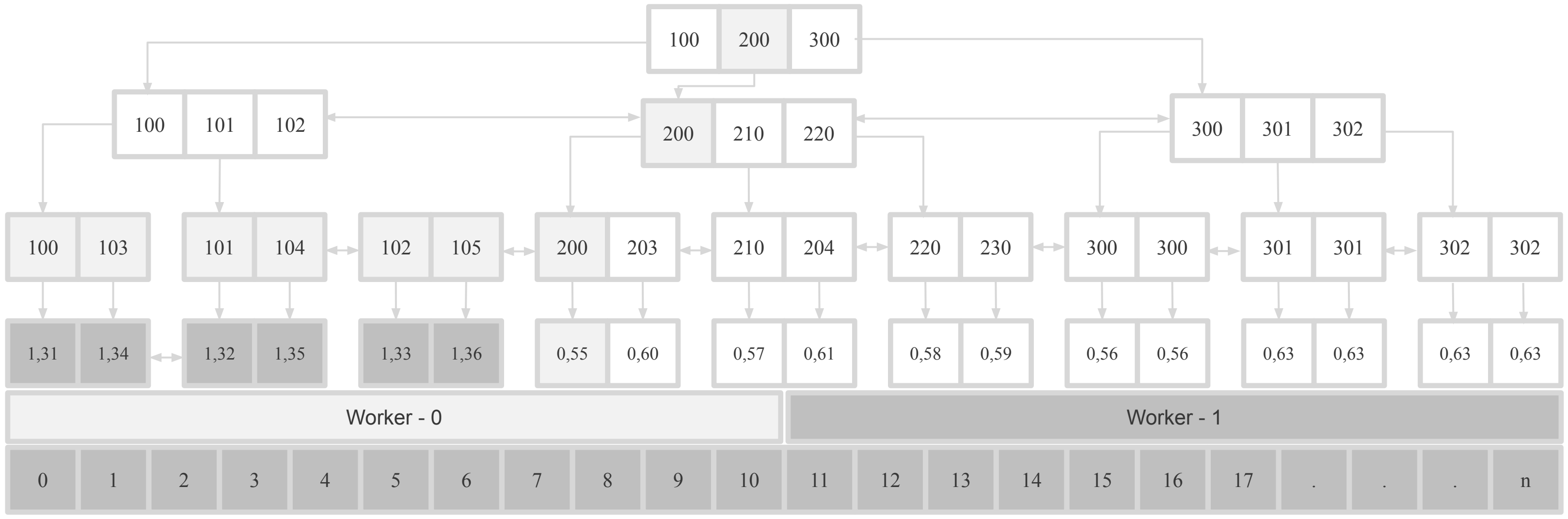
```
EXPLAIN (COSTS OFF )SELECT COUNT(*) FROM foo JOIN bar ON id =
v WHERE v > 10 AND name ~ 'name%';
                          QUERY PLAN
---------------------------------------------------------
 Finalize Aggregate
   ->  Gather
         Workers Planned: 2
         ->  Partial Aggregate
               ->  Nested Loop
                  ->  Parallel Bitmap Heap Scan on bar
                  Recheck Cond: (v > 10)
                       ->  Bitmap Index Scan on idx1
                             Index Cond: (v > 10)
                   ->  Bitmap Heap Scan on foo
                       Recheck Cond: (id = bar.v)
                       Filter: (name ~ 'name%'::text)
                       ->  Bitmap Index Scan on idx
                             Index Cond: (id = bar.v)
```

There is no benefit if bitmap size is very small.

Parallel Index Scan

```
EXPLAIN (costs off) SELECT id FROM foo WHERE val % 100 = 2 and id < 200000;
                    QUERY PLAN
--------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Index Only Scan using idx1 on foo
         Index Cond: (id < 200000)
         Filter: ((val % 100) = 2)
(5 rows)
```

- Worker-0 will start from the root node and scan until the leaf node 200.
- It'll handover the next block under node 105 to Worker-1, which is in a blocked and wait-state.
- If there are other workers, blocks are divided into the workers



Parallel Joins

Parallel Hash Join

In case of ***hash join*** the inner side is executed in full by every cooperating process to build identical copies of the hash table.

What if hash table is too big ?

In ***parallel hash*** *join the* inner side is a parallel hash that divides the work of building a shared hash table over the cooperating processes.

All workers participate in creating the Hash table.

After the hash, each worker can perform the join.

Merge Join

The inner side is always a non-parallel plan and therefore executed in full.

This may be inefficient, especially if a sort must be performed, because the work and resulting data are duplicated in every cooperating process.

Nested Loop Join

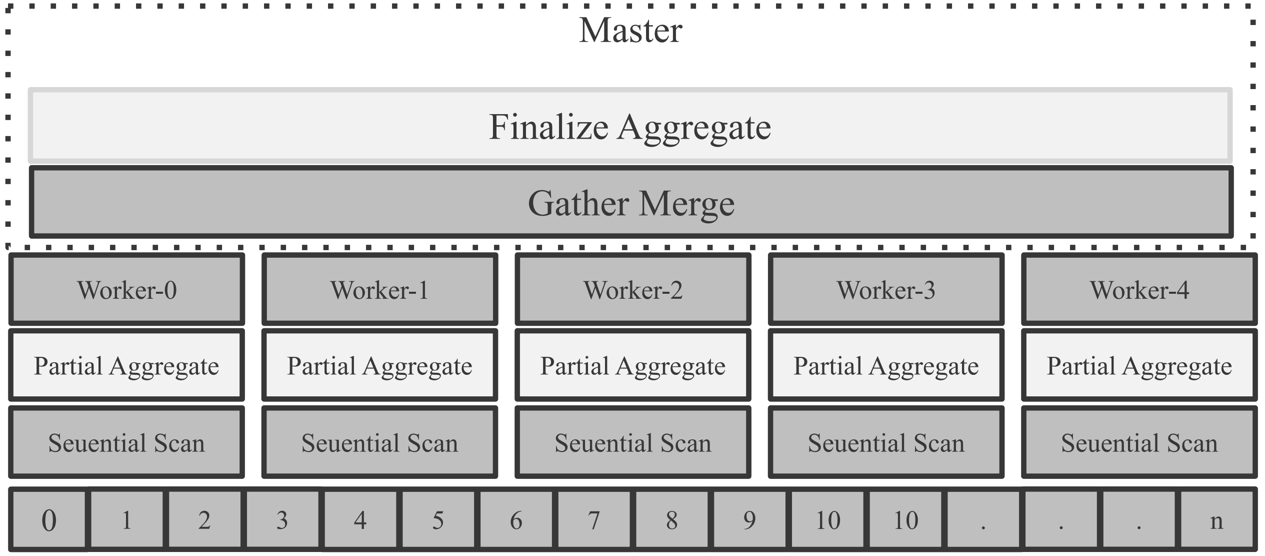
The inner side is always non-parallel.

The outer tuples and thus the loops that look up values in the index are divided over the cooperating processes.

```
EXPLAIN ANALYZE SELECT COUNT(*)
FROM foo JOIN bar ON id = v WHERE v > 10;
                          QUERY PLAN
-------------------------------------------------------------
-Finalize Aggregate
   ->  Gather
    Workers Planned: 2
    Workers Launched: 2
          ->  Partial Aggregate
                ->  Parallel Hash Join
                      Hash Cond: (bar.v = foo.id)
                      ->  Parallel Seq Scan on)
                            Filter: (v > 10)
                            Rows Removed by Filter: 500058
                      ->  Parallel Hash
                            ->  Parallel Seq Scan on foo
 Planning Time: 0.211 ms
 Execution Time: 758.847 ms
```

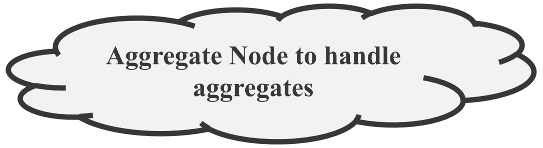
Parallel Aggregation

- Worker performs the scans on the pages and applies the filter.
- Transfer all the tuples to master.
- Master performs the final aggregate.

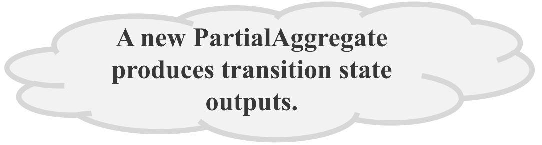


```
EXPLAIN (COSTS FALSE) SELECT sum(id) FROM foo WHERE id
%2 = 10;
                    QUERY PLAN
  -----------------------------------------------

 Aggregate
   -> Seq Scan on foo
        Filter: ((id % 2) = 10)
(3 rows)
```



```
EXPLAIN (COSTS FALSE) SELECT sum(id) FROM foo WHERE id %2 = 10;
                         QUERY PLAN
  -------------------------------------------------------------
 Finalize Aggregate
   -> Gather
        Workers Planned: 4
        -> Partial Aggregate
             -> Parallel Seq Scan on foo
                  Filter: ((id % 2) = 10)
```



Parallel aggregation is not supported if any aggregate function call contains DISTINCT or ORDER BY clause

Parallel Append

```
\d+ foo;
                Partitioned table "public.foo"
 Column |  Type    |
------+--------------
 id     | integer
 name   | text
Partition key: RANGE (id)
Partitions: foo1 FOR VALUES FROM ('-1000000') TO (1),
            foo2 FOR VALUES FROM (1) TO (1000000)
```

- **Append** node used to combine rows from multiple sources into a single result set.
- In **Append**, all the participating processes cooperate to execute the first child plan until it is complete and then move to the second plan at around the same time
- In **Parallel Append** executor spread out the participating processes to its child plans, so that multiple child plans are executed simultaneously
- Avoids CPU contention and enhances I/O parallelism.

```
EXPLAIN (costs off)SELECT count(id) FROM foo WHERE id <= 201;
                    QUERY PLAN
--------------------------------------------------------
 Finalize Aggregate
   ->  Gather
         Workers Planned: 2
         ->  Partial Aggregate
               ->  Parallel Append
                     ->  Parallel Seq Scan on foo1 foo_1
                           Filter: (id <= 201)
                     ->  Parallel Seq Scan on foo2 foo_2
                           Filter: (id <= 201)
```

Used to combine rows from multiple sources

enable_parallel_append can be used to disable/enable parallel append.

Parallel Labeling for Functions and Aggregates 1/2

- **Parallel safe**

  A parallel safe operation is one which does not conflict with the use of parallel query

- **Parallel restricted**

  A parallel restricted operation is one which cannot be performed in a parallel worker, but which can be performed in the leader while parallel query is in use

- **Parallel unsafe**

  A parallel unsafe operation is one which cannot be performed while parallel query is in use, not even in the leader.

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL PARALLEL RESTRICTED;

CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL PARALLEL SAFE;

\df+ add
                     List of functions
 Schema | Name | Parallel
+---------+----------+------------------
 public | add   | safe
(1 row)
```

All user-defined functions are assumed to be parallel unsafe unless otherwise marked.

Parallel Labeling for Functions and Aggregates 2/2

- The following operations are always parallel restricted:
  - Scans of common table expressions (CTEs).
  - Scans of temporary tables.
  - Scans of foreign tables, unless the foreign data wrapper has an IsForeignScanParallelSafe API which indicates otherwise.
  - Plan nodes to which an InitPlan is attached.
  - Plan nodes which reference a correlated SubPlan.

Parallel Index Creation

- PostgreSQL can build indexes while leveraging multiple CPUs in order to process the table rows faster. This feature is known as *parallel index build*.
- Generally, a cost model automatically determines how many worker processes should be requested, if any.

```
CREATE INDEX idx ON foo(id);
CREATE INDEX
Time: 11815.685 ms (00:11.816)

SET max_parallel_maintenance_workers = 8;
CREATE INDEX idx ON foo(id);
CREATE INDEX
Time: 17281.309 ms (00:17.281)
```

Only B-tree parallel build is supported.

Blog Posts

- Tuning PostgreSQL Database Parameters to Optimize Performance.
    - https://www.percona.com/blog/2018/08/31/tuning-postgresql-database-parameters-to-optimize-performance/
- Tune Linux Kernel Parameters For PostgreSQL Optimization
    - https://www.percona.com/blog/2018/08/29/tune-linux-kernel-parameters-for-postgresql-optimization/

Thanks!

Any questions?

You can find me at:

- Ibrar.ahmed@percona.com
- Pgelephant.com