credativ

PgBouncer Everything, Everywhere, All At Once About This Tool

PART 1

Josef Machytka <josef.machytka@credativ.de>

2026-01-27 - P2D2 2026 Workshop

credativ GmbH

- Founded 1999 in Jülich, Germany
- Close ties to Open-Source Community
- More than 40 Open-Source experts
- Consulting, development, training, support (3rd-level / 24x7)
- Open-Source infrastructure with Linux, Kubernetes, Proxmox
- Open-Source databases with PostgreSQL
- DevSecOps with Ansible, Puppet, Terraform and others
- Since 2025 independent owner-managed company again

Josef Machytka

credativ

- Professional Service Consultant - PostgreSQL specialist at credativ GmbH
- 33+ years of experience with different databases
- PostgreSQL (13y), BigQuery (7y), Oracle (15y), MySQL (12y), Elasticsearch (5y), MS SQL (5y)
- 10+ years of experience with Data Ingestion pipelines, Data Analysis, Data Lake and Data Warehouse
- 3+ years of practical experience with different LLMs / AI / ML including architecture and principles
- From Czechia, living now 12 years in Berlin

- LinkedIn: linkedin.com/in/josef-machytka
- Medium: medium.com/@josef.machytka
- YouTube: youtube.com/@JosefMachytka

- GitHub: github.com/josmac69/conferences_slides
- ResearchGate: researchgate.net/profile/Josef-Machytka

All My Slides:

Recoded talks:

What We Will Cover In Part 1

- Why Connection Pooling?
  - PostgreSQL connections are expensive
  - How Much Memory PostgreSQL Connection Uses?
  - Where is work_mem hidden in memory usage?
  - How many connections can run on 1 CPU core?
  - Connection Poolers Architectures
  - Thundering Herd Problem
  - Level-Triggered vs Edge-Triggered I/O
- Connection Poolers Overview
  - Pgpool-II
  - PgCat
  - Odyssey

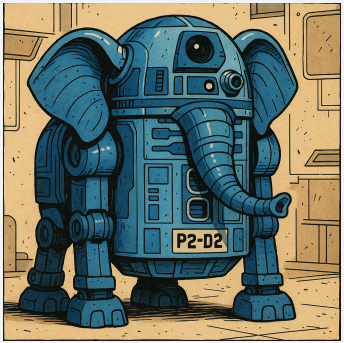


Image created by ChatGPT

credativ

Why Connection Pooling?

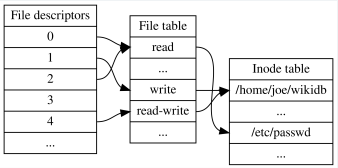
When We Shall Use Connection Pooling?

- Applications with high connection churn
- Microservices with many short-lived connections
- Jobs that spin up dramatically on demand
- Spiky workloads with unpredictable traffic
  - Black Friday (10x) / Cyber Monday sales (4x) / Sport events (4x-6x)
  - Periodic spikes (daily/weekly/seasonal)
  - Sudden bursts of traffic due to campaigns/news
- max_connections too high or reached frequently
- PostgreSQL docs recommend max 100 connections
- 300+ considered reasonable upper limit
- Some clients set it to 1500+ and see big issues

Image from the article Seasonal Web Traffic

PostgreSQL connections are expensive

- PostgreSQL uses process-per-connection model
- Client connects via TCP/IP or local Unix socket
- Leads to expensive CPU, memory, and context-switching overhead
- High connection counts (>300–400) degrade performance
- Too many context switches between processes
- Too many Translation Lookaside Buffer (TLB) misses
- Each connection uses at least one socket
- Socket needs typically from 16KB to 128KB of kernel memory
- ”cat /proc/net/sockstat” -> ”sockets: used 2057”
- One connection usually needs more file descriptors
- Each process needs one semaphore to coordinate with others



Schema af file descriptors from Wikipedia

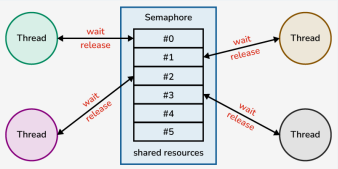


Image from the article Mutex vs Semaphore

PostgreSQL connections are expensive

- Idle session still holds memory (4-20 MB each)
- 100 idle connections = 400 MB to 2 GB RAM
- Each connection keeps cache of metadata for visited objects
- Tables with many partitions / many indexes increase cache size
- Prepared statements in session / cached query plans
- work_mem is setting per individual part of query
- Complex queries can use multiple work_mem allocations
- Multiple active queries can exhaust system memory quickly

Find more in my talk
PostgreSQL Connection Memory Usage

How Much Memory PostgreSQL Connection Uses?

- PC 32GB memory, PostgreSQL 17, shared_buffers=8GB, effective_cache_size=24GB, work_mem=64MB
- Connected with psql, connection is newly created, no command issued yet
- Python script with *psutil* library calls *memory_full_info()*
- When I closed the connection and opened again, I got similar numbers

```
## output of top command
   PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
190747 postgres  20   0 8701512  20248  16876 S   0.0   0.1   0:00.00 postgres: postgres postgres 172.18.0.1(40278) idle

## python script output - psutil.memory_full_info()
PID: 190747, Command: postgres: postgres postgres 172.18.0.1(40278) idle
    rss:          19.8 MB
    vms:        8497.6 MB
    shared:       16.5 MB
    text:          5.4 MB
    lib:           0.0 MB
    data:          3.6 MB
    dirty:         0.0 MB
    uss:           2.2 MB
    pss:           8.4 MB
    swap:          0.0 MB
```

How smaps Looks Like for PostgreSQL Connection?

credativ

- Another Python script used to parse and pivot the /proc/PID/smaps file, to show the memory usage
- Here is detailed view - showed 42 different /usr/lib/x86_64-linux-gnu/ libraries
- And many small regions without paths -> summarized together as [anonymous]

```
## output of top command
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 190747 postgres  20   0 8701512  20248  16876 S   0.0   0.1   0:00.00 postgres: postgres postgres 172.18.0.1(40278) idle

## python script output - smaps
```

Path	Size	Rss	Pss	Pss_Dirty	Shr_Clean	Shr_Dirty	Prv_Clean	Prv_Dirty	Swap	SwapPss	Cnt
/usr/lib/postgresql/16/bin/postgres	9296	4140	1156	75	3792	168	128	52	0	0	5
[anonymous]	1708	660	554	554	0	120	0	540	0	0	21
[heap]	1440	1132	821	821	0	368	0	764	0	0	2
/dev/shm/PostgreSQL.1436672634	1024	132	130	130	0	4	0	128	0	0	1
/dev/shm/PostgreSQL.3104938386	112	4	1	1	0	4	0	0	0	0	1
/dev/zero (deleted)	8624208	10352	5070	5070	0	9780	0	572	0	0	1
/usr/lib/postgresql/16/lib/auto_explain.so	20	8	0	0	0	8	0	0	0	0	5
/usr/lib/postgresql/16/lib/pg_stat_statements.so	44	8	0	0	0	8	0	0	0	0	5
/usr/lib/locale/locale-archive	2980	60	19	0	60	0	0	0	0	0	1
/usr/lib/x86_64-linux-gnu/libffi.so.8.1.2	48	8	0	0	0	8	0	0	0	0	5
/usr/lib/x86_64-linux-gnu/libgpg-error.so.0.33.1	160	8	0	0	0	8	0	0	0	0	5
/usr/lib/x86_64-linux-gnu/libgmp.so.10.4.1	516	8	0	0	0	8	0	0	0	0	5
....											
/SYSV00ce5741 (deleted)	4	0	0	0	0	0	0	0	0	0	1
/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2	208	80	18	9	64	8	0	8	0	0	5
[stack]	132	36	27	27	0	12	0	24	0	0	1
[vvar]	16	0	0	0	0	0	0	0	0	0	1
[vdso]	8	4	0	0	4	0	0	0	0	0	1
Total	8701512	20380	8553	6841	6356	11760	164	2100	0	0	251

Let's Run Some Heavy Query

credativ

- Let's run some heavy aggregations over the table not fitting into memory
- Memory is 32 GB, table has 38 GB, shared_buffers= 8 GB, work_mem= 64 MB
- No parallelism - max_parallel_workers_per_gather = 0
- What we see after the query execution (pg_buffercache_summary shows shared buffers fully used)

```
## top command output after query run
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 190747 postgres  20   0 8701848   8.2g   8.1g S   0.0  26.3   0:48.67 postgres: postgres postgres 172.18.0.1(40278) idle

## python script output - psutil.memory_full_info()
PID: 190747, Command: postgres: postgres postgres [local] idle
    rss:    8344.3 MB
    vms:    8535.2 MB
 shared:    8340.5 MB
   text:       5.6 MB
    lib:       0.0 MB
   data:       3.9 MB
  dirty:       0.0 MB
    uss:    8196.1 MB
    pss:    8251.4 MB
   swap:       1.0 MB
```

Let’s Run Some Heavy Query

- We must look into smaps again

```
## top command output after query run
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 190747 postgres  20   0 8701848   8.2g   8.1g S   0.0  26.3   0:48.67 postgres: postgres postgres 172.18.0.1(40278) idle

## smaps numbers after query run
Path                                            Size        Rss        Pss  Pss_Dirty  Shr_Clean  Shr_Dirty  Prv_Clean  Prv_Dirty      Swap  SwapPss  Cnt
-------------------------------------------------------------------------------------------------------------------------------------------------------
/usr/lib/postgresql/16/bin/postgres             9296       6508       3255         79       4176        164       2112         56         0        0    5
[anonymous]                                     1708        704        598        598          0        120          0        584         0        0   21
[heap]                                          1776       1516       1208       1208          0        364          0       1152         0        0    2
/dev/shm/                                       1136        148        143        143          0          8          0        140       980        0    2
/dev/zero (deleted)                          8624208    8532008    8443508    8443508          0     143376          0    8388632         0        0    1
/usr/lib/postgresql/16/lib/                       64         44         22          8         28          8          0          8         0        0   10
/usr/lib/locale/locale-archive                  2980         68         19          0         68          0          0          0         0        0    1
/usr/lib/x86_64-linux-gnu/                     60520       5020       1376        167       3556       1284        156         24         0        0  205
/SYSV00ce5741 (deleted)                            4          0          0          0          0          0          0          0         0        0    1
[stack]                                          132         44         44         44          0          0          0         44         0        0    1
[vvar]                                            16          0          0          0          0          0          0          0         0        0    1
[vdso]                                             8          4          0          0          4          0          0          0         0        0    1
-------------------------------------------------------------------------------------------------------------------------------------------------------
Total                                        8701848    8546064    8450173    8445755       7832     145324       2268    8390640       980        0  251
```

But How Much Memory are Connections Really Using?

credativ

- I did some playing with multiple sessions with/without parallelism
- Leading to different RSS numbers in top command -> Let's dive into smaps of all these sessions

```
## top command
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
190747 postgres  20   0 8701868   8.1g   8.1g S   0.0  26.2   0:48.69 postgres: postgres postgres 172.18.0.1(40278) idle
206119 postgres  20   0 8702808   4.7g   4.7g S   0.0  15.2   0:21.81 postgres: postgres postgres 172.18.0.1(44010) idle
219065 postgres  20   0 8802384   8.2g   8.1g S   0.0  26.6   2:56.64 postgres: postgres postgres 172.18.0.1(52912) idle
228832 postgres  20   0 8709912   3.4g   3.4g S   0.0  11.1   0:11.10 postgres: postgres postgres 172.18.0.1(60090) idle
230390 postgres  20   0 8701340   8.1g   8.1g S   0.0  26.3   0:51.89 postgres: postgres postgres 172.18.0.1(44802) idle

## smaps summaries with /dev/zero
Path                                      Size        Rss        Pss  Pss_Dirty  Shr_Clean  Shr_Dirty  Prv_Clean  Prv_Dirty      Swap  SwapPss  Cnt
-------------------------------------------------------------------------------------------------------------------------------------------------------
Total for /proc/190747/smaps           8701868    8529172    2196188    2195833       2960    8525332          0        880     61784     1116  256
Total for /proc/206119/smaps           8702808    4928096    1119095    1118712       3120    4924968          0          8     63996     2112  258
Total for /proc/219065/smaps           8802384    8635640    2296848    2295726       5472    8531892         12      98264     64896     4078  252
Total for /proc/228832/smaps           8709912    3609444     798269     795595       8660    3590600         60      10124     60936      100  252
Total for /proc/230390/smaps           8701340    8542712    2202431    2199069       8660    8531564        748       1740     60768       99  251
-------------------------------------------------------------------------------------------------------------------------------------------------------
                                      43618312   34285064    8611831    8613495      34872   34193356        820      19916    312480    17405 1279
## smaps summaries without /dev/zero
Path                                      Size        Rss        Pss  Pss_Dirty  Shr_Clean  Shr_Dirty  Prv_Clean  Prv_Dirty      Swap  SwapPss  Cnt
-------------------------------------------------------------------------------------------------------------------------------------------------------
Total for /proc/190747/smaps             77660       4416       1291        936       2960        576          0        880      3508     1116  255
Total for /proc/206119/smaps             78600       3708        448         65       3120        580          0          8      5720     2112  257
Total for /proc/219065/smaps            178176     104316      99427      98305       5472        584         12      98248      6620     4078  251
Total for /proc/228832/smaps             85704      19420      12853      10179       8660        576         60      10124      2660      100  251
Total for /proc/230390/smaps             77132      11716       5143       1781       8660        584        748       1724      2492       99  250
-------------------------------------------------------------------------------------------------------------------------------------------------------
                                        499272     169576     118162     110266      34872       2900        820      19984     18300    17405 1274
```

Where is work_mem hidden in memory usage?

credativ

- Where is work_mem used during query execution? I tested with JSONB data from GitHub events
- Table size 38 GB, 17.5 million rows, 1/3 of JSONB records toasted, lz4 compression, PG 17.2

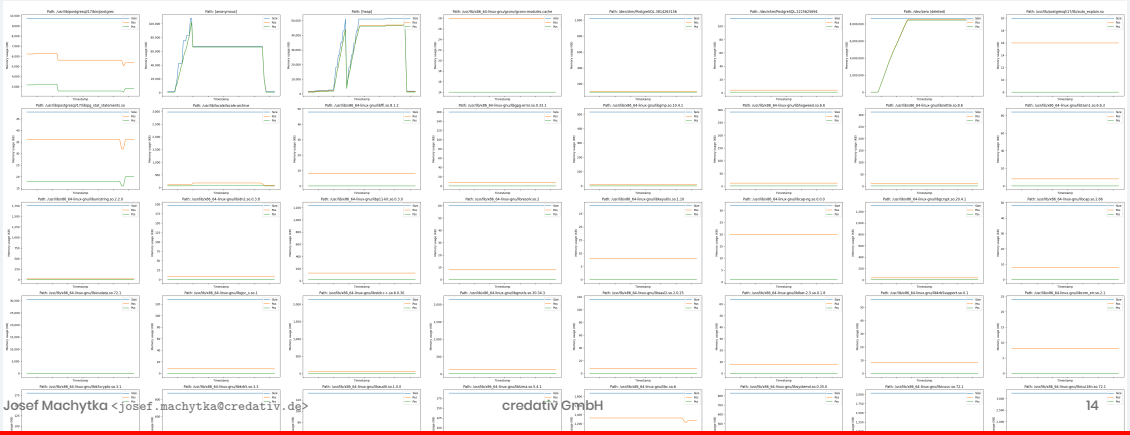
```
-- testing query
SELECT
    SUBSTR(jsonb_data->>'created_at'::TEXT,1,10) as created_at,
    count(*) as cnt
FROM github_events
GROUP BY 1 ORDER BY 1;

-- testing data
{   "id": "26167585827",
    "repo": {   "id": 581592468, "url": "https://api.github.com/repos/tiwabs/tiwabs_audio_door_tool",
                "name": "tiwabs/tiwabs_audio_door_tool" },
    "type": "PushEvent",
    "actor": {  "id": 48737497, "url": "https://api.github.com/users/tiwabs",
                "login": "tiwabs", "avatar_url": "https://avatars.githubusercontent.com/u/48737497?",
                "gravatar_id": "", "display_login": "tiwabs"  },
    "public": true,
    "payload": {"ref": "refs/heads/master", "head": "3ca247941f269bcedeb17e5b12e9b3b74b1c4da2",
                "size": 1, "before": "0dd5471667b12084b8fc88b1bca299780382d50a",
                "commits": [ {   "sha": "3ca247941f269bcedeb17e5b12e9b3b74b1c4da2",
                            "url": "https://api.github.com/repos/tiwabs/....12e9b3b74b1c4da2",
                            "author": { "name": "Tiwabs", "email": "mrskielz@gmail.com" },
                            "message": "fix(export): export nametable if export succed",
                            "distinct": true } ],
                "push_id": 12149772587, distinct_size": 1 },
    "created_at": "2023-01-01T13:39:55Z" }
```

Logging smaps for PostgreSQL process

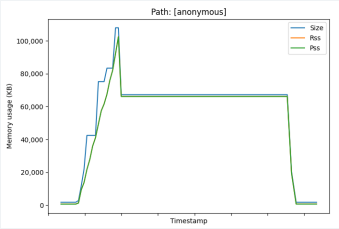
credativ

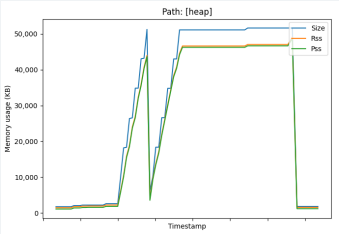
- Python script logs smaps data for PostgreSQL connections, roughly every 0.5 seconds
- RAM 32 GB, table 38 GB, shared_buffers 8 GB, work_mem 64 MB, toasted JSON, no parallel workers
- Query runs approx 70 seconds, after that we plot grid (Virt Size, RSS, PSS) for all paths

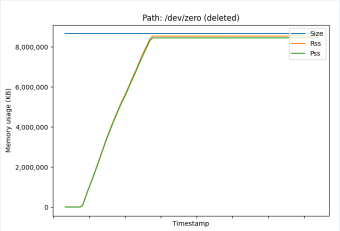


Analyzing logged smaps data

- All paths are mostly flat and un-interesting, except for 3 - [heap], [anonymous] and /dev/zero
- RAM 32 GB, table 38 GB, shared_buffers 8 GB, work_mem 64 MB, toasted JSON, no parallel workers

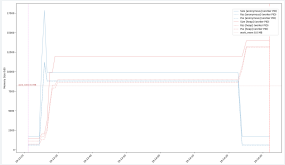






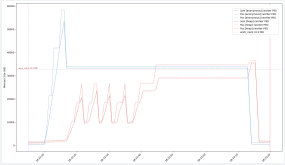
Smaps data - single process, work_mem 8 / 32 / 64 / 128 MB

work_mem = 8 MB



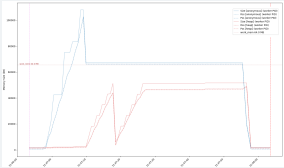
Max stacked RSS = 17.5 MB
(2.2x work_mem)
Sort Method: external merge
Disk: 307960kB

work_mem = 32 MB



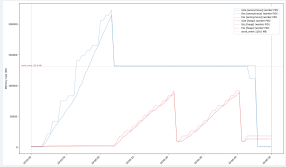
Max stacked RSS = 61 MB
(1.9x work_mem)
Sort Method: external merge
Disk: 307864kB

work_mem = 64 MB



Max stacked RSS = 110.6 MB
(1.7x work_mem)
Sort Method: external merge
Disk: 307822kB

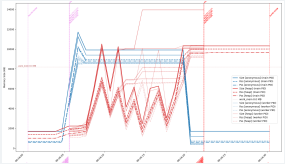
work_mem = 128 MB

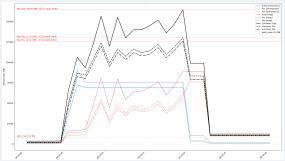


Max stacked RSS = 216 MB
(1.7x work_mem)
Sort Method: external merge
Disk: 307792kB

Parallel Query: main + 7 workes, work_mem 8 / 32 / 64 / 128 MB

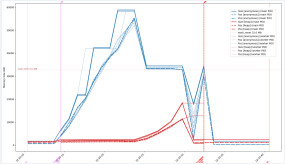
work_mem = 8 MB

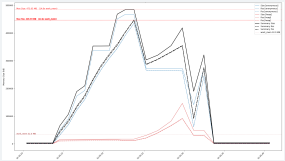




Max stacked RSS = 128 MB
(16x mork_mem)
Disk sorts 8x 38 MB

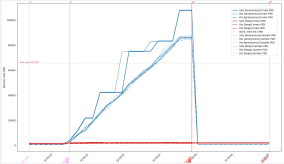
work_mem = 32 MB

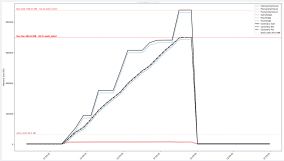




Max stacked RSS = 436 MB
(13.6x work_mem)
Disk sort 8x 38 MB

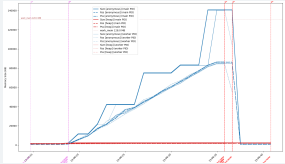
work_mem = 64 MB

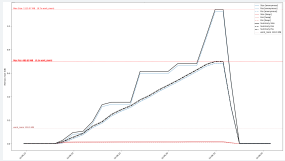




Max stacked RSS = 684 MB
(10.7x work_mem)
Memory sort 8x 64 MB

work_mem = 128 MB





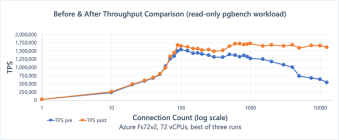
Max stacked RSS = 685 MB
(5.4x work_mem)
Memory sort 8x 96 MB

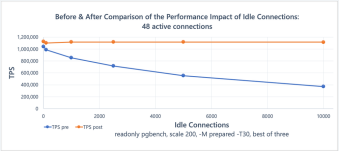
PostgreSQL connections are expensive

credativ

- MVCC snapshot management scaled very poorly in pre-v14
- Each session has transaction ID horizon / MVCC snapshot
- New transactions needed to check all sessions' snapshots
- Scans of Process Array require ProcArrayLock -> LWLock contention
- Big number of idle sessions degraded throughput of transactions/sec
- v14+ improved this significantly, throughput is now stable
- Each row has "visible since" (xmin) and "visible until" (xmax) XIDs
- Snapshot defines which XIDs are visible to transaction
- More connections -> more XIDs to track -> more overhead

- Improving Postgres Connection Scalability: Snapshots





PostgreSQL connections are expensive

- Many small queries lead to high random disk I/O
- If memory is exhausted, OS disk cache shrinks,
- Content of disk cache is evicted frequently
- Prefetching doesn't work well with smaller cache
- Prefetched pages often evicted before being used
- 2000 connection X 5 ops per second = 10,000 IOPS

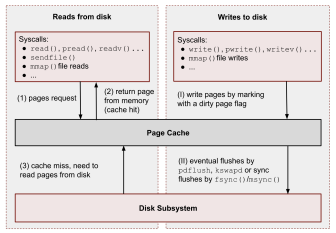


Image from the article Essential Page Cache Theory

How Many PostgreSQL Connections Are Too Many?

credativ

- max_connections -> memory footprint & file descriptors limit
- How many "active connections per core/thread" are reasonable?
- First of all - difference between CPU cores and threads
  - Modern CPUs have multiple cores
  - Core is physical execution unit
  - Each core can run multiple threads
  - -> Intel - Hyper-Threading Technology (HTT)
  - -> AMD SMT - Simultaneous Multi-Threading
  - Each thread shares core resources
  - -> one thread can be waiting for memory I/O
  - -> other thread can use core execution units
- Count of threads is typically 2x count of cores
- SMT/HTT can be disabled in BIOS/UEFI settings
- For PostgreSQL numbe of CPU cores is more important

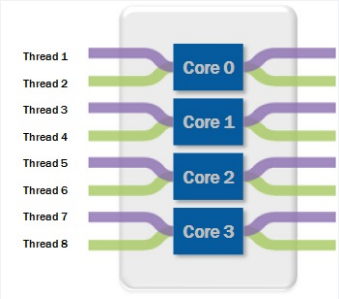


Image from article What Are Threads?

How Many PostgreSQL Connections Are Too Many?

- But how many ”active connections per core/thread”?
- -> depends on limitations of context switching mechanism
- Context switching bring overheads
  - APIC (Advanced Programmable Interrupt Controller) fires timer interrupt
  - APIC is integrated into each individual CPU core or thread
  - Intel APICv / AMD AVIC reduce virtualization overhead -> interrupt offloading
  - On interrupt, CPU saves current process state (context)
  - CPU loads new process state from memory
  - Process state includes CPU registers, program counter, stack pointer
  - TLB (Translation Lookaside Buffer) may need full or partial flush
  - -> newly TLB entries tagged with process IDs

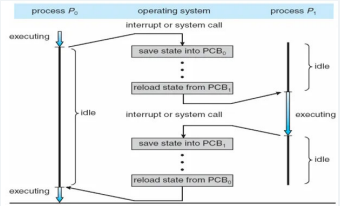


Image from the article Context Switching in Depth

How Many PostgreSQL Connections Are Too Many?

credativ

- Completely Fair Scheduler (CFS)
  - -> original scheduler used in Linux 2.6+
  - -> "ideal multi-tasking" - N processes, 1/N speed
  - -> Every process gets an equal share of CPU time
  - -> in kernel 6.6+ replaced with EEVDF scheduler
- Earliest eligible virtual deadline first (EEVDF)
  - -> newer scheduler, since October 2023
  - -> introduced small, more frequent bursts of CPU time
  - -> can prioritize interactive or latency critical tasks
  - -> but still should keep fair share of total CPU time
- Ubuntu uses 6.6+ from v24.04
- Debian 12 uses 6.1 -> Debian 13 jumps to 6.12
- RedHat Enterprise 9 uses 5.14 -> v 10 jumps to 6.12
- 6.12 contains many improvements and patches against 6.6+

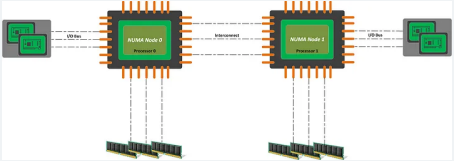
Completely Fair Scheduler vs Earliest deadline first

credativ

- What switch from CFS to EEVDF means for PostgreSQL connections?
- Typical PG connections are often ”sleeping” - network, data fetch, lock wait
- Some connections are ”bursty” - many small transactions
- CFS can sometimes penalize ”bursty” workloads
  - -> Context switching storm by waking up many sleeping processes
  - -> maintains ”average fairness” for all processes
  - -> uniformity, no latency preference, only some preemption for ”sleepers”
- EEVDF can prioritize latency critical tasks
  - -> ”bursty” workloads can show better latency - will be processed earlier
  - -> waiting processes should not slow down bursty workloads
  - -> overrunning CPU-heavy workloads will not starve other processes
  - -> most likely will improve mixed workloads (small/long transactions)
  - -> but OLAP only workloads might slightly suffer from more context switches (without tuning)
  - -> for OLAP we can use SCHED_BATCH policy - sacrifices responsiveness for throughput

PostgreSQL and NUMA Architectures

- NUMA (Non-Uniform Memory Access) architectures - divides system into nodes
- Each node has its own local memory and CPU cores
- Modern servers often have multiple NUMA nodes
- Accessing local memory is faster than remote memory in other nodes
- CPU can access remote memory, but with higher latency -> NUMA penalty - 2x-3x slower
- Linux kernel offers NUMA balancing mechanisms
  - -> automatic memory pages migration or migration of processes between nodes
  - -> can be tuned via /proc/sys/kernel/numa_balancing



PostgreSQL and NUMA Architectures

- PostgreSQL allocates shared buffers in local memory of the node where PG process started
  - -> parallel queries, background processes might access remote memory -> performance degradation
  - -> context switches between processes on different nodes -> TLB flushes, performance degradation
  - -> high number of connections increases chances of cross-node memory access
- Only PostgreSQL 18 has optimizations for NUMA architectures
  - pg_numa_available() - checks if PG was compiled with NUMA support
  - pg_shmem_allocations_numa view - tracks shared memory allocations per NUMA node
  - pg_buffercache -> pg_buffercache_numa view - tracks buffer usage per NUMA node
- -> Monster NUMA systems with many nodes can be challenging for PG performance

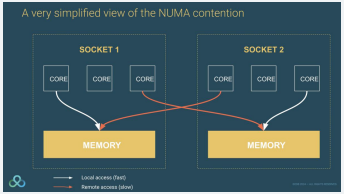


Image from article Postgres in the time of monster hardware

How Many PostgreSQL Connections Are Too Many?

credativ

- Another bottleneck - too many locked objects in the session
- PG 9.2 - v17: hard coded setting FP_LOCK_SLOTS_PER_BACKEND = 16
  - -> capacity of "fast-path" locking mechanism - relations locks are acquired locally
  - -> locks over this capacity require accessing central shared lock table
  - -> slows down the system - contention on shared lock table
  - -> increases latency linearly, planning time grows accordingly
  - -> causes huge spikes of "LWLock" waits in pg_stat_activity
  - -> good enough for 2010s query patterns with typically few tables per query
  - -> limiting for complex schemas of 2020s with many tables/indexes per query
  - -> partitioning made it much worse for some types of workloads
- PG 18 increases this limit to max_locks_per_transaction setting (default 64)
- -> planning time and latency seems to be more or less flat, no linear increase

How Many PostgreSQL Connections Are Too Many?

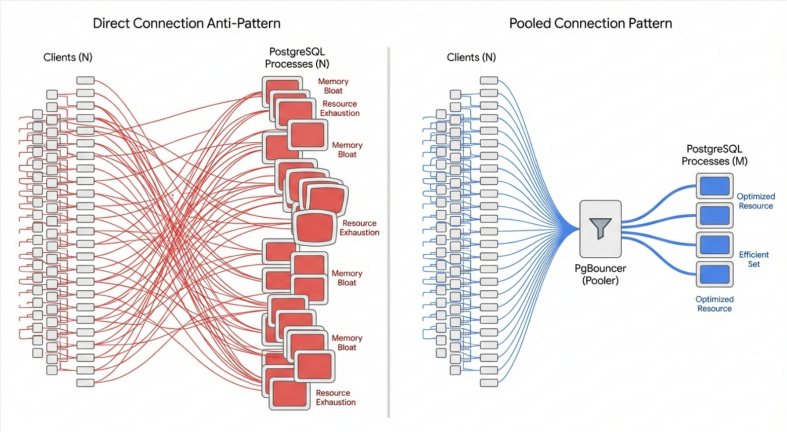
credativ

- Rule of thumb: 2 - 4 active OLTP connections per 1 CPU core
- -> applies only for OLTP workloads on NVMe/SSD storage with huge pages enabled
- -> sweet spot seems to be 3 active connections per 1 core
- -> down to 1-2 active connections per 1 core for HDD storage or without huge pages

- OLAP workloads without parallelism -> 1 connection per 1 core
- OLAP workloads with parallelism -> 1 connection per [1+workers] cores

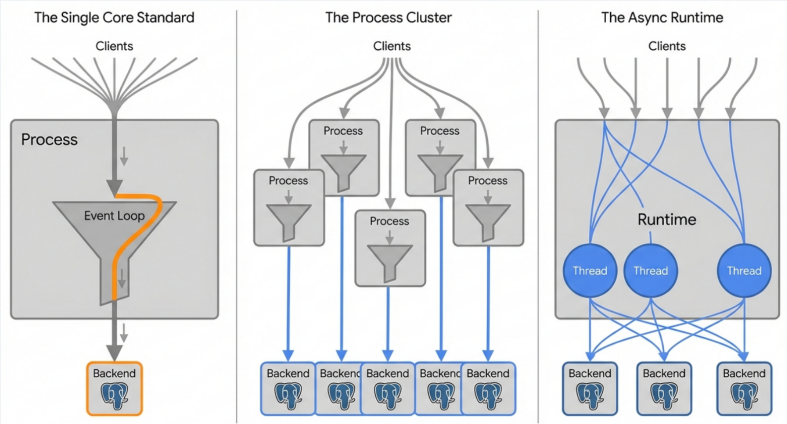
- -> Use PostgreSQL 14+ - allows linear scaling of MVCC
- -> Use PostgreSQL 18+ due to improved relation locking mechanism

Why Connection Pooling?



Picture created by Gemini Pro 3.0 AI Model

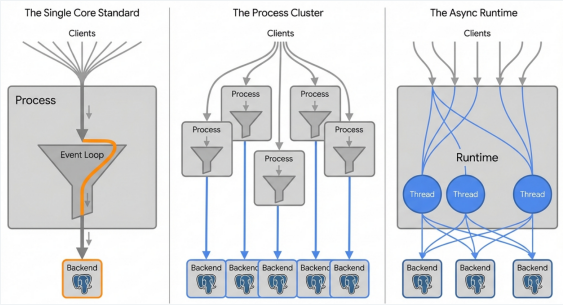
Connection Poolers Architectures



Picture created by Gemini Pro 3.0 AI Model

Connection Poolers Architectures

- ”Law of Large Numbers” applies to connection pooling:
  - Global pool better utilized than per-worker pools
  - Pool of 100 conn handles bursts better than 4 pools of 25
  - The more and smaller the pools, the worse the utilization
  - ”We cannot predict load per worker, but we can predict total load”



Thundering Herd Problem

- Applies to connection poolers with per-worker pools
- Pattern of non-linear performance degradation
  - Client sends SYN packet to listening port
  - Kernel receives packet, answers with SYN-ACK
  - Client sends ACK to complete TCP handshake
  - Sleeping clients waiting in accept() system call
  - Kernel wakes up one or more waiting worker processes
  - All woken workers compete to accept the new connection
  - Only one worker can accept, others go back to sleep
  - Woken workers waste CPU cycles in context switches
- Single-threaded poolers (like PgBouncer) not affected
- But single-threaded (1 CPU only) pooler can be bottleneck

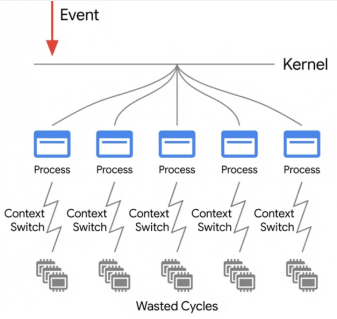


Image created by Gemini Pro 3.0 AI Model

Thundering Herd Problem

- On kernel level, solved by ”accept mutex” - serializes accept() calls
- When serializes, only one worker woken up to handle new connection
- SO_REUSEPORT socket option can divide load between multiple sockets
- Not aware of load per worker, some workers can be overloaded (Law of Large Numbers)

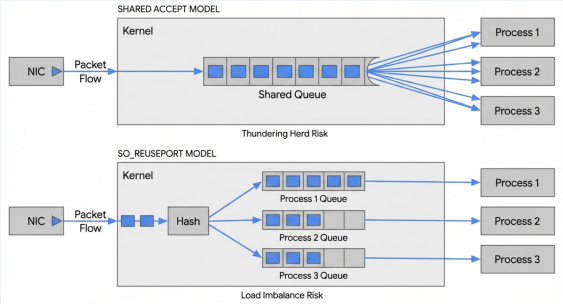


Image created by Gemini Pro 3.0 AI Model

Level-Triggered vs Edge-Triggered I/O

- Level-Triggered I/O (LT):
  - Default mode for most I/O multiplexing APIs (select, poll)
  - Notifies application when data is available to read/write
  - If application does not read all data, will be notified again
  - Simpler to implement, but can lead to inefficiencies
- Edge-Triggered I/O (ET):
  - Notifies application only when new data arrives
  - Application must read/write all available data in one go
  - More complex to implement, but can be more efficient
  - Reduces number of notifications and context switches

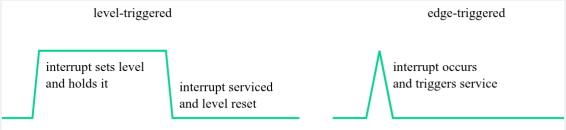


Image from article Tracing Optimized Edge-Triggered Systems

credativ

Connection Poolers

Connection Poolers

credativ

- **pgbouncer**: Lightweight connection pooler for PostgreSQL
- **Pgpool-II**: More advanced, feature-rich
- **PgCat**: Modern Rust-based, ”pgbouncer on steroids”
- **pgagroal**: protocol-native, high-performance
- **Supavisor**: massive scalability - millions of connections
- **Odyssey**: New, high-performance, multi-threaded pooler

PgBouncer: Ultra Lightweight Pooler

credativ

- Ultra lightweight pooling daemon for PostgreSQL
- Programmed in C, Open-source, first released in 2007
- Developed and maintained by pgbouncer.org community
- Listens on TCP and/or Unix domain sockets
- Widely used, easy configuration
- Very low memory footprint (around 2KB per conn), very low overhead and latency
- Easiest, simplest to deploy connection pooler for PostgreSQL
- In current version still only single-threaded architecture
  - Can become bottleneck under very high conn counts
  - New multi-threaded versions in development
  - Multiple workarounds possible with existing version

Pgpool-II: The Swiss Army Knife

- Open-source, originally personal project, published 2006
- Now maintained by Pgpool Global Development Group
- Complex middleware with rich features
- Comprehensive DB cluster management tool
- Higher overhead than pgbouncer, multiprocess architecture
- Global idle in transaction timeout (!)
- Parent process + 32 child processes

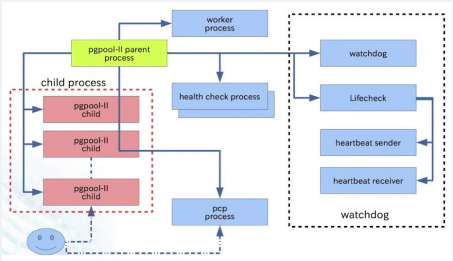
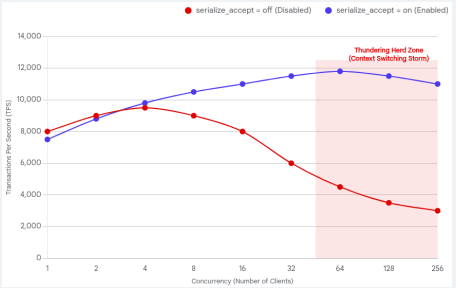


Image from talk Pgpool-II Performance and best practices *

Pgpool-II: The Swiss Army Knife

- Parent process + 32 child processes
- Each child has its own pool of server connections
- Each child process can handle only one client at a time
- Once connected, client is handled by the same child process
- Setting ”serialize_accept” = ON
- -> only one child woken to execute accept()
- -> prevents thundering herd problem on new connections



Performance degradation due to context switching storms

Pgpool-II: The Swiss Army Knife

- In memory query caching
  - Caches SELECT query & results in pooler memory
  - Subsequent identical queries served from cache
  - Does not parse queries, only text matching
  - Cache invalidated on table updates
  - Whitelists / blacklists for tables for caching
  - Best for read-heavy workloads
  - Higher overhead for often updated tables

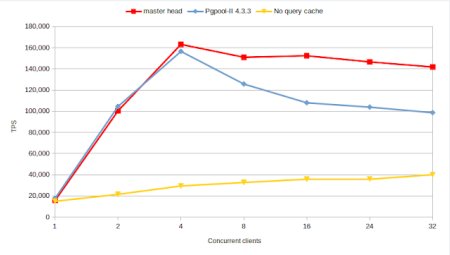


Image from article
Playing with PostgreSQL and Pgpool

Pgpool-II: The Swiss Army Knife

- Project offers pgpool2_exporter for Prometheus
- Load balancing between multiple PG servers
- Uses PG parser for context-aware routing
- Transparent replication support
  - Directs write queries to master node
  - Read queries go to replicas
  - currval()/nextval() routed to master

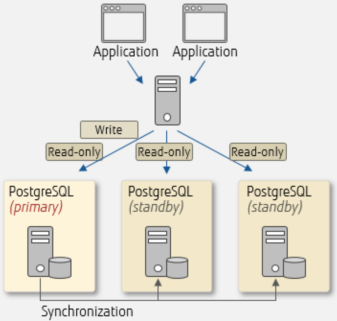


Image from article Pgpool-II with PostgreSQL

Pgpool-II: The Swiss Army Knife

- Transaction pooling mode only
- Always tracks transaction state
- Queries in open transaction routed to same conn
- Temporary tables supported -> stay on same conn
- Can reuse server connection for same client
- High Availability features
  - Can perform automatic failover of DB nodes
  - Can promote replica to master role
  - Can remove failed DB nodes from pool
  - Multiple pgpool-II instances with Watchdog
  - Can start recovery of failed DB node and re-add to pool

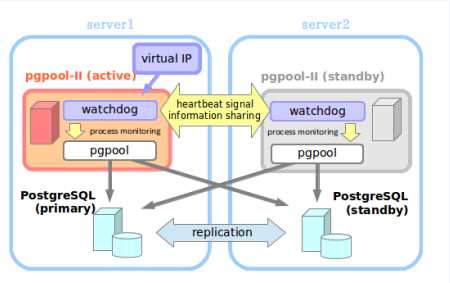


Image from Pgpool-II Documentation

Pgcat: Modern Rust Proxy

credativ

- ”Nextgen” PostgreSQLconnection pooler, written in Rust
- Open-source, first stable release in 2023
- Session / Transaction / Statement pooling
- Load balancing reads between replicas
- Automatic failover support for failed nodes
- Health checks for PostgreSQL nodes
- Not answering nodes removed from pool
- Queries automatically balanced between replicas and primary
- Enhanced transaction pooling mode to handle abandoned transactions
- Special ”pgbouncer” and ”pgcat” databases for monitoring
- Built-in Prometheus metrics endpoint

Pgcat: Modern Rust Proxy

credativ

- Prepared statements NOT supported in transaction pooling mode
- Load balancing using either random or least-connections algorithms
- Reads and writes detected using parser - routed to replicas / primary
- Custom routing SQL syntax "SET SERVER ROLE TO" (primary/replica/any/auto)
- All DB connections checked with fast query before given to client
- Interesting experimental features:
  - - Sharding using extended SQL syntax
  - -> "SET SHARD TO <shard_id>"
  - -> "SET SHARDING KEY TO <value>"
  - - Sharding using comments parsing
  - -> /* shard_id: value */ SELECT ...
  - -> /* sharding_key: value */ SELECT ...
  - - Automatic detection of sharding keys
  - - Mirroring of traffic to multiple DB nodes for testing with real traffic

Pgcat: Modern Rust Proxy

- Multithreaded architecture - Tokio async runtime
- Uses Reactor-Executor concurrency model
- Reactor handles I/O and network events asynchronously
- Thread does not wait in read() event for data
- Async access thread only tries to see if data is ready
- -> if not ready, attempt fails fast, thread can do other work
- Implements M:N threading model - M tasks over N threads
- Uses Edge-triggered kernel event notifications
- -> reduces number of syscalls and context switches
- -> improves performance for thousands of connections
- Client connection encapsulated as a ”task”
- -> Similar to threads, but managed in user space

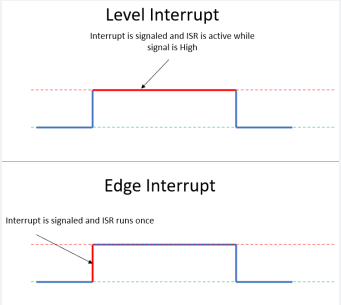


Image from article Kernel Interrupts

Pgcat: Modern Rust Proxy

- Tokio runtime uses work-stealing scheduler
- Each worker thread has its own task queue
- If thread idle, can ”steal” tasks from other threads
- Improves CPU utilization and load balancing
- Pgcat scales well with increasing CPU cores
- -> Global queue for incoming tasks
- -> Local queues on each worker thread
- -> Idle threads steal tasks from busier threads

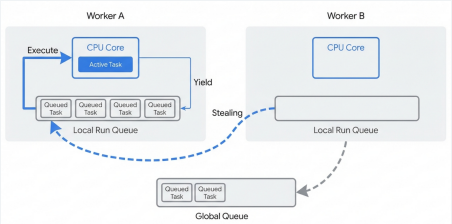
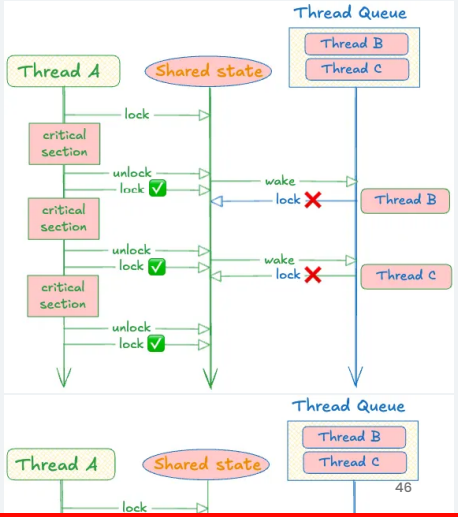


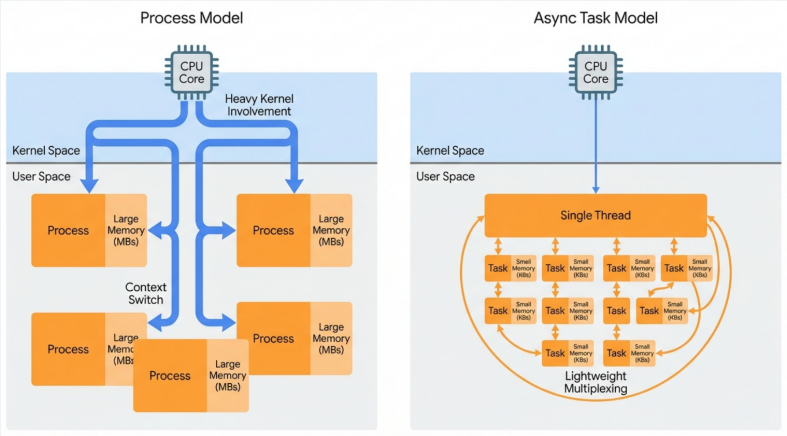
Image created by Gemini Pro 3.0 AI Model

Pgcat: Modern Rust Proxy

- PostgreSQL connection requires synchronous operations
- Pgcat uses ”parking_lot” synchronization crate
- -> thread attempts to acquire lock on mutex
- -> if already locked, spins for a few iterations
- -> repeatedly checks if lock is free
- -> if still locked, thread ”parks” itself
- -> each thread has lock only for short time
- -> reduces context switches and improves throughput
- -> ”handoff” prevents starvation of other threads



Connection Poolers Concurrency Models



Picture created by Gemini Pro 3.0 AI Model

Odyssey: The Performance Specialist

credativ

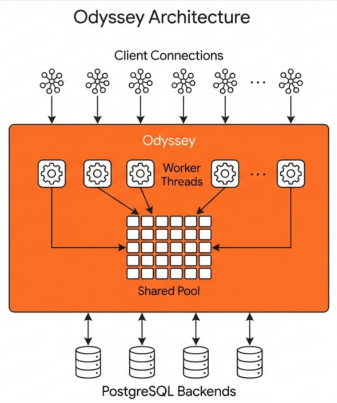
- Open-source, developed by Yandex (Russian company)
- Multi-threaded architecture - worker threads
- Tens of thousands of connections
- Session or Transaction pooling modes
- Advanced transaction state tracking
  - Tracks current transaction state of connections
  - Can cancel trans on server side if client crashes
  - Can rollback abandoned transactions
- Last connection owner is remembered
- Owner can reuse session to limits setting up environment
- Pool per DB and user - separate auth, pooling mode, limits
- Best for env with very high concurrency & multi-core CPUs
- Contains native Prometheus metrics endpoint

Odyssey repository

Odyssey website

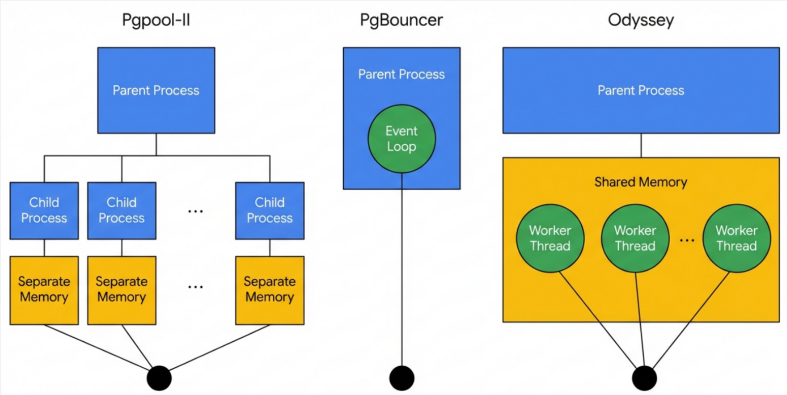
Odyssey: The Performance Specialist

- Scales linearly with CPU cores by adding more workers
- Each worker has its own set of client connections
- All workers share same global DB connection pool
- Custom library Machinarium for async I/O
  - Read does not block the thread
  - If data not ready, library saves status of coroutine
  - Switches to another coroutine ready to do work



Picture created by Gemini Pro 3.0 AI Model

Connection Poolers Concurrency Models



Picture created by Gemini Pro 3.0 AI Model

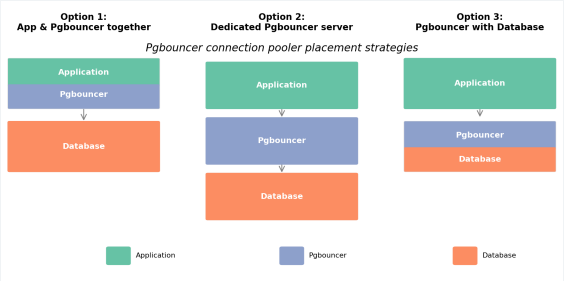
Connection Poolers - Feature Matrix

credativ

Feature	PgBouncer	Pgpool-II	Odyssey	PgCat	Supavisor
Transaction Pooling	✅ Yes	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Prepared Statements (Tx Mode)	⚠️ Protocol level	❌ No	✅ Yes	✅ Yes	✅ Yes
Multi-threaded Architecture	❌ No (Single)	❌ No (Fork)	✅ Yes	✅ Yes	✅ Yes (Actor)
Auto-Failover	❌ No	✅ Yes	❌ No	✅ Yes	⚠️ Planned
Sharding Support	❌ No	❌ No	❌ No	✅ Yes	❌ No
Read/Write Load Balancing	❌ No - conf split	✅ Yes	❌ No	✅ Yes	✅ Yes
Zero-Downtime Reload	✅ Yes	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Native Prometheus Metrics	⚠️ External	⚠️ External	✅ Yes	✅ Yes	✅ Yes
SCRAM-SHA-256 Auth	✅ Yes	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Multi-Tenancy (High Scale)	❌ No	❌ No	⚠️ Partial	✅ Yes	✅ Yes
TLS Termination	✅ Yes	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Query Cancellation	✅ Yes	✅ Yes	✅ Yes	✅ Yes	✅ Yes

Where to run Connection Poolers?

- On application servers:
  - Reduces network traffic to database server
  - Each app server has its own pool
  - But consumes resources on app servers
- On separate dedicated pooling servers
  - Centralized pooling for multiple app servers
  - Simplifies management
  - Requires additional network hop
  - Adds network latency and bandwidth cost
- On database server
  - Centralized pooling
  - But adds load to database server



PostgreSQL Connection Poolers Resources

credativ

- Percona talk ”Elephant by the pool”
- PGConf Europe 2024: Comparing Connection Poolers For PostgreSQL
- PgBouncer vs Pgcat vs Odyssey on VPS in 2025
- Postgres connection pooling: Comparing PgCat and pgbouncer
- POSETTE 2024: PostgreSQL Connection Poolers Comparison
- Use Connection Pooling to Enable Postgres Proxy and to Improve Database Performance

Thank you for your attention!

credativ

End of Part 1

To Be Continued...