Tiered Storage

Warning

The WiredTiger Tiered Storage support has not been developed, tested or tuned in production scenarios, and should be viewed as experimental code.

Background

A tiered table is a single btree (possibly with multiple checkpoints) made up of blocks that reside in one or more files on the local file system and zero or more cloud objects (e.g., stored in S3). The btree spans this set of files and objects by encoding a block's location in the pointer that links blocks in the btree. A tiered btree is contrasted with a regular (non-tiered) btree that is contained in a single file. It is also different from an LSM (see **Log-Structured Merge Trees**) tree, which WiredTiger implements as multiple regular btrees, each in its own file.

When in use, a tiered btree, like a regular btree, may have some recently used or modified data that resides in memory pages. This in memory representation is the same between a tiered btree and a regular btree, it is only how data is stored on disk and in cloud objects that makes these btrees different.

All writes (i.e., pages that are reconciled and written to storage by eviction or checkpoint) are written to a designated file, called the active file. All other files and objects that are part of the tiered table are read-only. Each object or file is given an object number, which appears as part of its name on disk, cloud and in metadata.

When a tiered btree is created, there is a single file associated with it, it is given object number 1. This is the active file, and it only resides on disk. This is always true for an active file. When a file ceases to be an active file, writes to it will no longer be permitted. At that point, such a file can be copied to cloud storage. How this transition to a new active file occurs is the subject of the next section.

The flush_tier operation

A fundamental operation for a tiered table is *flush_tier*, this occurs as a result of a call to **WT_SESSION::checkpoint** with a configuration string that includes `"flush_tier=(enabled=true)"`. Because flush_tier is a part of checkpoint, it is a system-wide operation. Thus, flush_tier allows the data for all tiered btrees to move from the local file system to object storage.

The flush_tier checkpoint does all the operations normally associated with a checkpoint, and in addition:

1. identifies any tiered btrees that have changes since the previous checkpoint
2. for each such modified btree:
    a. eviction is temporarily disabled on this btree, while waiting for any writes to the active file to drain
    b. a new empty file is created named with the next object number, this will become the table's active file
    c. switch the table's active file
    d. enable eviction on this btree
    e. queue the old active file to be written to object storage in the background
3. when every modified btree has finished switching in this way, the flush_tier part of the checkpoint is finished. Writes to object storage are likely to be incomplete at this stage, they are completely asynchronous.

As each object storage write completes, the local file that was the source of that write will be removed. The removal is coordinated with any active readers of that file. Since the file is no longer an active file, by definition there can be no outstanding writers to the file, only readers.

A user of the WiredTiger API does not need to be aware of this coordination. Cursors may remain open on a tiered table, and reads, writes and transactions may occur as usual, unaffected by the machinery of the flush_tier. Generally the changes to an application are minor:

- indicate on a **wiredtiger_open** call that tiered storage is being used, and specify the storage provider and bucket name.
- for tables that should not participate in tiered storage, or have a different storage provider or bucket, indicate on the **WT_SESSION::create** calls for that table.
- modify the application's checkpoint thread to include `"flush_tier=(enable=true)"` on calls to **WT_SESSION::checkpoint**. `flush_tier` need not be specified on every checkpoint call; typically the cadence of `flush_tier` is much less than the cadence of ordinary checkpoints.

Example with timeline

The figure below provides a high level overview of flush_tier, showing the state of a single tiered table over time. The green arrows at the top of the figure indicate write requests. These come from eviction, except during checkpoint processing. At T0, we start with two objects in object storage, Object 1, and Object 2, and File 3 as the writable active file. At T1, we start a flush checkpoint. At T2, that checkpoint completes, and WiredTiger switches the writable active file from File 3 to File 4. At this point there should be no outstanding writes to File 3 because the checkpoint has completed and eviction to the table is disabled. All future writes will go to File 4. So we can queue the copy of File 3 to object storage. The copy completes at T3. At any later time (T4) WiredTiger can remove File 3 from the local file system.

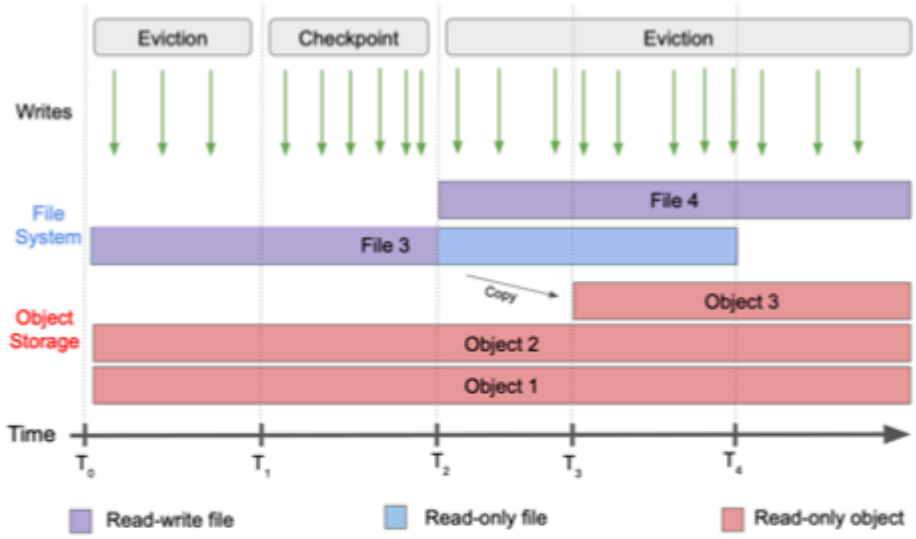

Figure 1: Flush tier operation.