SDC 20

BY Developers FOR Developers

Storage Developer Conference
September 22-23, 2020

How can persistent memory make database faster, and how could we go ahead?

Takashi Menjo
NTT Software Innovation Center
(Nippon Telegraph and Telephone Corporation)

Persistent memory and PostgreSQL

Persistent memory (PMEM)

- Byte-addressable and non-volatile
- Faster-to-access than disk
- Available on major operating systems
- Encouraged by PMDK, a dedicated library
- Application? Database! PostgreSQL!!

PostgreSQL

- Relational database management system
  - Represents data as two-dimensional tables
- Disk-based
  - Stores tables and indices on disk
- Open-source
  - Developed by many contributors including NTT

PostgreSQL getting faster with PMEM

How we've tried to make it PMEM-aware:

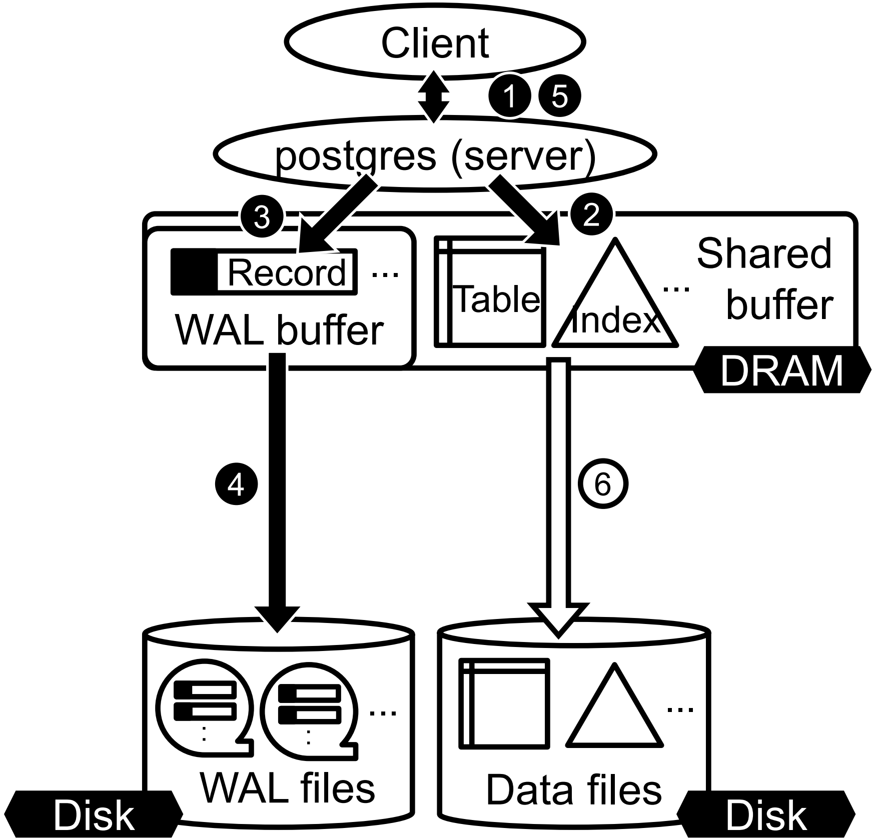
- Use PMEM as an alternative of disk
  - Straightforward, but not really "PMEM-aware" yet
- Replace file I/O API with PMDK
  - Simple, good for middle step, but less effective
- Re-design its architecture
  - Challenging, but more effective

Outlines

- Persistent memory and PostgreSQL
- PostgreSQL’s architecture and its re-design
- Evaluation setup
- Performance and profile results
- How could we go ahead?

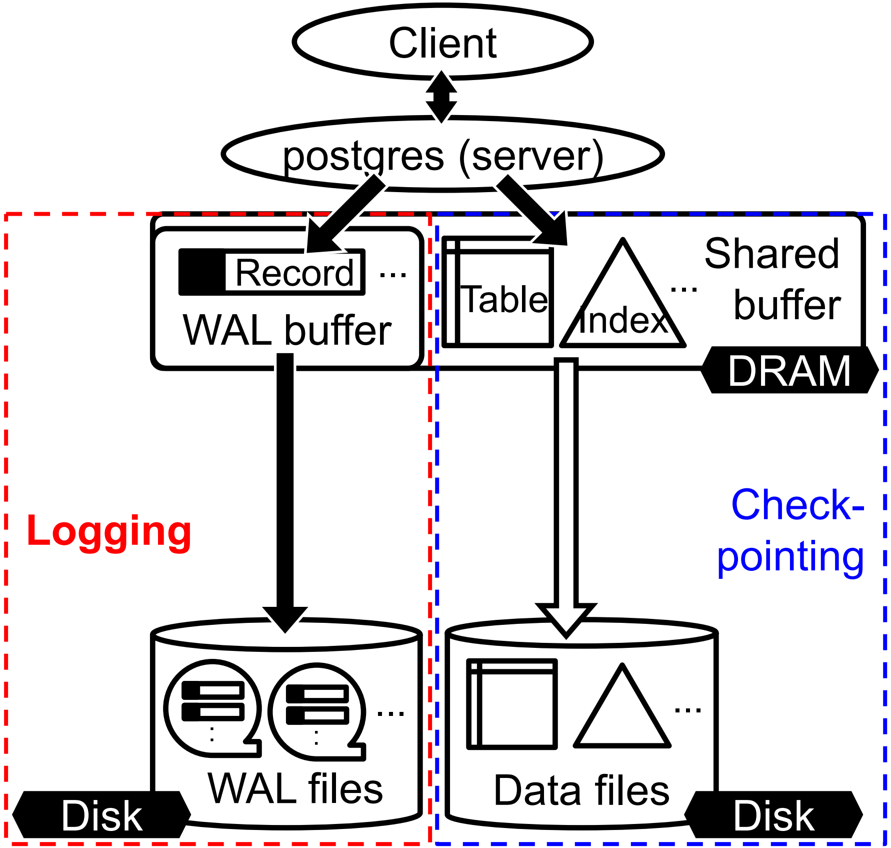
PostgreSQL’s architecture and its re-design

Overall architecture



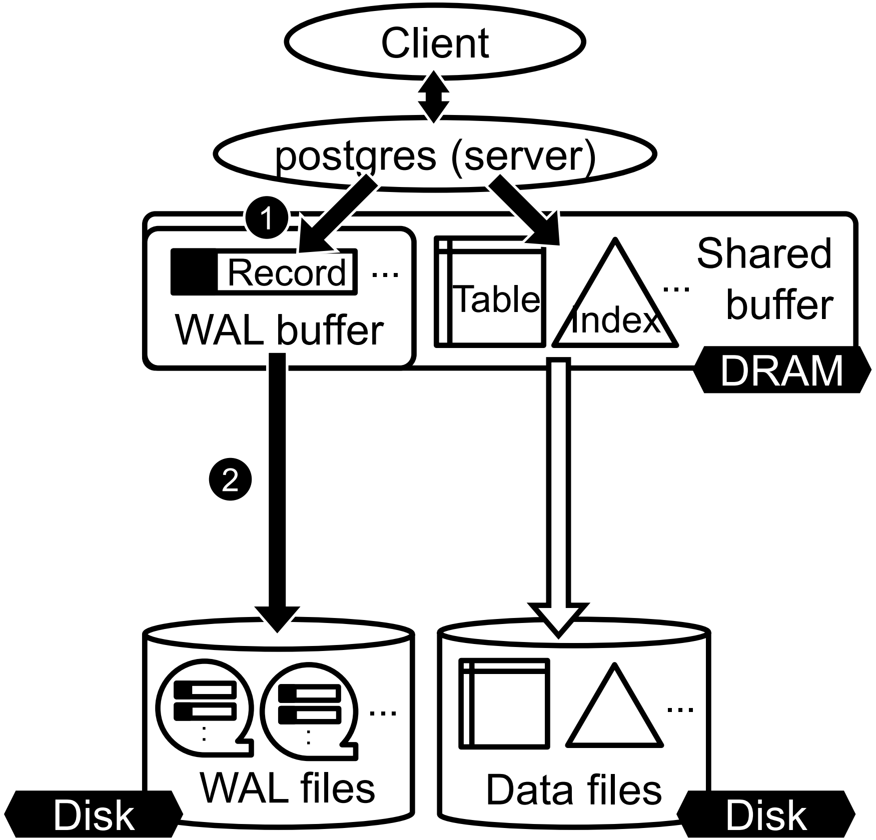
1. Receives a transaction from a client
2. Updates tables etc. to process the transaction
3. Stores transaction log records (aka Write-Ahead Log) to the ring buffer called "WAL buffer"
4. On commit, writes the records out to WAL files **synchronously**, that is, **waits until the records hit to disk**
5. Sends an acknowledgement to the client to finish the transaction
6. **On checkpoint**, writes tables etc. back to data files

Logging and checkpointing



- **Logging** is more critical than checkpointing for transaction performance
  - Logging is **synchronous** with transaction processing, while checkpointing is asynchronous
- We've focused to improve logging by using PMEM

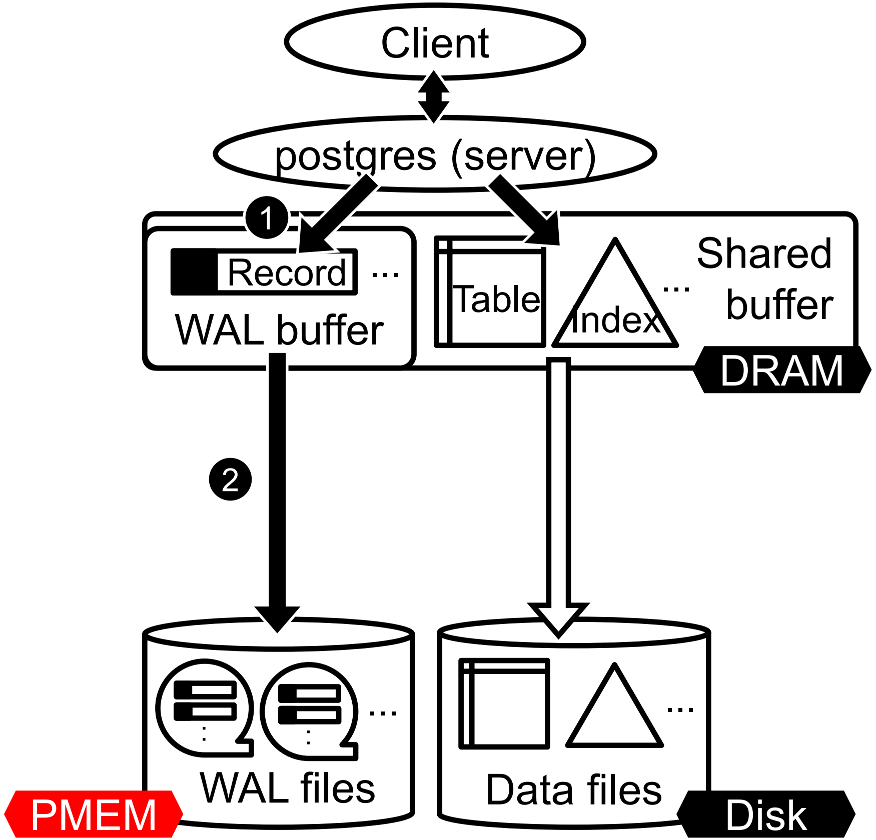
Two-level logging architecture



1. Buffers WAL on DRAM; then
2. Writes them out to disk

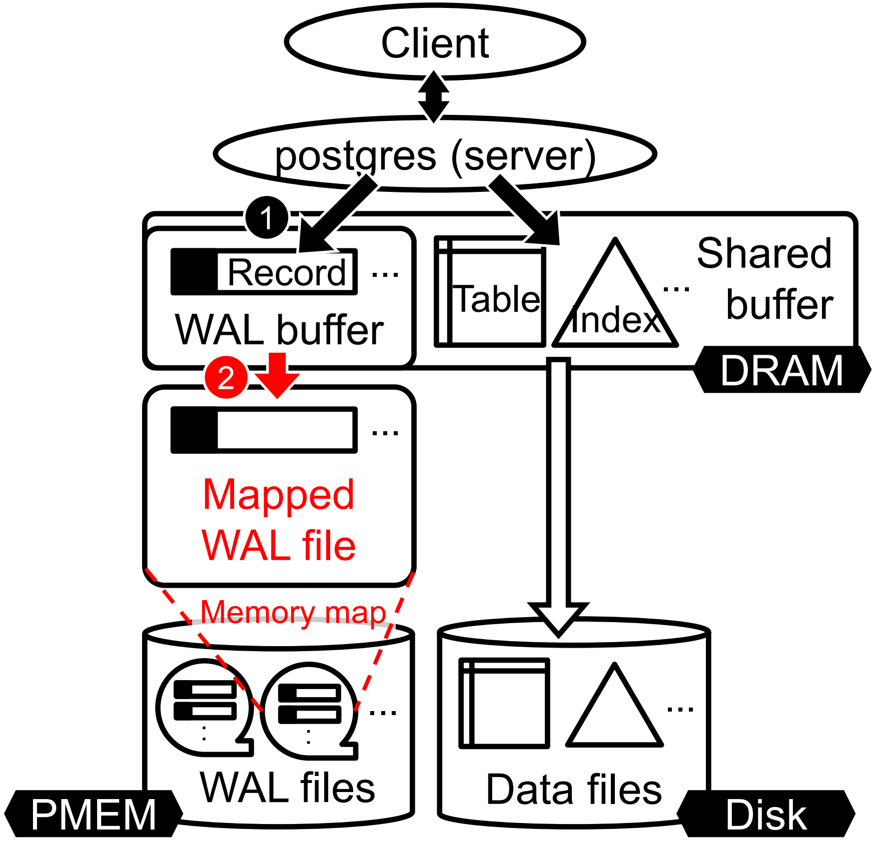
- Gains much performance when using disk
  - Disk is worse at small and/or random access than DRAM
  - Buffering makes large and sequential access which is suitable for disk

Using PMEM as an alternative of disk



- Create a **PMEM-aware filesystem** on PMEM
  - Bypasses page cache
- Use PostgreSQL **as is**
  - Calls the same file API as before, such as lseek, write, fdatasync etc.
- Not a PMEM-aware application yet, and still two-level logging

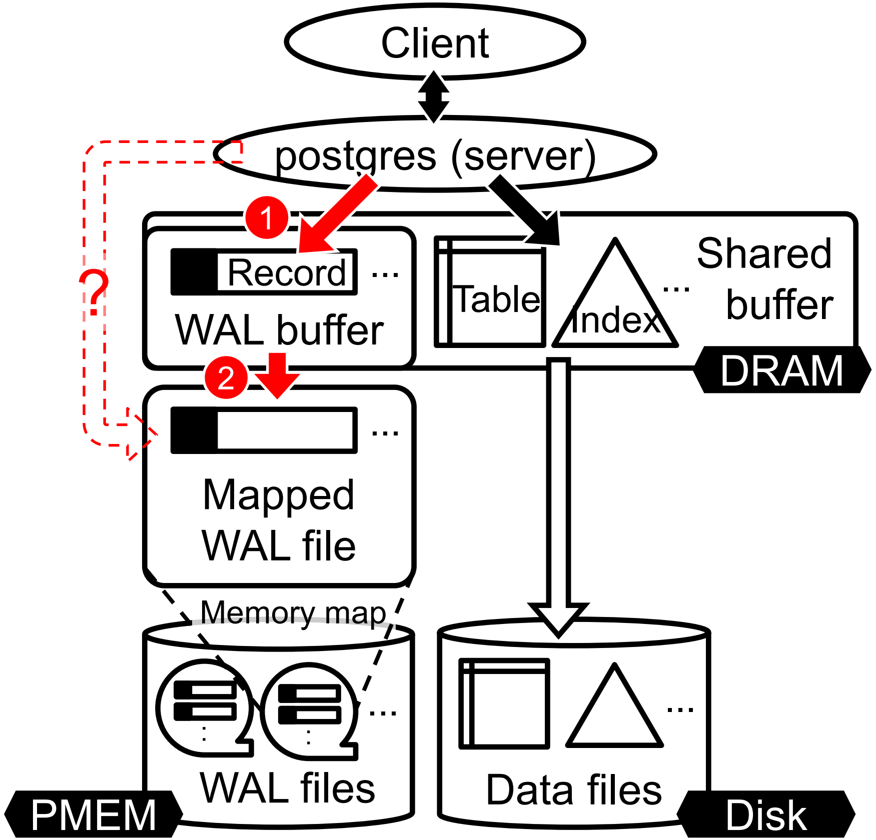
Replacing file I/O API with PMDK



(Yoshimi talked in SDC 2018)

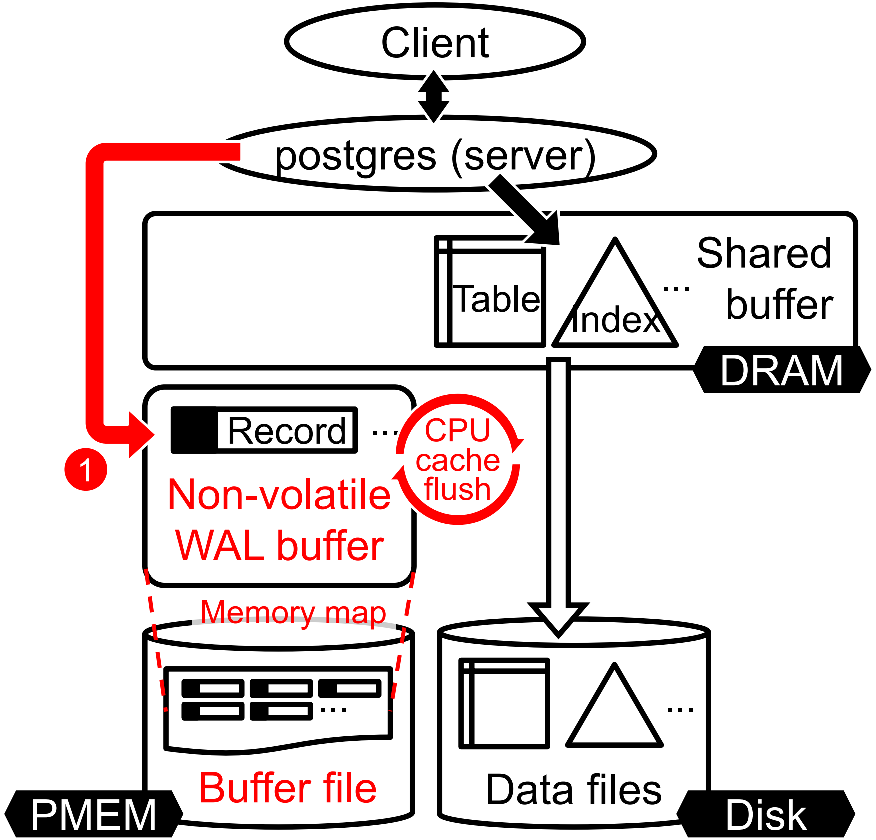
- Map WAL files onto memory
  - Bypass page cache by using DAX feature of PMEM-aware filesystem
- Use memory copy, CPU cache flush, and memory barrier functions instead of file API
- PMEM-aware application, but still two-level logging

An idea for architecture re-design



- Is it necessary to store the same WAL twice when using PMEM?
- Why not store WAL directly to PMEM to improve performance by reducing their copy to one?
  - It could be possible because PMEM is byte-addressable, non-volatile, and **better at small and random access than disk**

Single-level logging architecture



- Make and map a buffer file as **non-volatile WAL buffer**
- Store WAL directly to that buffer
  - Need only one copy for each log record
- On commit, flush CPU cache for those WAL to let them persist onto PMEM

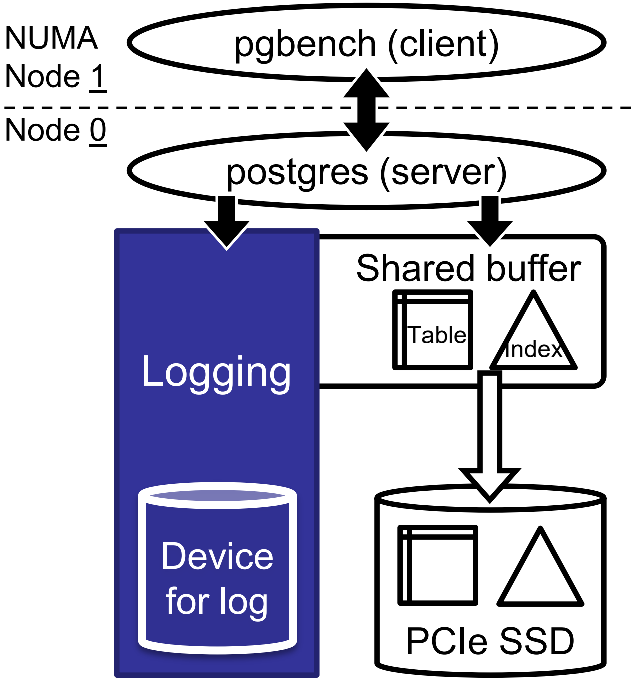
Evaluation setup

Purpose and method

- Evaluate our re-design on the following points:
  - Transaction performance
  - Performance profile

- Use pgbench and VTune Profiler to measure…
  - Transaction throughput and average latency
  - CPU time of each function, especially logging one
    - For a certain amount of transactions
    - For a certain benchmark duration (30 minutes)

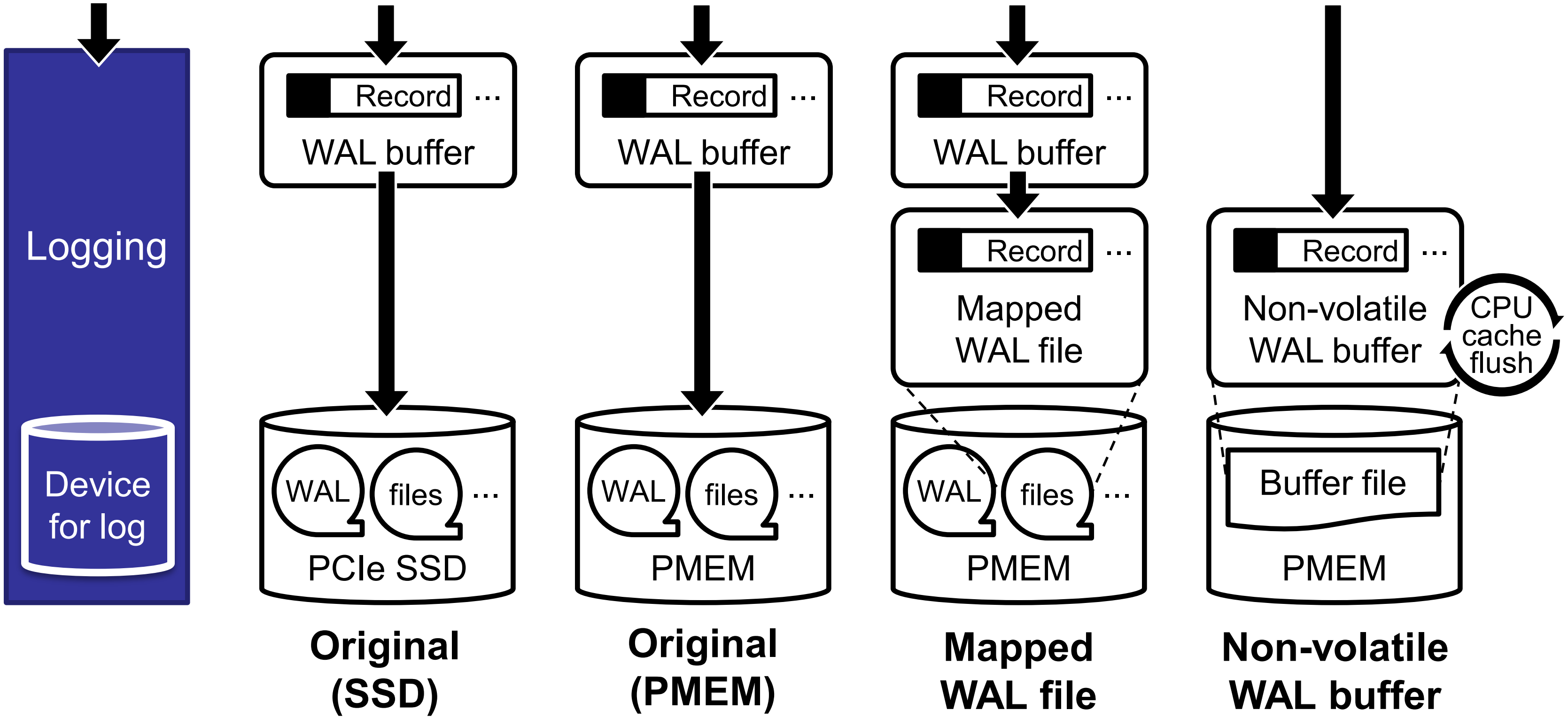
Setup (1/3)



For all setups:

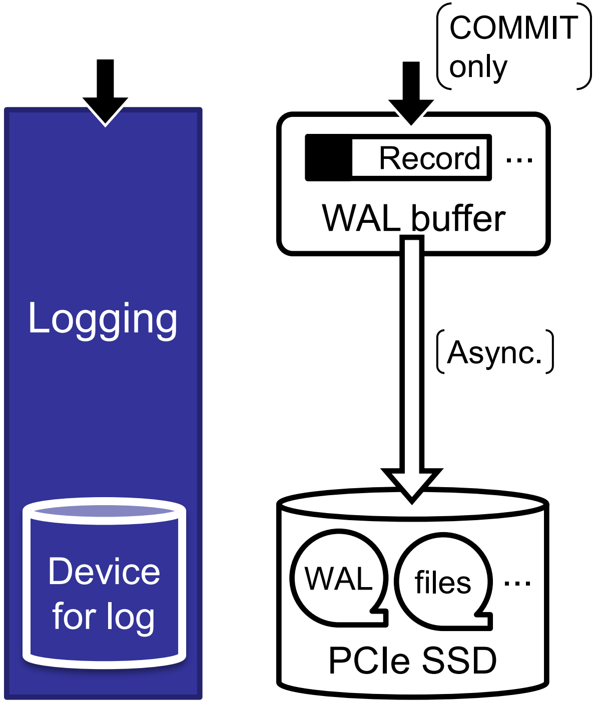
- Run server and client processes on the same machine but distinct NUMA nodes to get stable performance
- Store logs in a device for log (PCIe SSD or PMEM), and tables etc. in a separate SSD to reduce the impact of checkpointing
- Choose several values for the degrees of parallelism of pgbench (# connections and # threads, reffered to as “c” and “j”) to see performance variation

Setup (2/3)



Logging
Device for log
Record …
WAL buffer
WAL files …
PCIe SSD
Original (SSD)
Record …
WAL buffer
WAL files …
PMEM
Original (PMEM)
Record …
WAL buffer
Record …
Mapped WAL file
WAL files …
PMEM
Mapped WAL file
Record …
Non-volatile WAL buffer
CPU cache flush
Buffer file
PMEM
Non-volatile WAL buffer

Setup (3/3)



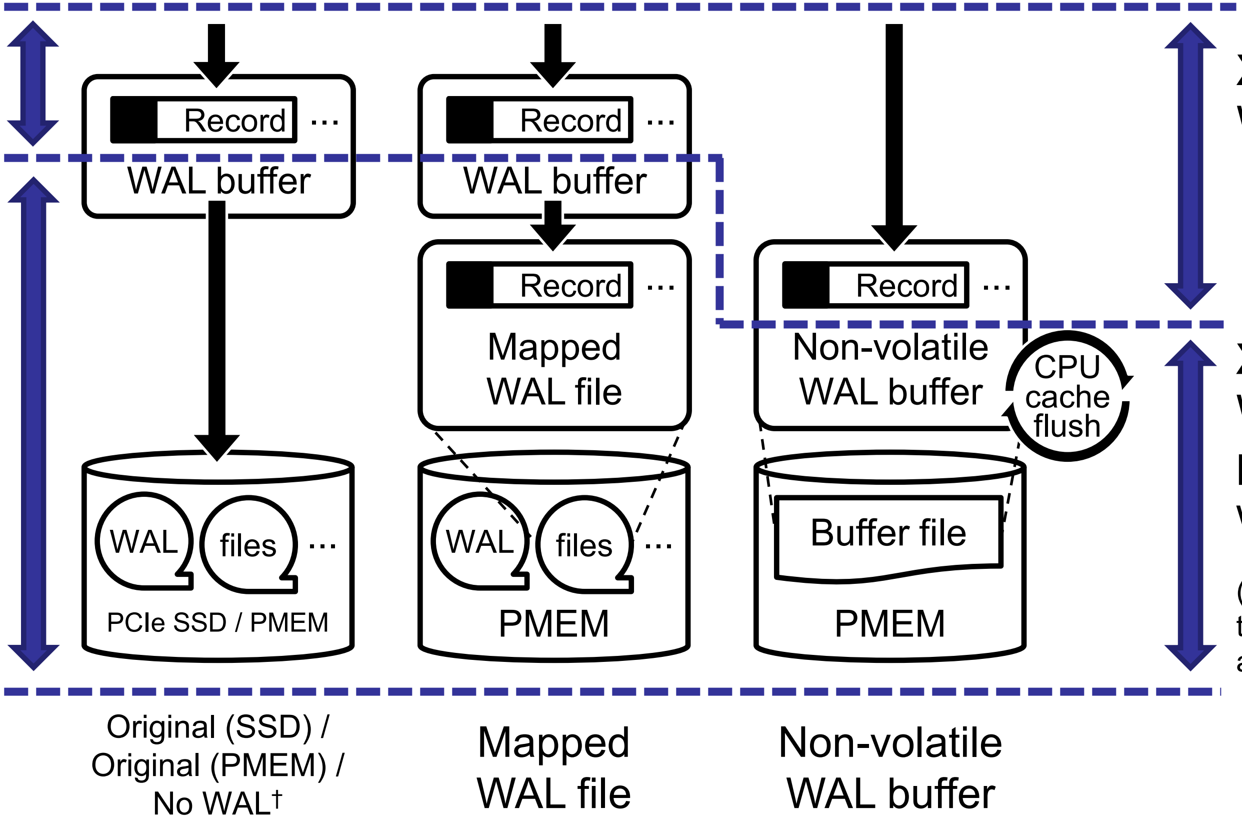
No WAL

“No WAL” setup:

- Suppress logging AFAP by using the following PostgreSQL’s features:
  - **Unlogged table** to store <u>no WAL</u> except COMMIT records
  - **Asynchronous commit** to send ack. to the clients <u>before</u> writing WAL
- Used as a reference of “upper-limit performance”
  - Not durable, but probably the fastest of the five setups

(*NVDIMM-N or Optane PMem)

Logging functions



XLogInsert stores WAL to the buffer

XLogFlush makes WAL durable onto persistent device when committing

(† "No WAL" spends no time for this function because it uses asynchronous commit)

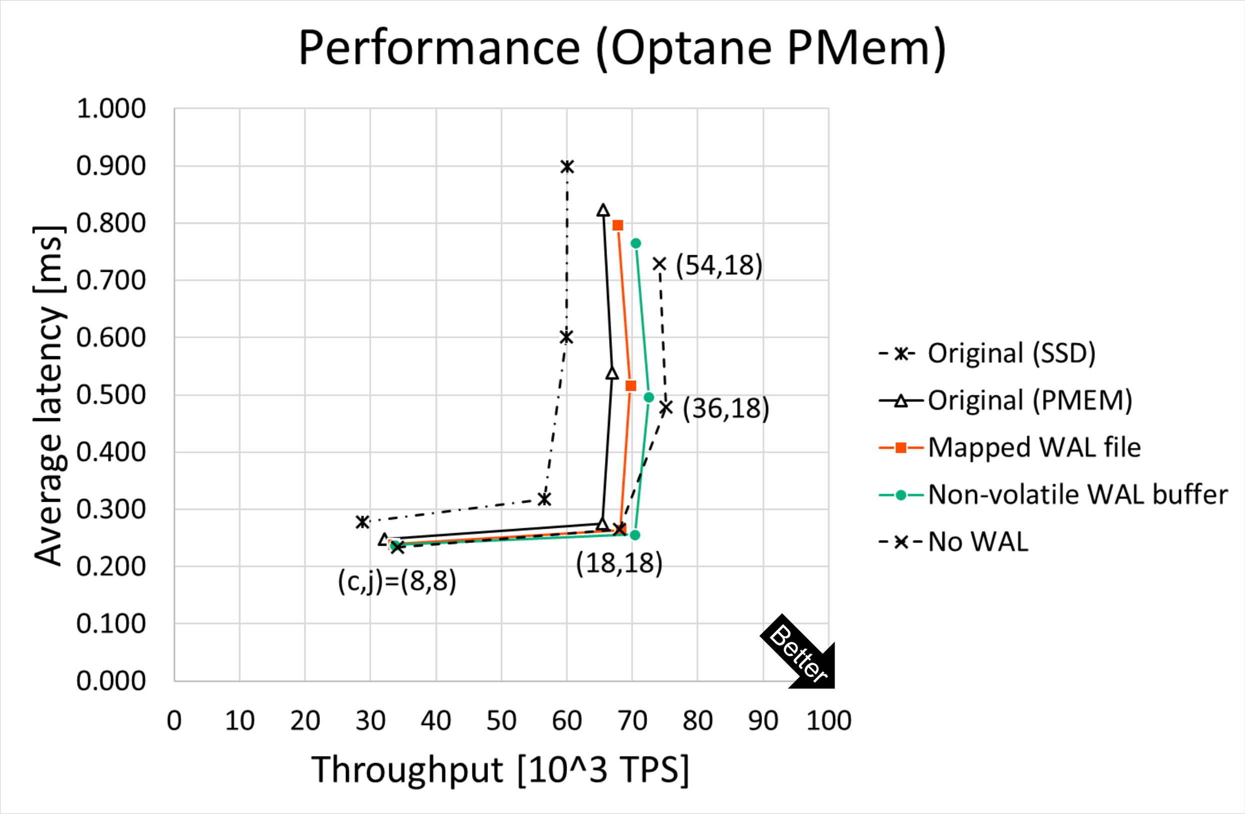
Hardware, software, and configuration

Hardware	
System	HPE ProLiant DL380 Gen10
CPU	Intel Xeon Gold 6240M x2 sockets (18 cores per socket; HT disabled by BIOS)
DRAM	DDR4 2933MHz 192GiB/socket x2 sockets (32GiB per channel x 6 channels per socket)
Optane PMem	DDR4 2666MHz 1.5TiB/socket x2 sockets (Apache Pass; 256GiB per channel x 6 channels per socket; interleaving enabled; AppDirect Mode)
PCIe SSD	DC P4800X Series SSDPED1K750GA
Software	
Distro	Ubuntu 20.04.1
C compiler	gcc 9.3.0
libc	glibc 2.31
Linux kernel	5.7 (vanilla kernel)
Filesystem	ext4 (DAX enabled when using Optane PMem)
PMDK	1.9
PostgreSQL	14devel (200f610 on Jul 26, 2020)
VTune Profiler	2020 Update 2 (build 610396)

postgresql.conf	Value
max_connections	300
shared_buffers	32GB
checkpoint_timeout	12min
checkpoint_completion_target	0.7
{max,min}_wal_size	80GB
random_page_cost	1.0
effective_cache_size	96GB
autovacuum_max_workers	4
autovacuum_freeze_max_age	2000000000 (2×10^9)
autovauum_vacuum_cost_limit	400
pgbench	
Scale factor (-i -s)	50
Database connection	Unix domain socket
Query mode (-M)	prepared
Transaction script (-b)	tpcb-like (default)

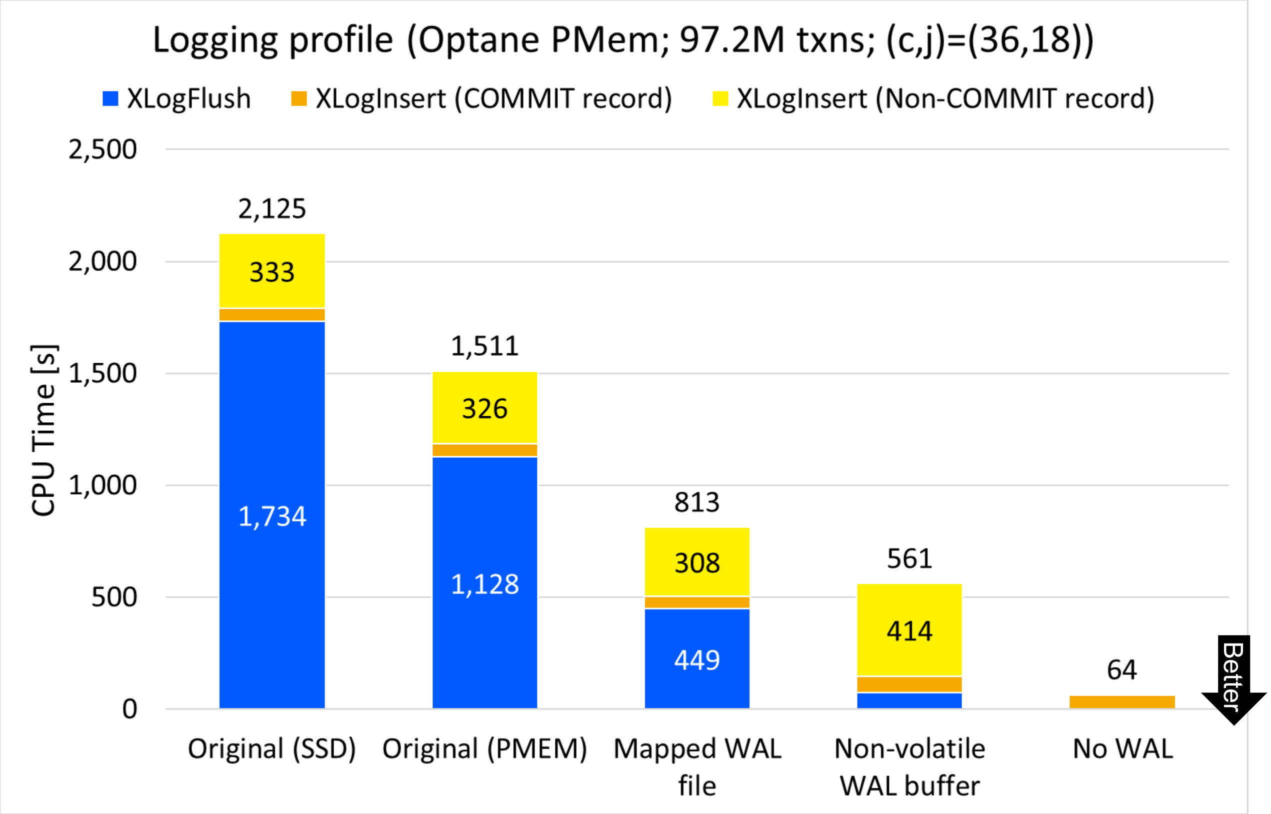
Performance and profile results

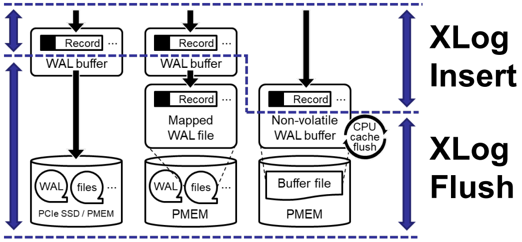
Performance (Optane PMem)



- "Non-volatile WAL buffer" achieves better throughput and latency than the other durable setups
  - Close to "No WAL" or even better than that when (c, j)=(18,18)

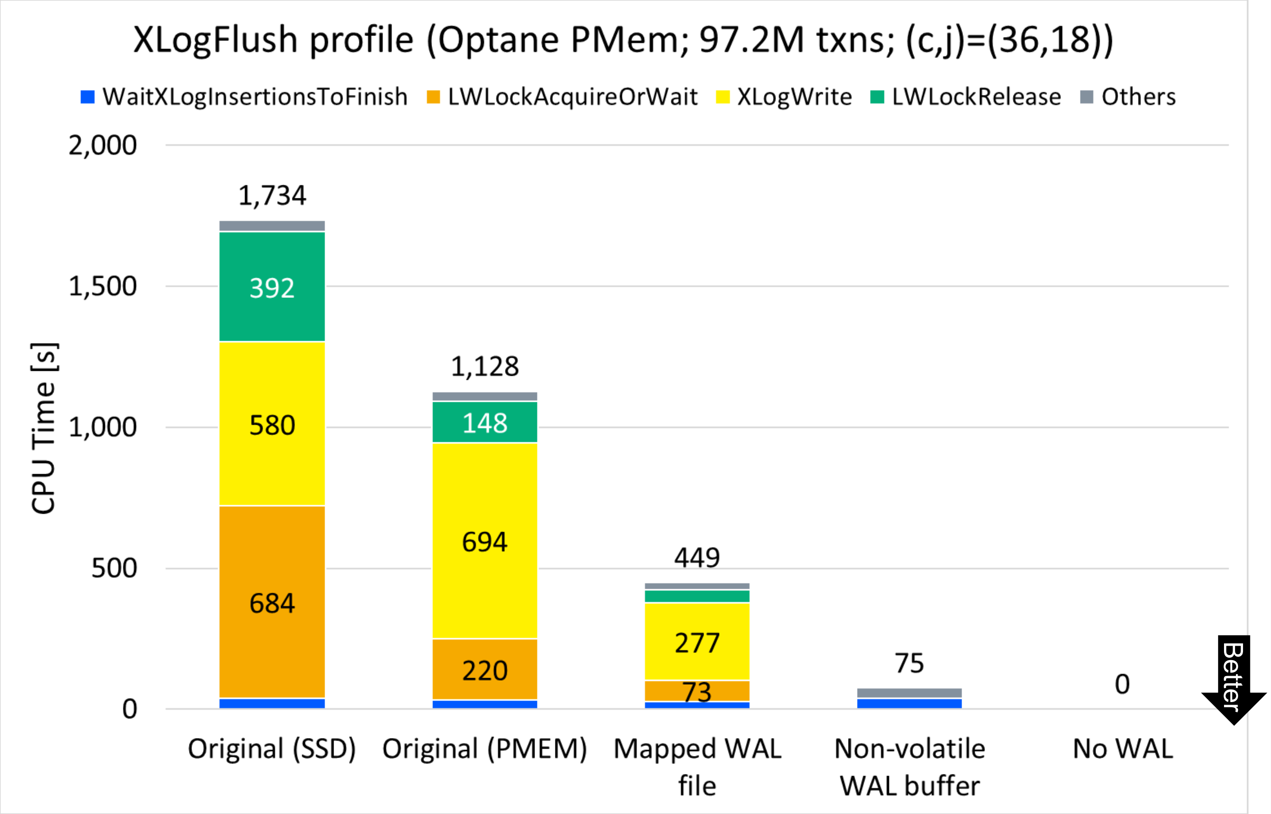
Logging profile (Optane Pmem)

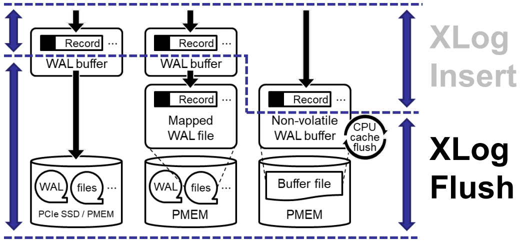




- XLogFlush time decreases (details in the next slide)
- In regard to “Non-volatile WAL buffer,” XLogInsert time increases while total logging time decreases
  - Probably because Optane PMem is a little slower than DRAM

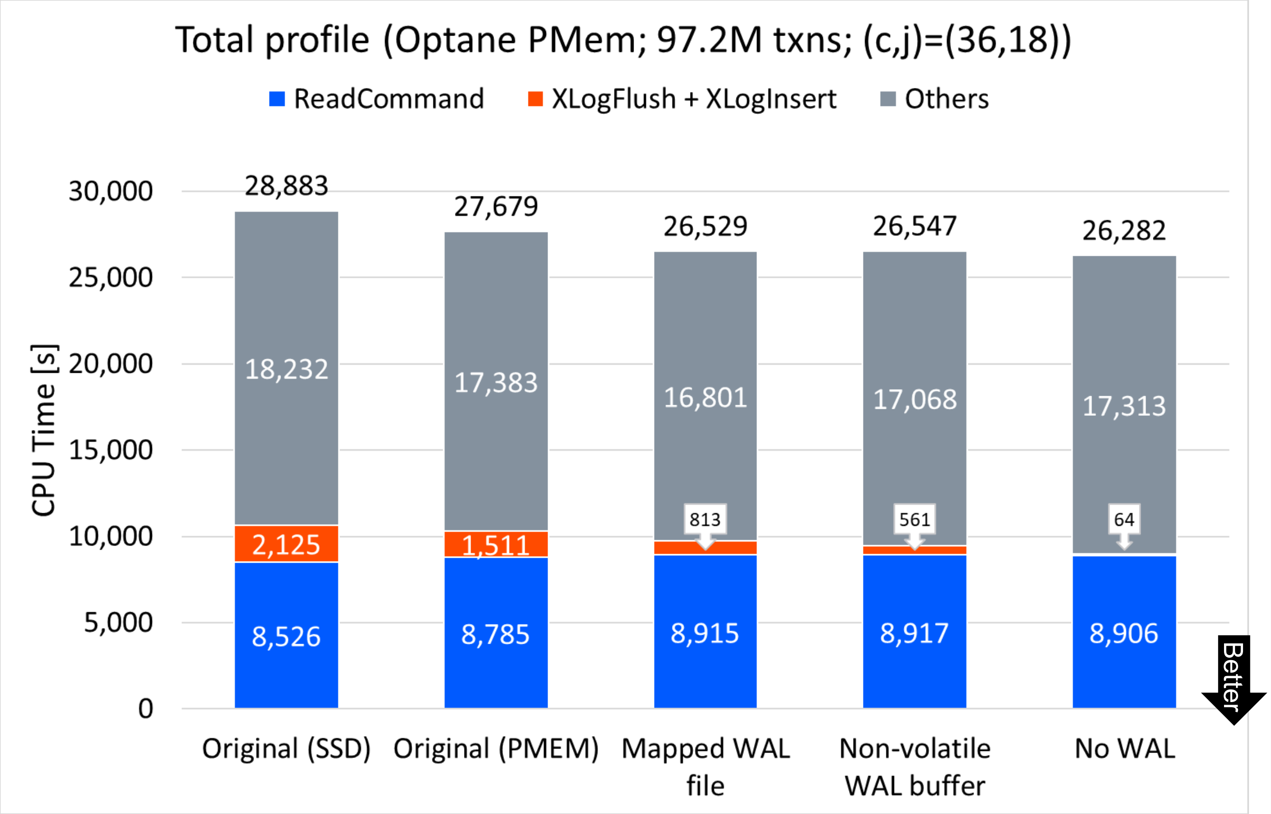
XLogFlush profile (Optane Pmem)





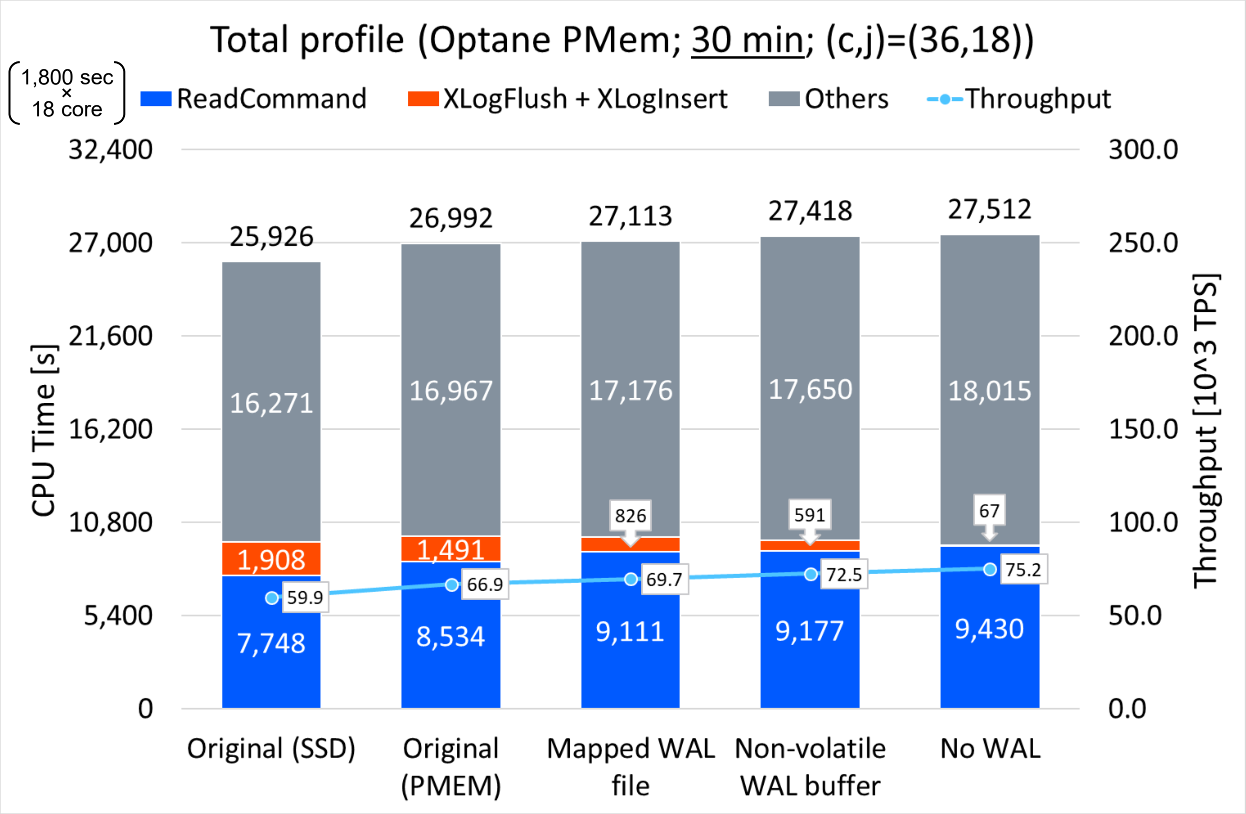
- "Non-volatile WAL buffer" eliminates write and **lock** time in XLogFlush
  - Needs no write but CPU cache flush to make log records durable
  - Needs no WAL lock **coming with write**

Total profile (Optane Pmem)



- Logging (XLogFlush + XLogInsert) time decreases
- ReadCommand (reading transactions sent by clients) time is almost equal in each setup

30-minues profile (Optane Pmem)



- More time is spent for ReadCommand and other functions, that is, transaction processing
  - Throughput increases as ReadCommand time increases

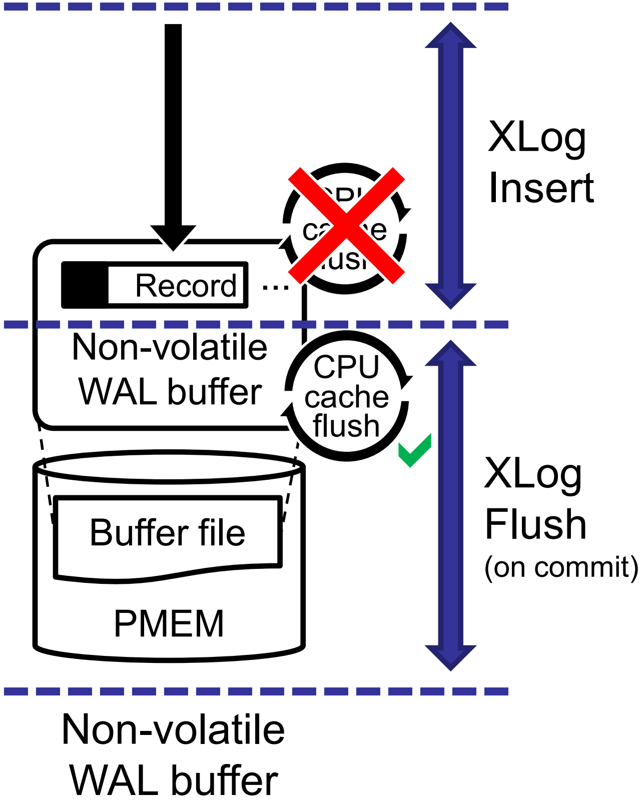
Evaluation of our re-design

PostgreSQL can use PMEM better than before by single-level logging architecture!

- Improves both throughput and latency
- Eliminates lock time in addition to write time
- Spends more time for reading and processing transactions

How could we go ahead?

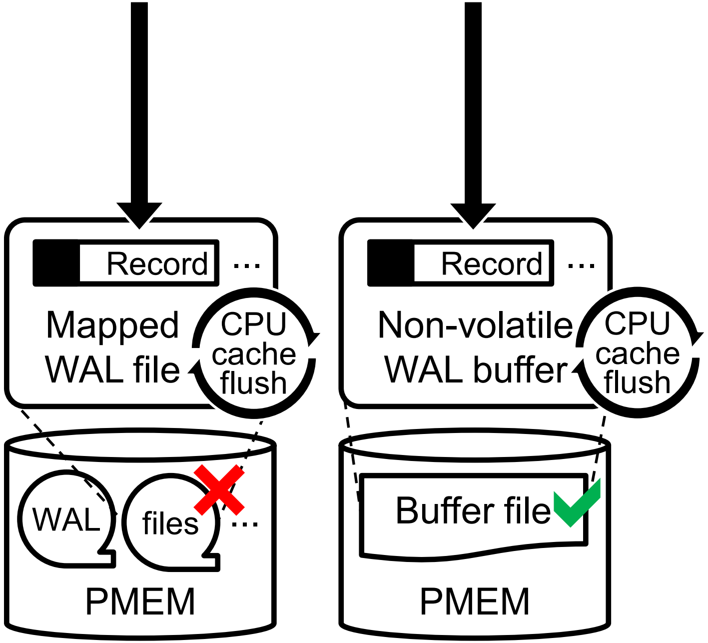
Less flush, better performance



Flushing CPU cache <u>on insertion</u> was turned out to be a **bad** idea:

- Simpler to implement, but required **more flush** for each transaction
  - One transaction may insert multiple log records while commits at most once
- Got bad performance, **even worse than "Original (PMEM)"** when using Optane PMem

Less overhead, better performance



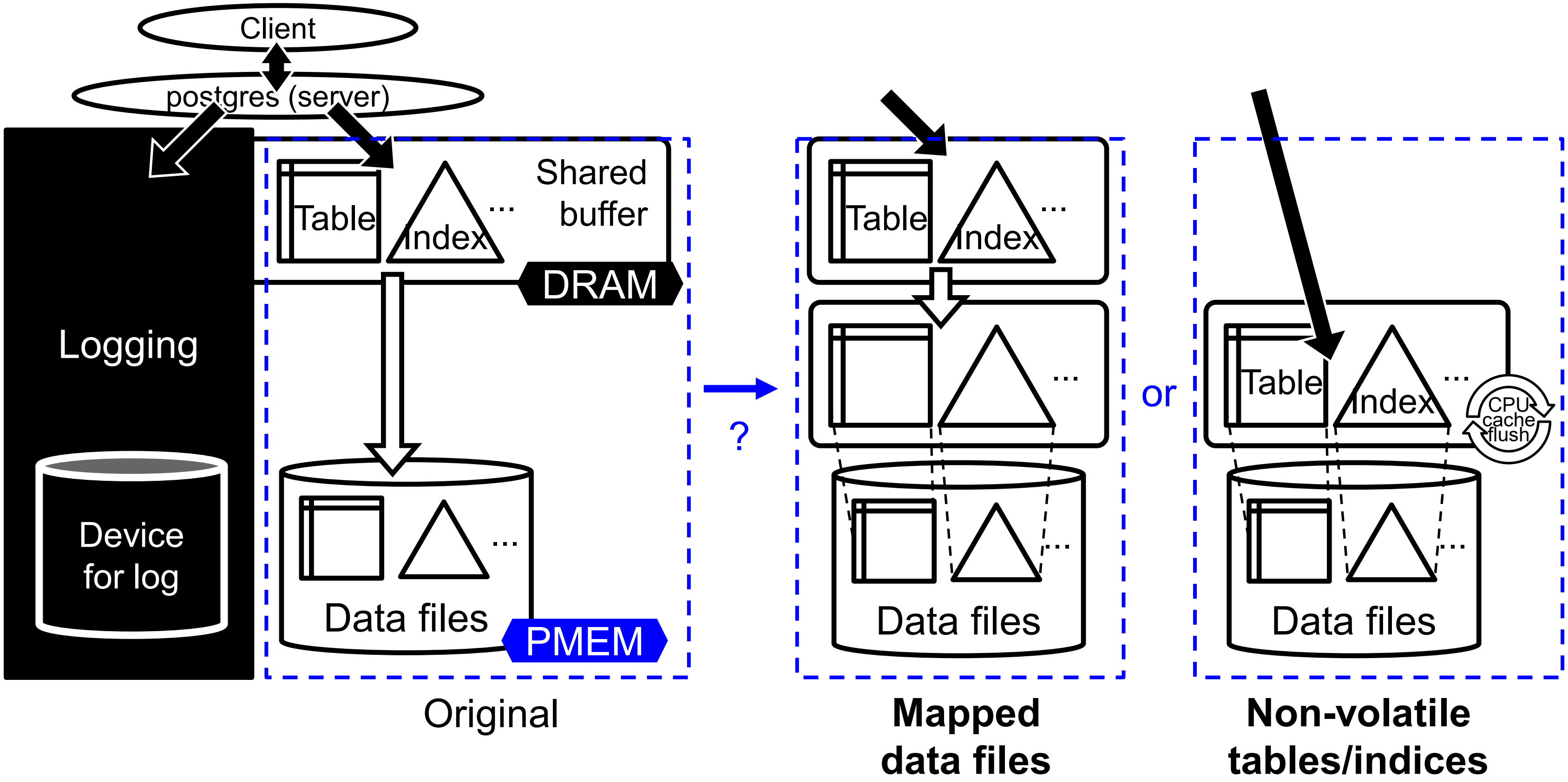
WAL file as WAL buffer

Non-volatile WAL buffer

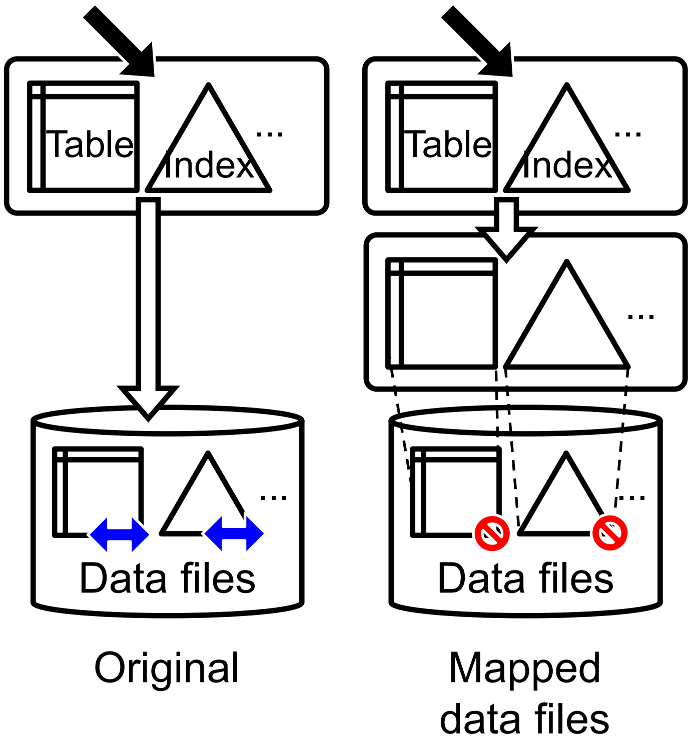
Mapping the existing WAL file as WAL buffer turned out to be a **bad** idea:

- Simple to implement, but required to **switch to the next file** for every 16 MB (default size of WAL file)
- Got **worse** performance than “Non-volatile WAL buffer” due to …
  - Unmapping old file and mapping new one
  - Page fault for the newly-mapped file

How about tables and indices?



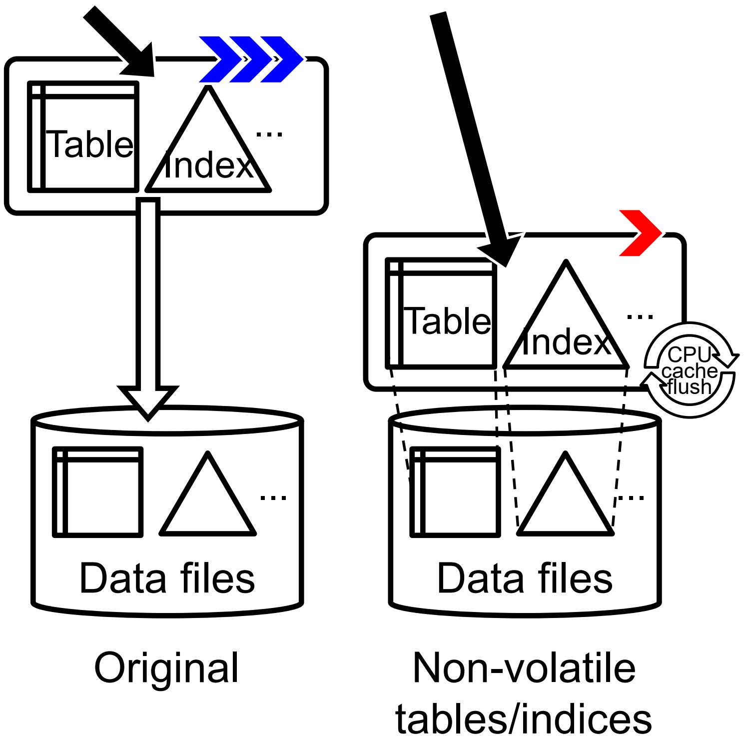
Mapped data files for checkpointing



"**Impedance mismatch**" between file API and memory-mapped file:

- Data files are **extensible** while mapping requires **fixed** file size
  - Cf. each WAL file has fixed size
- No silver bullet
  - Pre-allocating large files would waste PMEM space
    - Critical when using small NVDIMM-N
  - Remapping on extension would degrade performance

Non-volatile tables/indices



PMEM's **speed** and capacity:

- NVDIMM-N is fast but small to store large tables and indices
- Optane PMem is large but a little slower than DRAM in both **load** and store, so it may even degrade **read-only** transaction performance

PMEM-aware data structure

Disk-based to **memory-based:**

- Granularity: Page to byte or cache-line
- Arrangement: Sequential to random
- Referencing: Offset to pointer

Proposal to PostgreSQL community

Non-volatile WAL buffer:

- https://www.postgresql.org/message-id/flat/002f01d5d28d%2423c01430%246b403c90%24%40hco.ntt.co.jp_1
  - Or search the Web with "Non-volatile WAL buffer," find the page on www.postgresql.org, and view "Whole Thread"
- The latest patchset is v3 (or might be v4). Feedbacks are welcome :)

Resources

- "Mapped WAL files" patchset
  - https://www.postgresql.org/message-id/flat/CAOwnP3ONd9uXPXKoc5AAfnpCnCyOna1ru6sU%3DeY_4WfMjaKG9A%40mail.gmail.com
- Yoshimi's presentation in SNIA SDC 2018
  - https://www.snia.org/sites/default/files/SDC/2018/presentations/PM/Ichiyanagi_Yoshimi_Challenges_for_Implementing_PMEM_Aware_Application_with_PMDK.pdf