Parallel Recovery in PostgreSQL

Koichi Suzuki

May 31st, 2023

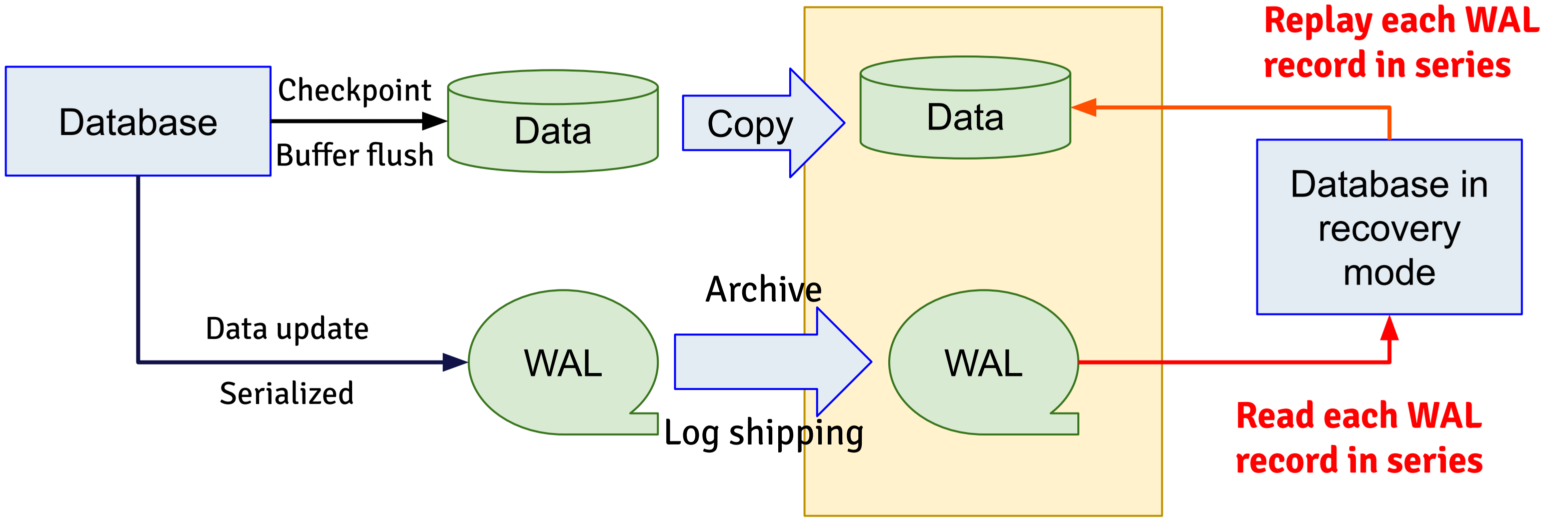
EDB

Agenda

- Recovery Overview
- Possibility of recovery parallelism
- Additional rules and synchronization
- Implementation architecture
- Current achievement
- Remaining works

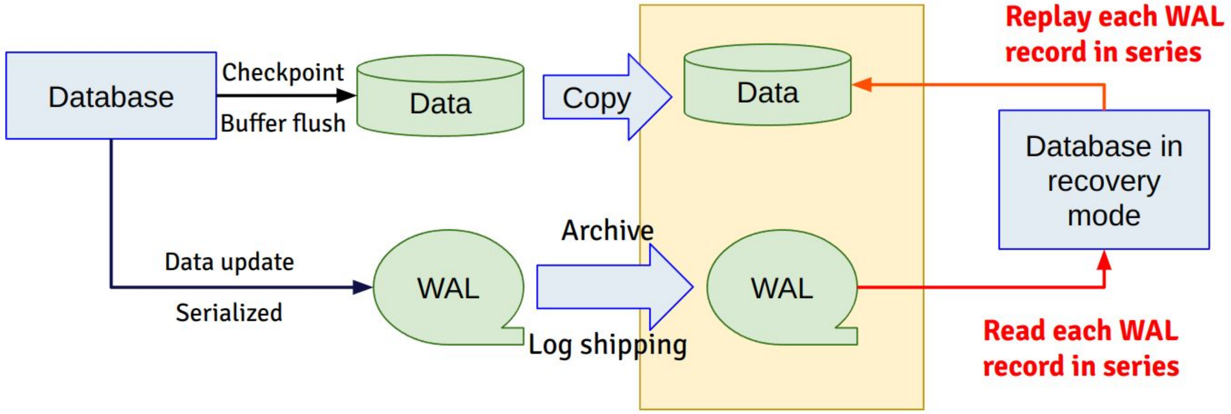
Recovery Overview

- WAL (Write-Ahead-Log) takes essential role in backup and recovery.
- This is also used in log-shipping replication.
- Recovery is done by dedicated startup process.



Single replay thread, why?

- Single thread replay guarantees consistent replay
  - This is the order of write to the database
  - Replaying in the write order is simple and the safest.



Current WAL replay architecture

WAL record type:

- Categorized by “**resource manager**”
  - For types of data object
    - Heap, transaction, Btree, GIN, ….
  - Each resource manager provides replay function for each record.
- Replay (“**redo**” in the source) reads the resource manager ID (type `RmgrId`) of each WAL record and calls replay function for these resources.
  - `StartupXLOG()` in `xlog.c`
- Replay is done by dedicated **startup process**.
- Many more to handle in `StartupXLOG()`
  - Determine if postmaster can begin to accept connection,
  - Feedback commit/abort back to the primary,
  - Recovery target handling,

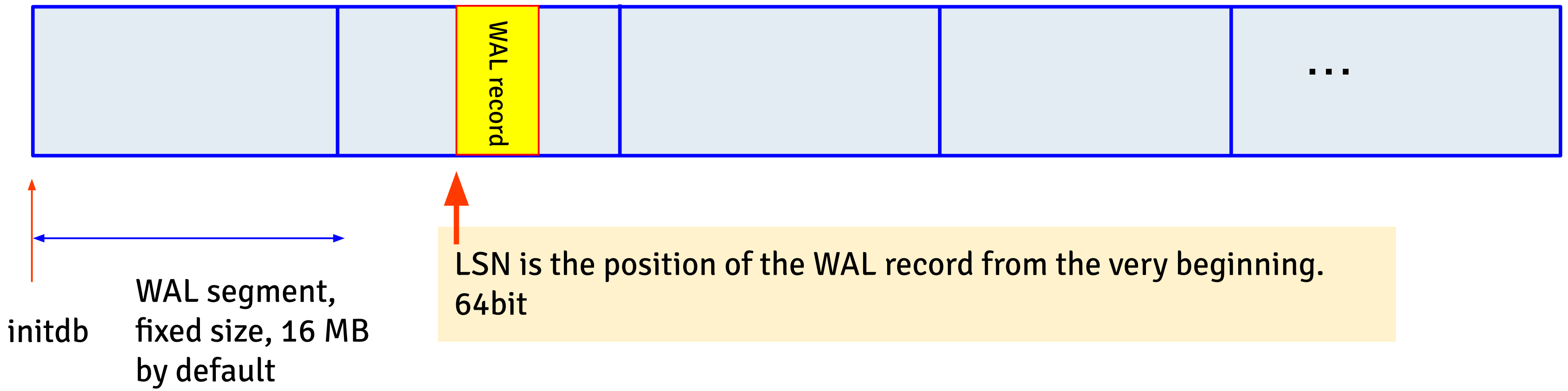
  ….(many more)...

Current list of RmgrId (as of V14/15)

```
XLOG
Transaction
Storage
CLOG
Database
Tablespace
MultiXact
RelMap
Standby
Heap2
Heap
Btree
Hash
Gin
Gist
Sequence
SPGist
BRIN
CommitTs
ReplicationOrigin
Generic
LogicalMessage
```

When new resource manager (for example, access method and index) are added, new RmgrID and its replay function are added. No changes are needed in current redo framework.

Structure of WAL (simplified)



LSN is the position of the WAL record from the very beginning.
64bit

- Each WAL segment has its own header,
- WAL record may span across multiple WAL segment files.

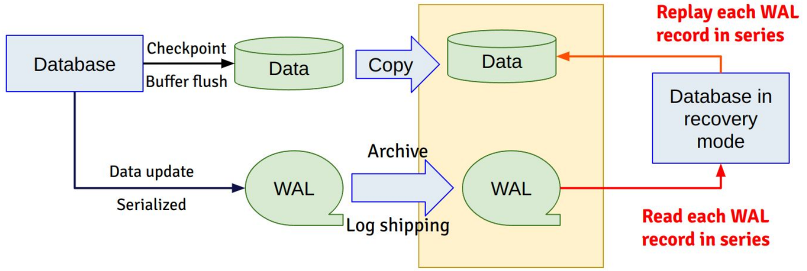
Motivation of parallelism in the recovery?

Motivation

- Save time of restoration from the backup.
- Improve replication lag in log-shipping replication.
- At least we can replay WAL record for different database page in parallel.

Challenge (for example)

- There are many constraints which we must follow during WAL replay
  - When replaying COMMIT/ABORT, all the WAL records for the transaction must have been replayed.
  - Some WAL records has replay information for multiple pages.



Basic idea of constraints in parallel replay

1. For each database page, replay must be done in LSN order.
2. Replaying WAL for different pages can be done in different worker process in parallel.
3. If WAL is not associated with database page, they are replayed in LSN order by dedicated thread (transaction worker).
4. For some WAL record, such as timeline change, all the preceding WAL records must have been replayed before replaying such WAL record.
5. If a WAL record is associated with multiple pages, they will be replayed by one of the workers.
    - To maintain the first constraint, we need some synchronization (mentioned later)
6. Before replaying commit/abort/prepare, it must be confirmed that all the WAL record for the transaction must have been replayed. (mentioned later).

Worker processes for parallel recovery

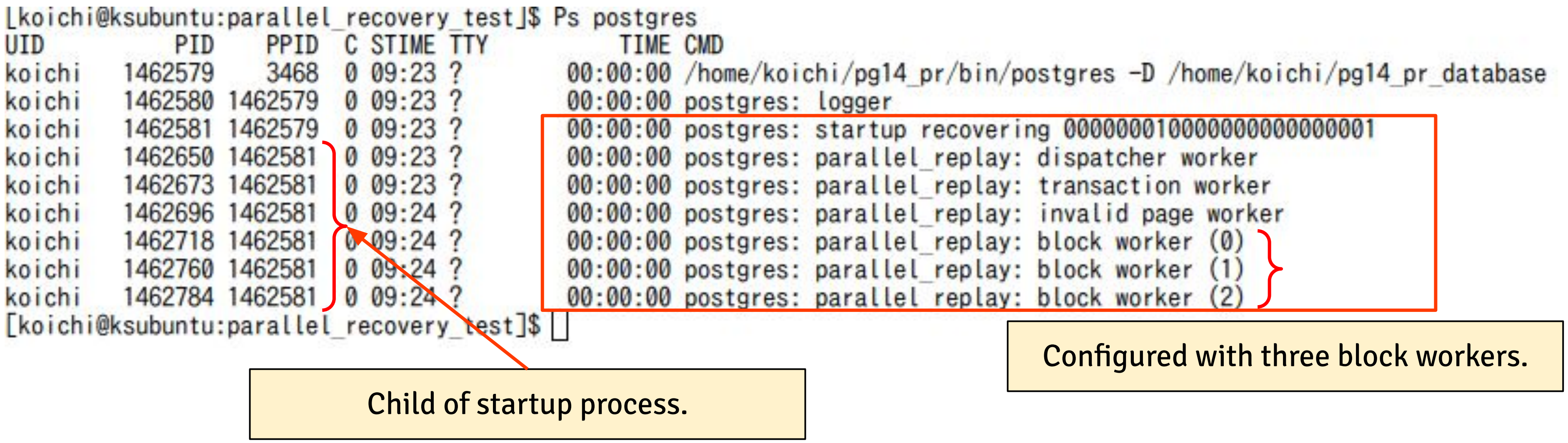
All the worker processes are startup process or its child process.

- **Reader worker** (dedicated)
  - This is startup process itself. Read each WAL record and ques to the distributor worker
- **Distributor worker** (dedicated)
  - Parses each WAL record and determine which worker process to handle.
  - If WAL record is associated with block data, then it is assigned to one of the block worker based on the hash value of the block,
  - Add each WAL record to the queue of each worker process in LSN order.
  - Can be merged into reader worker.
- **Transaction worker** (dedicated)
  - Replays all the WAL records without block information.

Worker processes for parallel recovery (cont.)

- **Block worker** (multiple, configurable)
  - Replays assigned WAL record.
  - Handling of multi-page WAL will be mentioned later.
- **Invalid page worker** (dedicated)
  - Target database page may be missing or inconsistent because table might have been dropped by subsequent WALs.
  - Current recovery registers this as invalid page and make correction when replaying corresponding DELETE/DROP WAL.
  - Now dedicated process to reuse existing code using hash functions based on the memory context belonging to a process,
  - May receive queue from txn and block workers.
  - Can be modified to use shared memory. In this case, this worker is not needed.

Worker processes



```
[koichi@ksubuntu:parallel_recovery_test]$ Ps postgres
UID          PID     PPID  C STIME TTY          TIME CMD
koichi   1462579     3468  0 09:23 ?        00:00:00 /home/koichi/pg14_pr/bin/postgres -D /home/koichi/pg14_pr_database
koichi   1462580  1462579  0 09:23 ?        00:00:00 postgres: logger
koichi   1462581  1462579  0 09:23 ?        00:00:00 postgres: startup recovering 000000010000000000000001
koichi   1462650  1462581  0 09:23 ?        00:00:00 postgres: parallel_replay: dispatcher worker
koichi   1462673  1462581  0 09:23 ?        00:00:00 postgres: parallel_replay: transaction worker
koichi   1462696  1462581  0 09:24 ?        00:00:00 postgres: parallel_replay: invalid page worker
koichi   1462718  1462581  0 09:24 ?        00:00:00 postgres: parallel_replay: block worker (0)
koichi   1462760  1462581  0 09:24 ?        00:00:00 postgres: parallel_replay: block worker (1)
koichi   1462784  1462581  0 09:24 ?        00:00:00 postgres: parallel_replay: block worker (2)
[koichi@ksubuntu:parallel_recovery_test]$ 
```

Configured with three block workers.

Child of startup process.

Shared data among workers

- WAL record and its parsed data is shared among workers through dynamic shared memory (dsm).
  - Allocated by reader worker (startup) before other workers are folked.
  - Handling of multi-page WAL will be mentioned later.
- Other shared data structures are accommodated in the same dsm.

Shared data among workers (cont.)

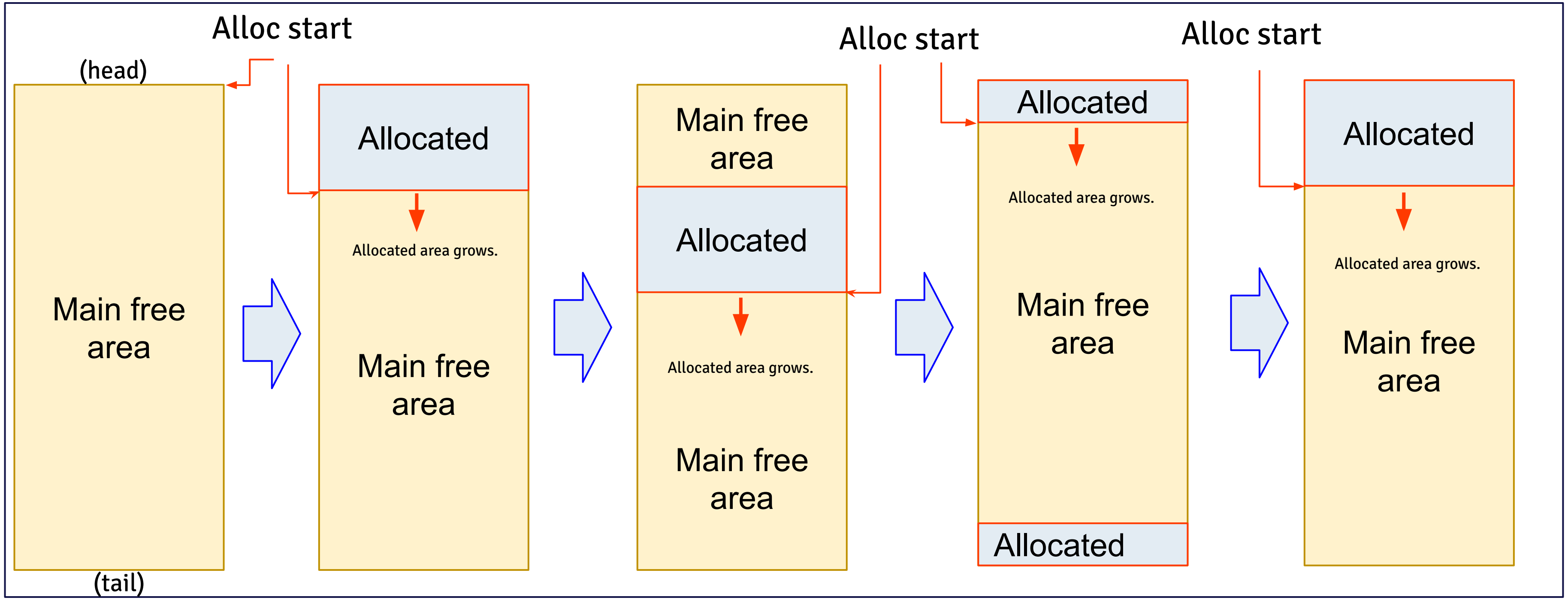
- Buffer for WAL and its parsed result,
- Status of outstanding transaction,
    - Latest LSN assigned to each block worker,
- Status of each worker,
- Queue to assign WAL to workers,
- History of outstanding WAL waiting for replay,
    - Used to advance replayed LSN.

- Data protection using spinlock.
    - No I/O or wait event should be associated while a spinlock is acquired.
    - Only one spinlock should be acquired by a worker.
        - May have to allow exception though…

WAL buffer is allocated in cyclic manner

- WAL buffer is allocated in cyclic manner to avoid fragmentation.
- WAL is eventually replayed and its data is freed.
- If we allocate WAL buffer in cyclic manner, we don’t have to worry about memory fragmentation.
- Buffer is allocated at the start of main free area.
- When a buffer is freed, it is concatenated with surrounding free area and if necessary, main free area info is updated.
- If free area is among allocated buffers, we don’t take care of it until it is concatenated to main free area.

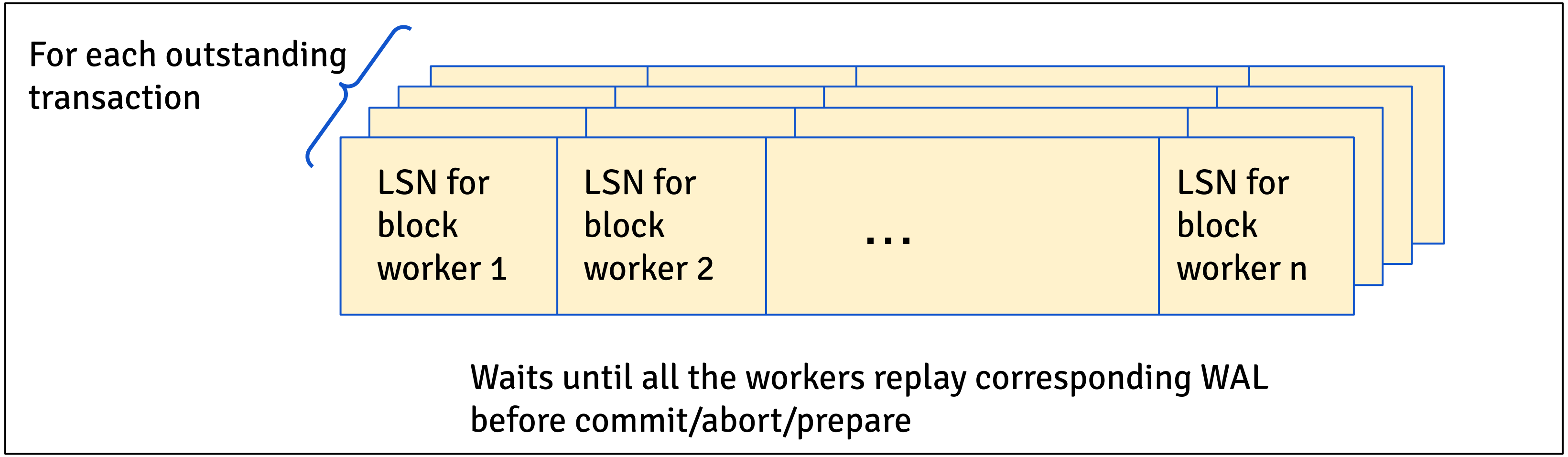
WAL buffer is allocated in cyclic manner (cont.)



- If sufficient main free area is not available, wait until other workers replay WALs and free buffers.
- If all workers is waiting for the buffer, reply aborts and suggests to enlarge shared buffer (configurable through GUC).
- With three block workers and ten queues for each worker, 4MB of buffer looks sufficient. We may need more with more block workers and more queue depth.

Transaction status

- Composed of set of latest assigned WAL LSNs to each block worker for each transaction.
- When the worker replays this LSN, it is cleaned.
- When transaction worker replays commit/abort/prepare, it checks all the WALs for this transaction has been replayed.
- If not, waits until such workers replay all the assigned WALs (mentioned later).


For each outstanding transaction
LSN for block worker 1
LSN for block worker 2
...
LSN for block worker n
Waits until all the workers replay corresponding WAL before commit/abort/prepare

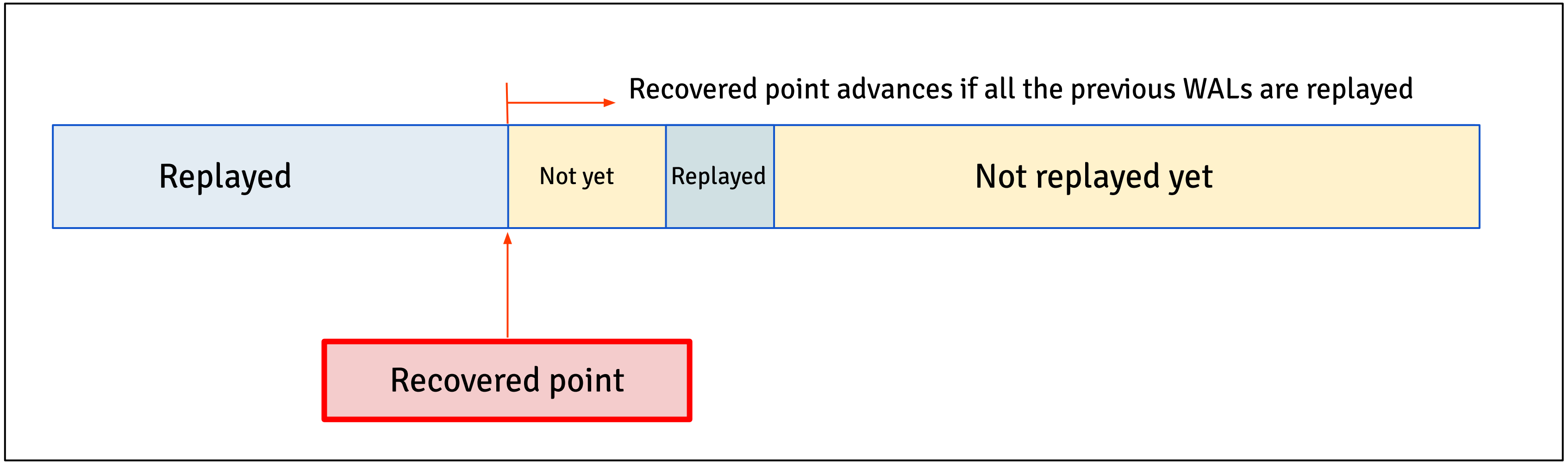
Worker status

- Latest assigned WAL LSN,
- Latest replayed WAL LSN,
- Flags to indicate
  - the worker is waiting for sync from other workers,
  - If the worker failed,
  - If the worker terminated.

WAL history

- Recovered point can be advanced only all the WALs are replayed.
- If any WAL has not been replayed, we need to wait to advance recovered point until such WAL is replayed.

Wal history tracks this status.


Recovered point advances if all the previous WALs are replayed
Replayed
Not yet
Replayed
Not replayed yet
Recovered point

Synchronization among workers

- Dedicated data is queued asking for sync message.
- Sync message is sent via socket (unix domain udp). Each worker has its own socket to receive synch.
- For multi-page WAL, another mechanism is implemented for performance.

Wal history tracks this status.

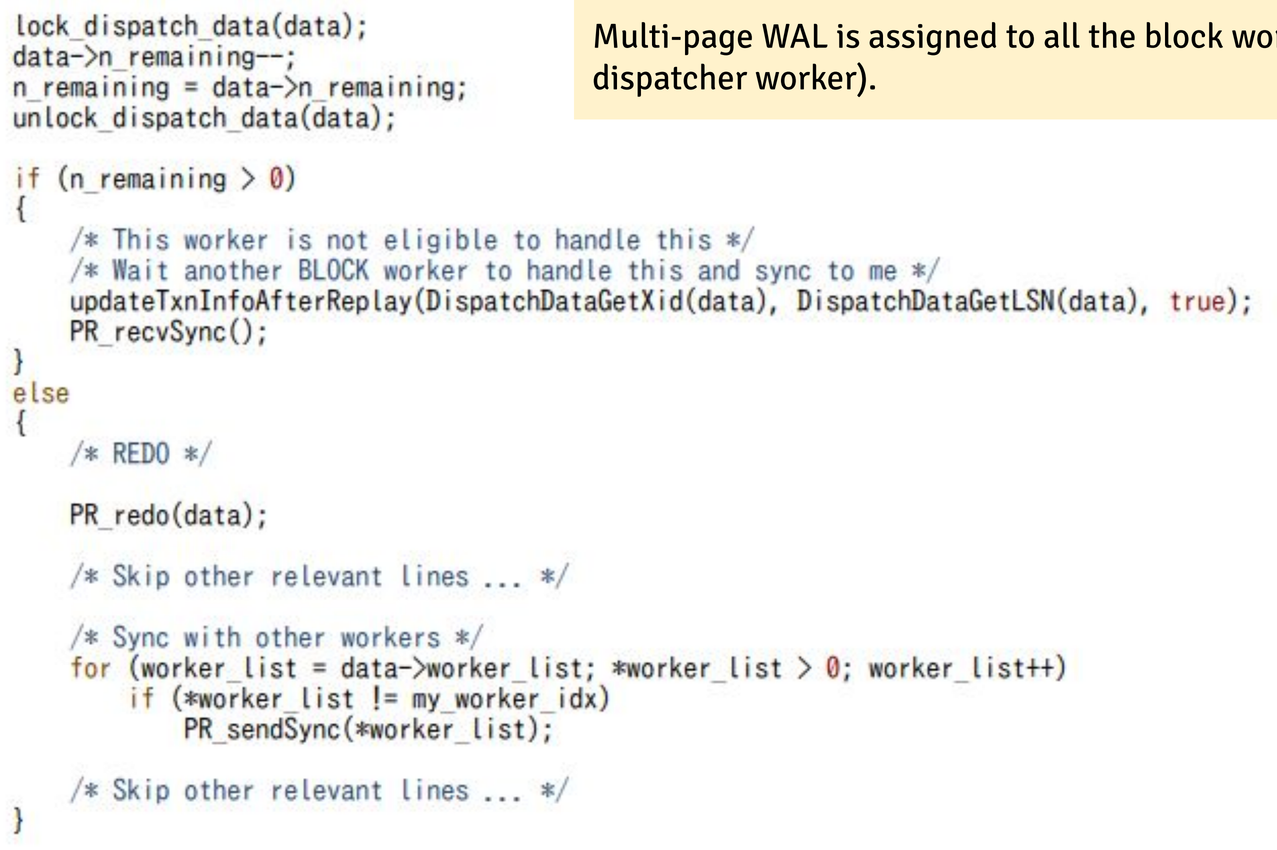
Waiting for all the preceding WAL replay

- Distributor worker sends queue asking for synchronization.
- Distributor reads data from the target worker from the socket.
- When the target worker receives synchronization request, writes synchronization data to the socket.

Synchronization for multi-page WAL

- Multi-page WAL is assigned to all the block workers based on the hash value of the page.
- WAL is associated with array of assigned workers and number of remaining workers (initial value is number of assigned block worker).
- When a block worker receives WAL, it checks if remaining worker is one (only myself).
  - If so, then the worker replays the WAL and then write synch to all the other workers.
  - If not, then the worker waits sync from the above worker.
- This guarantees that WAL replay to each page is in the order of WAL LSN.

Synchronization for multi-page WAL (cont.)



```
lock_dispatch_data(data);
data->n_remaining--;
n_remaining = data->n_remaining;
unlock_dispatch_data(data);

if (n_remaining > 0)
{
    /* This worker is not eligible to handle this */
    /* Wait another BLOCK worker to handle this and sync to me */
    updateTxnInfoAfterReplay(DispatchDataGetXid(data), DispatchDataGetLSN(data), true);
    PR_recvSync();
}
else
{
    /* REDO */

    PR_redo(data);

    /* Skip other relevant lines ... */

    /* Sync with other workers */
    for (worker_list = data->worker_list; *worker_list > 0; worker_list++)
        if (*worker_list != my_worker_idx)
            PR_sendSync(*worker_list);

    /* Skip other relevant lines ... */
}
```

Multi-page WAL is assigned to all the block workers for each page (by dispatcher worker).

Wait some other block worker replays

Last block worker to pick up WAL. Replay and sync others.

Current status of the work and achievement

Current Status

- Code is almost done.
- Still with many debug codes.
- Runs under gdb control.
- Parallelism looks working without issue.
- **Existing redo functions are running without modification**.

Affected major source files

Modified/Added	Source	Outline
A	`src/backend/access/transam/parallel_replay.c` `src/include/access/parallel_replay.h`	Main parallel recovery logic. Each worker loop.

M	`src/backend/access/transam/xlog.c`
	`src/backend/access/transam/xlogreader.c`
	`src/backend/access/transam/xlogutils.c`
	`src/include/access/xlog.h`
	`src/include/access/xlogreader.h`
	`src/include/access/xlogutils.h`
		Read and analyze WAL into shared buffer
		Pass WAL to distributor worker.

M	`src/backend/postmaster/postmaster.c`
	`src/backend/postmaster/startup.c`
	`src/backend/storage/lmgr/proc.c`
	`src/backend/utils/activity/backend_status.c`
	`src/backend/utils/adt/pgstatfuncs.c`
	`src/include/miscadmin.h`
	`src/include/postmaster/postmaster.h`
	`src/include/storage/proc.h`
	`src/include/utils/backend_status.h`
		Fork worker processes.

Affected major source files (cont.)

Modified/Added	Source	Outline
M	`src/backend/utils/misc/guc.c`	Additional GUC parameters

M	`src/backend/utils/error/elog.c`
	`src/backend/utils/activity/backend_status.c`
	`src/backend/utils/misc/ps_status.c`
		Add workers to log and status.

Additional major GUC parameters

Name	Type	Default	Description
parallel_replay	bool	off	Enable parallel recovery,
num_preplay_workers	int	5	Number of parallel replay worker, including reader/distributor/txn and invalid pag.
num_preplay_queues	int	128	Number of queue element used to pass WAL to various workers.
num_preplay_max_txn	int	max_connec tions	Number of transactions running in parallel in the recovery.
preplay_buffers	int	(calculated)	Shared buffer size, allocated using dynamic shared memory.

Remaining work

Remaining work

- Cleasnup test code.
- Cleanup structure, which still have some unused data from past try-and-error attempt.
- Validation of consistent recovery point determination.
- Determination of the end of WAL read.
- Port to version 16 or later.
- Run without gdb and measure the performance gain.
  - Archive recovery,
  - Log-shipping replications.

Issues for further discussion/development

Issues for discussion

- Synchronization:
  - Now using unix domain socket directly. May need some wrapper for portability.
- Queueing
  - Now using POSIX mqueue directly. It is very fast and simple and is supporting multi to one queueing.
  - Is it portable to other environment such as Windows?
- Bringing to PG core
  - Large amount of code: around 7k lines with debug code.
  - Maybe around 4k lines without debug code.
  - Need expert's help to divide this into manageable smaller pieces for commit fest.

Resources

Source repo

https://github.com/koichi-szk/postgres.git
Branch: parallel_replay_14_6

Test tools/environment

https://github.com/koichi-szk/parallel_recovery_test.git

Branch: main

PostgreSQL Wiki:

https://wiki.postgresql.org/wiki/Parallel_Recovery

They are now all public. Please let me know if you are interested in.

EDB

Thank you very much

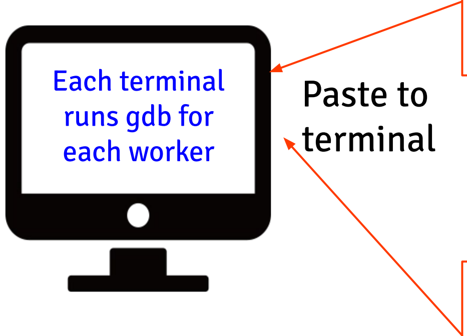
Koichi Suzuki

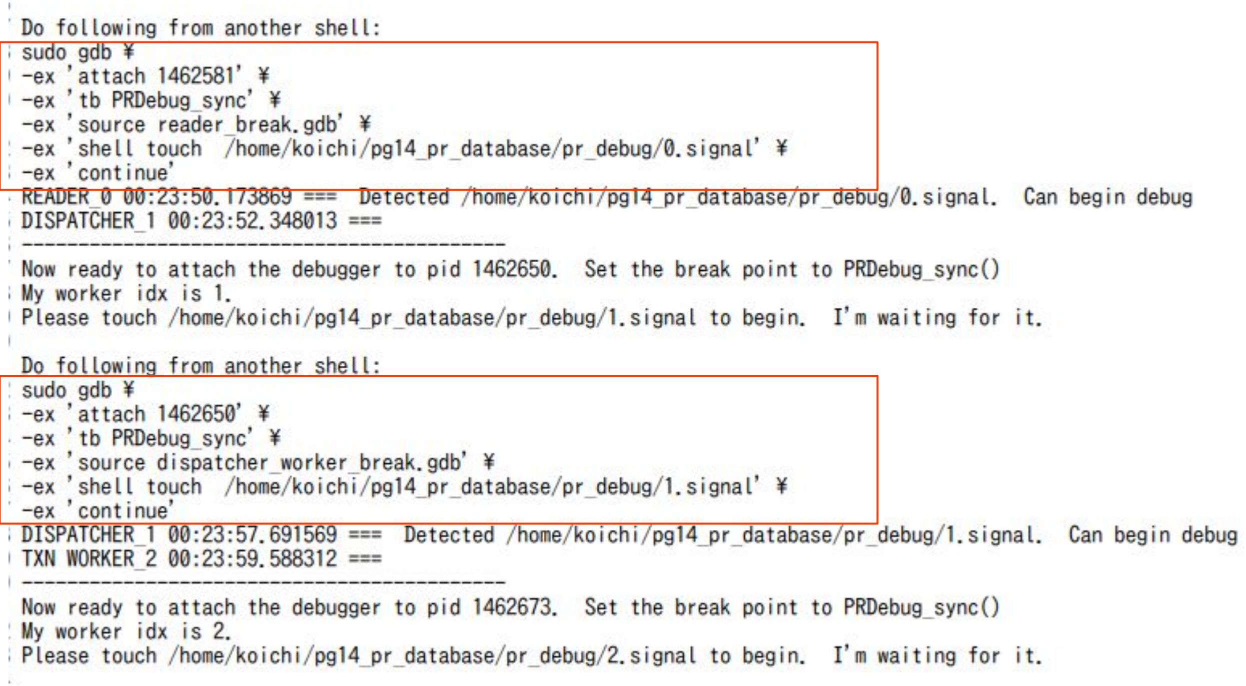
koichi.suzuki@enterprisedb.com

koichi.dbms@gmail.com

Challenge in the test

- Startup process starts before postmaster begins to accept connection.
- No direct means to pause and notice its PID to attach to gdb.
- Writes these information to a debug file, wait until it is attached to gdb and signal file is touched.





```
Do following from another shell:
sudo gdb ¥
-ex 'attach 1462581' ¥
-ex 'tb PRDebug_sync' ¥
-ex 'source reader_break.gdb' ¥
-ex 'shell touch  /home/koichi/pg14_pr_database/pr_debug/0.signal' ¥
-ex 'continue'
READER_0 00:23:50.173869 ===  Detected /home/koichi/pg14_pr_database/pr_debug/0.signal.  Can begin debug
DISPATCHER_1 00:23:52.348013 ===
------------------------------------------
Now ready to attach the debugger to pid 1462650.  Set the break point to PRDebug_sync()
My worker idx is 1.
Please touch /home/koichi/pg14_pr_database/pr_debug/1.signal to begin.  I'm waiting for it.

Do following from another shell:
sudo gdb ¥
-ex 'attach 1462650' ¥
-ex 'tb PRDebug_sync' ¥
-ex 'source dispatcher_worker_break.gdb' ¥
-ex 'shell touch  /home/koichi/pg14_pr_database/pr_debug/1.signal' ¥
-ex 'continue'
DISPATCHER_1 00:23:57.691569 ===  Detected /home/koichi/pg14_pr_database/pr_debug/1.signal.  Can begin debug
TXN_WORKER_2 00:23:59.588312 ===
------------------------------------------
Now ready to attach the debugger to pid 1462673.  Set the break point to PRDebug_sync()
My worker idx is 2.
Please touch /home/koichi/pg14_pr_database/pr_debug/2.signal to begin.  I'm waiting for it.
```