Troubleshoot PostgreSQL Performance

By Use Case

Wanda He
Prin. Database Specialist Solutions Architect, AWS

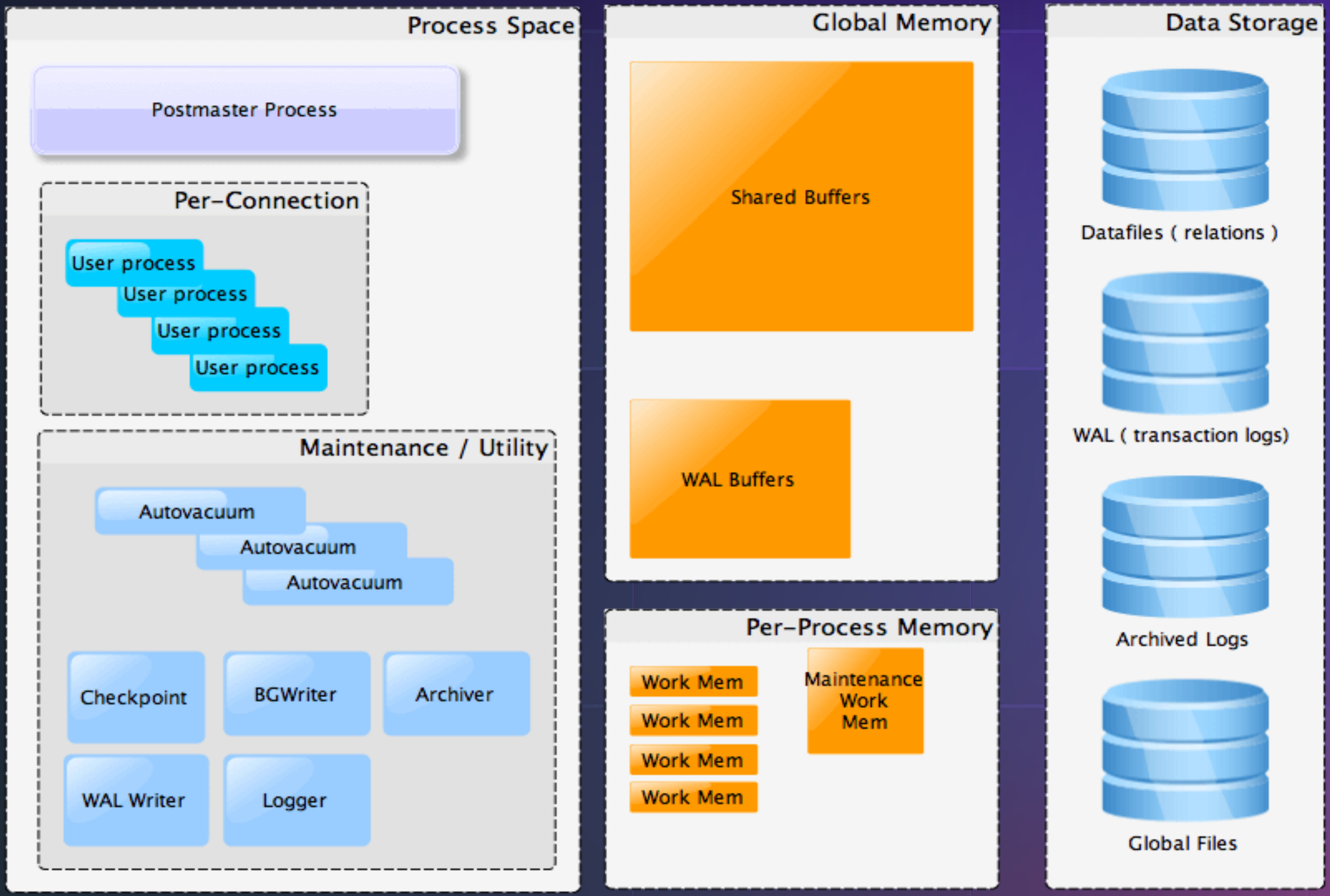
Thuymy Tran
Database Specialist Solutions Architect, AWS

Agenda

- PostgreSQL architecture
- Key performance factors
- Demo of performance use cases
  - Impacts for idle connections
  - Vacuum not able to clean-up dead tuples
  - Impacts of subtransactions
- 5 Tips to better performance

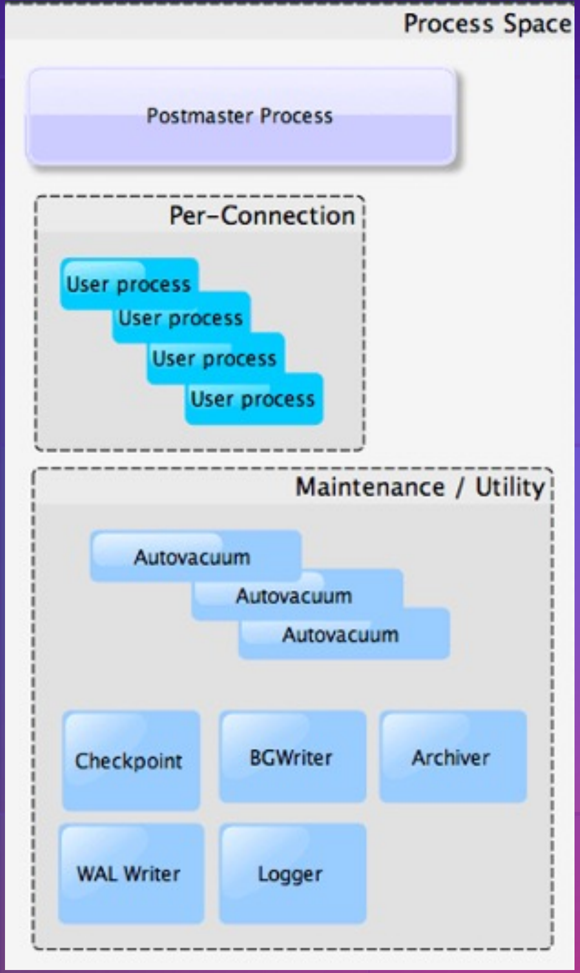
PostgreSQL Architecture

PostgreSQL Architecture



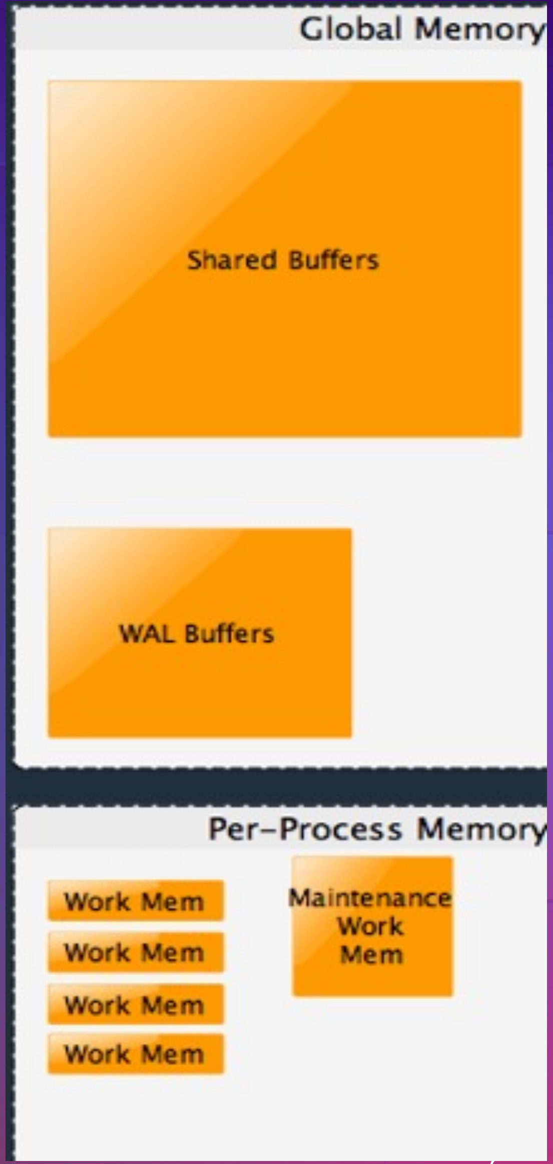
Processes

- PostgreSQL utilizes a multi-process architecture
- Each connection maps to a server (backend) process
- Types of processes
  - Primary (postmaster)
  - Per-connection backend process
    - Dedicated, per-connection server process
    - Known as a ‘worker’ process
    - Responsible for fetching data, communicating with the client
  - Utility (e.g. checkpointer, wal-writer, autovacuum, etc.)



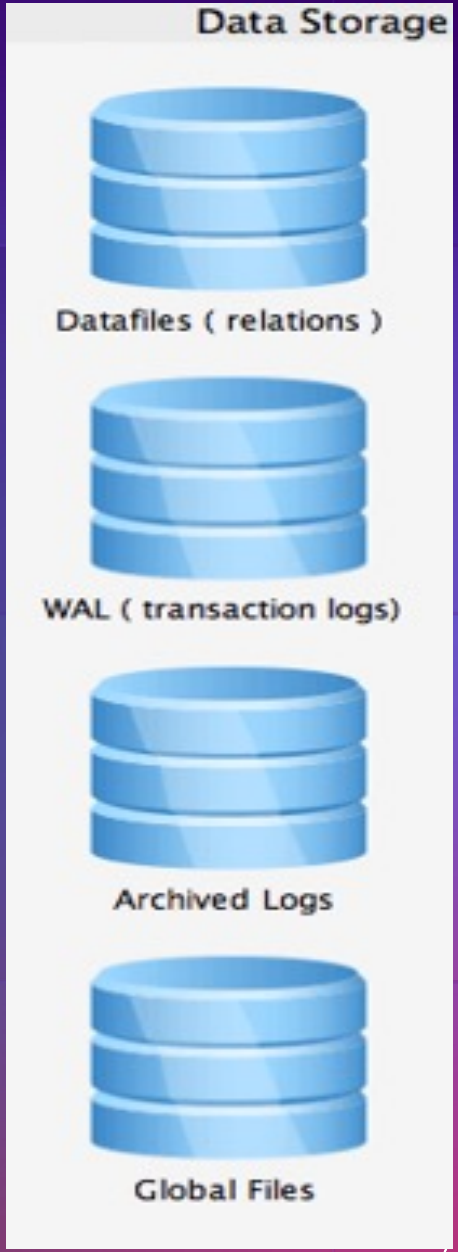
Memory

- Global shared memory
  - Shared buffers – cache for data pages
  - WAL buffers
- Local private memory
  - Per process memory
  - Caches for
    - Meta data
    - Prepare statement, execution plans, etc.
    - Query execution memory (e.g., work_mem, maintenance_work_mem)



On-disk storage

- Data files – “base” subdirectory
  - Table and index data
  - 1 GB file/segment
  - 8K block/page size, use TOAST for large value storage
- Write ahead log (WAL) files – “pg_wal” subdirectory
  - Also known as: WAL, xlog, pg_xlog, transaction log, journal, REDO
  - 16 MB file/segment
  - max_wal_size
- Archived Logs – archived area of WALs
- Global files - cluster-wide tables, system views

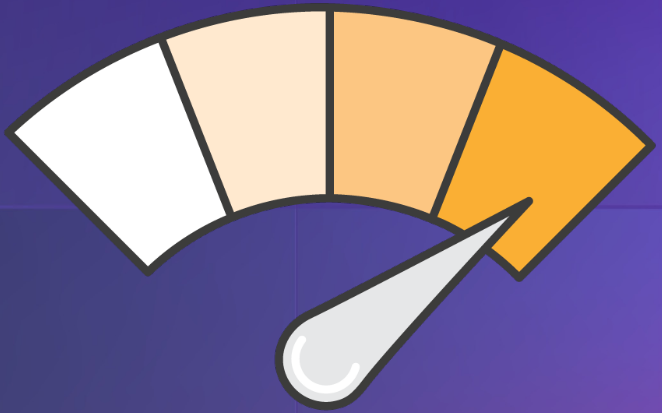


Key Performance Factors

- CPU
- Memory
- Storage
- Application Access Pattern

CPU

- Common pitfalls
  - Sub-optimal query
  - Under-size or over-subscription of CPU
    - # of active connections : # of vCPU ratio too high
  - Large parallel query
    - max_parallel_workers_per_gather
- Monitoring
  - % Total CPU Utilization
  - % System CPU Utilization
  - % Wait CPU Utilization
  - PostgreSQL pid % CPU Utilization with pg_proctab extension



Memory

- Common pitfalls
  - High # of connections
  - Sub-optimal memory intensive query
  - Over configuration (e.g. work_mem)
- Configuration setting
  - shared_buffers
  - work_mem
  - logical_decoding_work_mem (v13+)
  - maintenance_work_mem

- Monitoring tools
  - pg_proctab extension
  - pg_backend_memory_contexts (v14+)
  - pg_log_backend_memory_contexts (v14+)
  - aurora_stat_memctx_usage() Aurora PostgreSQL system function
- Metrics
  - Freeable memory
  - % buffer cache hit ratio

Storage

- Performance dimensions
  - IOPs
  - MB/s
  - Latency – key factor for OLTP workloads
- Common pitfalls
  - Insufficient sizing
  - Sub-optimal query
  - Table/Index bloat → vacuum not effective
  - In-efficient checkpoint
- Configuration tuning
  - max_wal_size
  - checkpoint_timeout
  - autovacuum_*
  - work_mem
- Metrics
  - Disk read/write latency per IO
  - Disk reads/writes per second
  - Disk MB reads/writes per second

Application access patterns

- Blocking/locking due to application access pattern
- Common pitfalls
  - Long-running transaction
  - Idle in transaction
  - Idle connections
  - Greater than 64 subtransations
  - Excessive use of FKs

Demo of performance use cases

Impacts of idle connections

Background:

If a connection is not consuming resources with it’s not doing anything. Therefore, it’s good to have large number of idle connections open so that they can be ready to serve when they are needed.

Demo

Vacuum not able to clean-up dead tuples

Background:

Query performance can degrade overtime when vacuum is not able to clean-up tuples effectively

Demo

Greater than 64 subtransactions

Background:

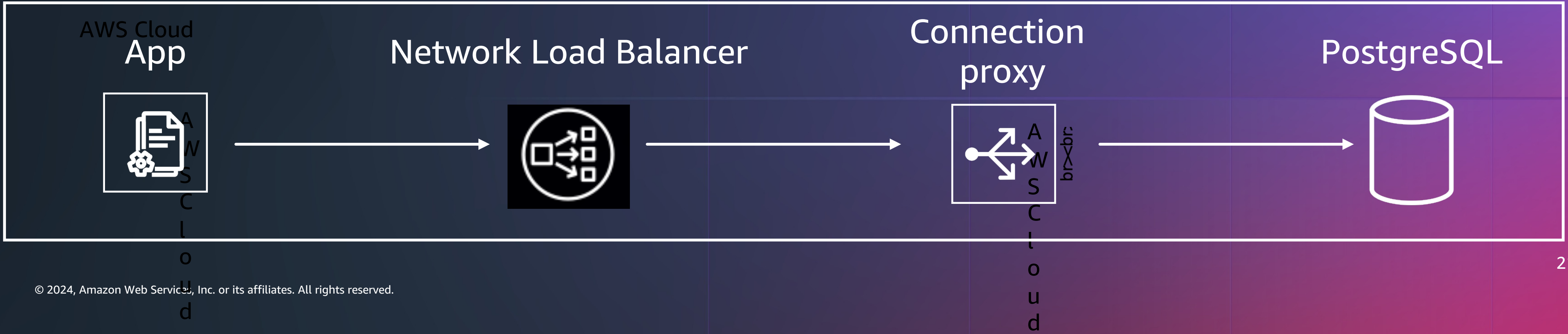
Each PostgreSQL backend can cache up to 64 subtransactions in SLRU. Excessive use of subtranctions (> 64) creates SLRU cache contention. Performance degrades due to SLRU page locking and additional disk I/O to bring pages from the pg_subtrans directory.

Demo

5 Tips to Better Performance

Tip #1: Scale with connection pooling

- Connection is expensive
  - Connection local cache (catalog cache, prepare statement, and etc.)
  - High CPU context switches when ratio of CPU : active connections is high
- Enhance scalability with connection pooling solution
  - PgBouncer, Pgpool-II, Amazon RDS Proxy



Tip #2: Control bloat with effective vacuuming

- Create process for ongoing monitoring of bloated table / index
  - https://wiki.postgresql.org/wiki/Show_database_bloat or pgstattuple extension
- Tune autovacuum/manual vacuum to minimize bloat
  - Default setting may not be sufficient
  - Use table level tuning for large tables
  - Avoid blocking for vacuum
- Rebuild to release storage back to OS
  - Rebuild index (online option)
  - Online rebuild with pg_repack extension (online)
  - Rebuild with vacuum full (offline operation, generally not recommended)

Tip #3: Query performance tuning

- Recommended configuration settings
  - Random_page_cost: 1.0 (for SSD storage)
  - Work_mem: 4MB for OLTP only, up to 4GB for DW only
  - Max_parallel_workers_per_gather: -1 for OLTP only, up to 8 from DW only
  - Shared_preload_libraries: pg_stat_statements, pg_hint_plan
  - Default_statistics_target: 256
- Optimize index usage
  - Check missing index: https://www.postgresonline.com/journal/archives/65-How-to-determine-which-tables-are-missing-indexes.html
  - Check redundant/Unused indexes: https://wiki.postgresql.org/wiki/Index_Maintenance
- Consider partitioning very large tables (e.g. >100 million rows)

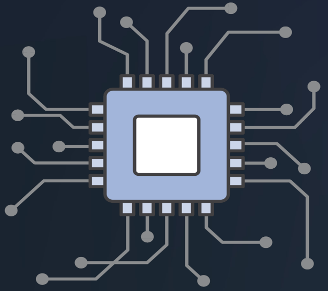
Tip #4: Implement scalable application access pattern

- Avoid long-running transactions
  - Reduce/eliminate idle-in-transaction
  - Use explicit “begin … commit/rollback” for transaction
  - Enable autocommit
- Avoid connection storm
  - Implement application timeouts, retries, backoff
- Avoid using subtransactions when possible
- Avoid excessive use of FKs
  - Implement reference integrity check with application business logics if possible

Tip #5: Scale with managed infrastructure options

AWS flexible scaling options with managed relational database offerings

Amazon RDS/Aurora Database instance Class

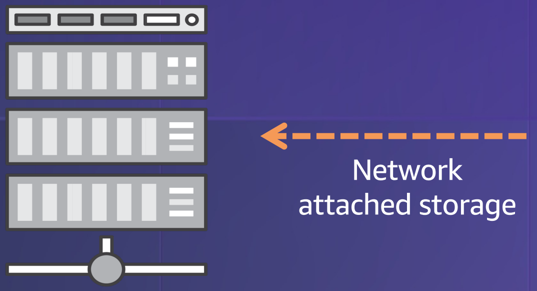


Compute Capabilities

Up to 128 vCPUs

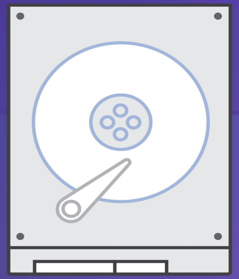
Memory Capabilities

Up to 4TB of RAM



Network Performance

Up to 100 GB/s (Throughput)



Storage Performance

I/O Throughput

Amazon RDS Storage Type

Thank you!

Wanda He
wanhe@amazon.com

Thuymy Tran
thuymyt@amazon.com