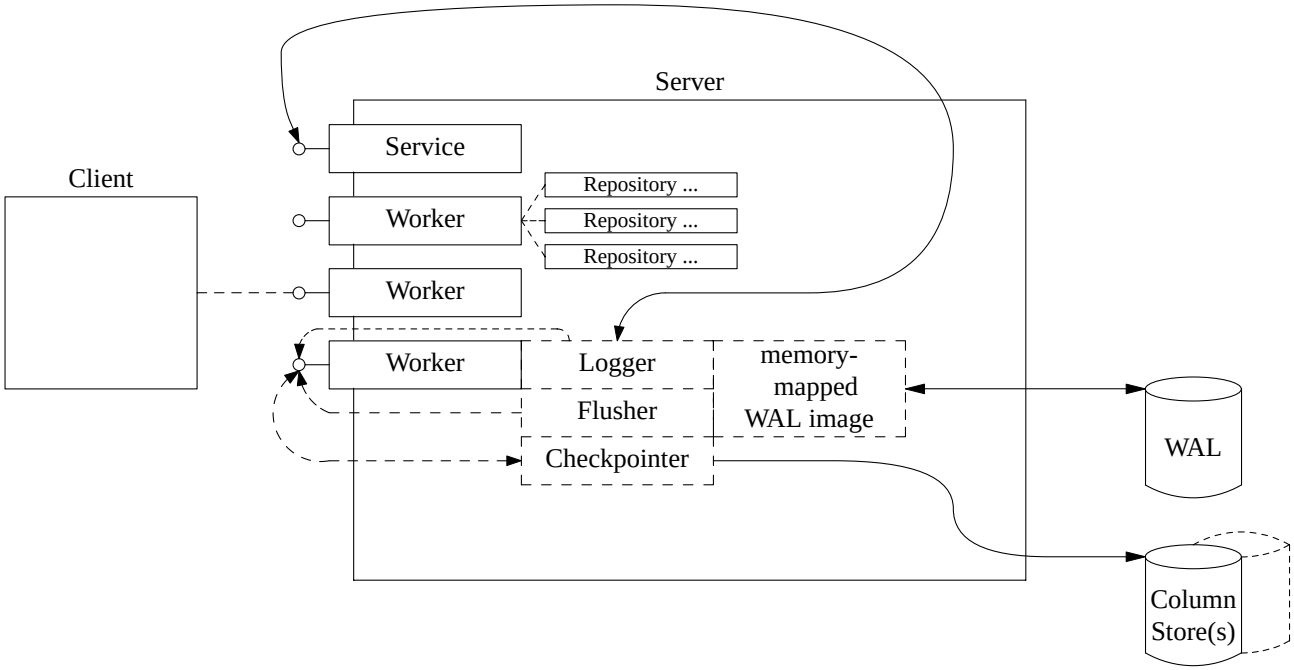
# Notes on Server Design

The architecture has a physical and logical vocabulary.

| Logical | Physical |
|---|---|
| Topology | Hive |
| Host | Service |
| Pod | Worker |
| Column | Repository |

A **Hive** comprises one or more *servers*, each with a **Service** governing one or more **Workers**, where *Workers* ≤ *Processors* – 1.

Each worker maintains the in-memory repository for one or more columns. It is single-threaded; it fetches requests from the socket on which it listens — known as its *front door* — and answers via the same socket. Requests that cannot be satisfied without a blocking call to a system service are "parked"; other requests are serviced until notification arrives (via the front door) that the parked connection is ready to be resumed.




The **Logger** is an object instantiated by the worker. It spawns two sub-processes, each connected by a pipe: the **Flusher** and the **Checkpointer**.

To process a `write()` function call (invoked by the worker), the logger updates the WAL memory image and *does not call* the **Flusher** Rather, it calls `msync(2)` with the flag setting `MS_ASYNC`. On Linux, this schedules the pages for synchronization with disk, cf. *The Linux Programming Interface*:

> If we take no further action after an `MS_ASYNC` operation, then the modified pages in the memory region will eventually be flushed as part of the automatic buffer flushing performed by the *pdflush* kernel thread....

The **Flusher** exists solely to handle an explict `Flush()` call, provoked by a Flush request from the client. The Flusher calls `msync(2)` with the flag setting `MS_SYNC`. When the call returns, it posts a Flush-Done message to the Worker's front door.[1]

The **Checkpointer** listens on its pipe for explicit Checkpoint requests from the Logger, provoked by one of:

- explicit Checkpoint requests from the client
- *checkpoint pressure* detected by the Logger
- a user-defined timeout, subject to **Server** idleness

The Checkpointer never writes to Logger memory. Rather, it communicates with the Logger via a pipe. Explicit and pressure-induced checkpoint requests are written to the Checkpointer pipe; timeouts occur if no explicit requests arrive.

Checkpoint pressure is a function unapplied non-inserts — number, size, and age — plus Hive activity. How exactly these combine to provoke a "checkpoint situation" is unclear at this time, but we know that algorithm will belong to the Logger, because it has the most information about the WAL status and can use the Worker's connection to the Service to obtain Hive status.

On completion (regardless of why) the Checkpointer will have changed the name of the new column store (and the status of the old one), as well as the last checkpointed revision for the column. To avoid entangling the Worker uselessly, that information is posted to the Logger-Checkpointer pipe, where in due time the Logger will read it and update its books, notifying the Worker if appropriate. This suggests that the prelude in every Logger function body include a loop to process completion status messages on the Checkpointer pipe.[2] The Logger notifies the Worker via the front door of completed explicit checkpoint requests.

To move data from the Worker's repository to the on-disk column store, the Checkpointer acts much as a client: it issues a range request (with a for-checkpoint indicator) for all rows in the repository. The Worker responds by first freezing the repository and then, per normal, sending sets of rows out its front door.

**Notes**

Each worker is responsible for N column stores, each cached in a Repository.

Each worker uses a Wal object to capture transactions to a log. The Wal object updates the WAL as a memory-mapped file. On construction the Wal object creates a separate thread of control, known as the Flusher, and a pipe to communicate with it. The Flusher blocks against a read to the pipe.

When the client sends a Change set to the worker, the work calls Wal::Write, which reacts as follows:

1. Update the memory-mapped WAL image
2. Return OK to the worker

When the client sends a Flush request to the worker, the worker calls Wal::Write(), parks the connection, and returns. Wal::Write acts as follows:

1. Write a *sync* request to the Flusher pipe
2. Return OK to the worker
3. Flusher reads sync request
4. Flusher calls OS sync function e.g. **msync(2)** and blocks
5. Flusher sends Flush_OK to the worker's socket

[1] No attempt is made to flush *parts* of the WAL; a flush for any column managed by a worker is, in effect, a flush for all of them.

[2] N.B. Fancier arrangements could be used: the Checkpointer could update the Logger's memory, or the Logger could use a signal handler to read the completion status messages. These both introduce volatility to the Logger's memory, and injure the worker's single-threaded cache. By checking the status messages at the start of each function call, we keep the Logger single-threaded and avoid a set of race conditions.

On receipt of Flush_OK, the worker unparks the connection and returns OK to the Client.

The Flusher is required to sync the memory-mapped image periodically, perhaps every 500 milliseconds. (Tunable. Another option might be every N pages or 30 seconds, whichever came first.) The is effected by a timeout on reading the pipe using e.g. **poll(2)**. On timeout, the Flusher reacts as to a Flush request, except that no notification is sent to the worker.