Sync Ups

Aug 31, 2026

- Confirmed the bug I was talking about last friday (setting prune_ts to ckpt N + 1 and then still acquiring a dhandle for N) with
- Created two tickets for the reproduced problems:
    - https://jira.mongodb.org/browse/WT-18532
    -

Aug 28, 2026

(Ivan)

- The checkpoint pickup side:
    - AFAIS in `__wt_conn_dhandle_find` we skip OUTDATED dhandles for checkpoints only if `dhandle->session_inuse != 0`
    - So it seems that outdating a dhandle on the checkpoint pickup actually doesn't prevent us from reopening it
    - I am wondering whether we can mode the logic that OUTDATES dhandle to the place where we close the last cursor so session->inuse becomes 0. Will it help us to acquire a dhandle lock without performance problems?
- The step-down side
    - The step-down side is a bit different since for checkpoint pickup we also update metadata (which closes some gaps)
    - Also, for the checkpoint pickup logic outdated is not so important because checkpointed stable btrees never produce dirty content so we should not care about the eviction
    - We should consider outdating live btrees during step-up separately and check what could happen if we miss the OUTDATED flag that was set on any other read side
        - Especially on cursor open (which is probably find since step-down holds schema + checkpoint lock)
        - And especially on the eviction side (which might be OK as well since the step-down takes the exclusive eviction lock)

Aug 27, 2026

(Ivan)

The PR that introduced the checkpoint pickup F_SET(OUTDATED) - https://github.com/wiredtiger/wiredtiger/commit/8a7c668c28f0e9ca7b3599ec02ec4ed83dd99fac

The problem
It seems like there is no simple solution to this problem. Before DisAgg, dhandle flags were set either in places where no other threads could access them concurrently, or by the sweep server, which always tried to acquire a write lock and backed off if the lock was already read-acquired by another thread.

By setting the OUTDATED flag during the checkpoint pickup process, we introduced a very new situation where we force a dhandle flag change in the middle of the application cycle. In this case, we don't have "try" semantics – we just always set the flag.

Simply doing this under a write mutex is not possible because cursors acquire the read mutex for their entire lifetime. Waiting for the write mutex while still holding the checkpoint mutex could therefore cause significant performance degradation, which mostly manifests as timeouts in tests and looks like a deadlock. However, we might be overlooking something here, and there could potentially be an actual deadlock as well.

Possible solutions
At the moment we have many ideas on how this problem might be potentially solved including:

- Introducing a new mutex
- Using atomics
- Postponing outdating dhandles during step-down/checkpoint-pickup

But all these solutions require deeper understanding of the problem and careful evaluation of whether they are feasible and not breaking any other parts of the logic.

Next steps
Given the explanation above, I don't think this is an easy issue to solve. I believe the next step should be to take a step back and understand the overall logic around dhandle flags, including what depends on seeing or not seeing particular flags.

The focus should be on risk assessment: we need to understand whether this issue is actually serious enough to be considered a release blocker.

One particularly interesting question is why we haven't seen any consequences from this issue in the ~18 months since the initial commit, despite running a variety of testing workloads.

We should also start looking beyond WT-18452 and investigate WT-18334, since it appears to be a very similar issue, but originating from step-down.

Relevant Tickets

Primary Tickets

- AF-19791 Invalid access at address when running aggregation with $collStats (incomplete stack trace)
- WT-18452 Unlocked WT_DHANDLE_OUTDATED flag update in __wti_conn_dhandle_outdated races dhandle close
- WT-18334 Step-down's btree marking loop races the sweep server on stable dhandle flags
-

Past tickets:

- WT-11099 Fix unsafe setting of dhandle flags
  - Don documented that we should **exclusively lock dhandles** before setting their flags - it seems like we don't
- SLS-1449 Make sure layered cursors work with step-up
  - Origin of the WT_DHANDLE_OUTDATED flag (Mar 18th 2025)
- SLS-760 Fix __assert_ckpt_matches failure for stable tables in the layered datastore
  - Origin of the __wti_conn_dhandle_outdated helper function

Future tickets:

- WT-18333 WT_LAYERED_TABLE_STEP_DOWN_CREATED is read and cleared without atomics — formal data race
-

Potentially relevant FIXMEs:

- WT-16494 Ensure checkpoint order is strictly increasing across nodes in disagg
  - FYI this is actually in progress right now! ^
  - FIXME comment in `__disagg_update_file_meta:711`
- WT-17772 Consider marking data handles outdated at step-down or step-up
  - FIXME comment in `__disagg_update_file_meta:727`
- WT-17040 Investigate whether the creation of shared metadata is necessary on followers
  - FIXME comment in `__disagg_metadata_table_init:890`
-

Ivan

Sep 1, 2026

My current understanding of a potential solution:

A potential solution can be using two mixed approaches - RW mutex where synchronisation is important (checkpoint pickup + dhandle find) + relaxed atomics where it's protected by something else or not the ordering is not important (e.g. eviction)

So we do the following:

- Move the OUTDATED flag to a separate field (there are different flag fields in in the dhandle structure, can we reuse some of them)
- Introduce a RW mutex to protect this field
    - Dhandle find should acquire read mutex for the interval since reading the flag till increasing session_inuse, so no-one can re-acquire an outdated dhandle because of a TOCTOU problem.
- But some places will be allowed to read it without lock and just use relaxed atomics:
    - Eviction
        - the step-down write path is having an eviction exclusive lock so there is no need to protect it more
        - the pick-up write path that outdates checkpointed dhandles and it seems like the eviction OUTDATED check is mostly not needed for them
        - for the pick-up write path that outdates live btree is quite strange and i am not sure whether it is needed
    - Sweep
        - Sweep just needs to eventually read this flag to make this dhandle a candidate for a sweep

Aug 29, 2026

- Is there any reason to set OUTDATED for readonly btree for eviction? Or we do it only for dhandles opening + sweep on checkpoint-pickup?

```
None
// t1
cursor operation starts
it reads dhandle for checkpoint N (but session_in_use == 0)
```

```
     OUTDATED is not set
thread gets suspended on CPU

// t2
picks up ckpt N + 1
updates metadata to N + 1
sets OUTDATED for dhandle for N
updates LSN
tries to update prune timestamp (currnetly N) -> N + 1

// t1 continues execution on CPU
session_in_use++ (on already OUTDATED dhandle)
continues using dhandle on checkpoint N
```

Current question ->

Could there be a situation when - **Yes it could but was proven for a suspended execution - not a atomics race**

None

```
cursor_tmp opens stable at ckpt N
cursor_tmp leaves

cursor starts and reads checkpoint N from metadata
pickup happens and sets checkpoint to N+1
then cursor tries to open N's dhandle and misses OUTDATED so it
opens the N
```

Common OUTDATED logic - attempt 2

The OUTDATED flag is stored in two places:

- When we apply OUTDATED + READONLY on a step-down to all the live stable btrees
- When we apply OUTDATED to the previous checkpoint on a newer checkpoint pickup

The interesting part is that we never clean the flag once it’s set

What does OUTDATED flag means for different read places:

- For the sweep server - this dhandle is a potential candidate for being discarded
    - If we miss the setting - nothing wrong happens, we just need to read it eventually and once we read it - it all good since it’s never going to be cleared
- For the dhandle read - if it is a live btree or sessions_inuse == 0, don’t return this dhandle, try to open a new one
    - > Open a new one always takes checkpoint + schema + dhandle lock so it’s more serialised than just pulling it out from session cache
    - **WHAT GOES WRONG IF WE MISS THE SETTING:** We can read an old outdated dhandle and increase session_inuse for it
        - We already know it might be a bug for the checkpoint pickup
        - However, it just because we might open an old checkpoint (let’s say checkpoint N) and ingest will be pruned to N+1 because we never change any checkpointed views on stable
        - It might be fine for the step-down since there are no concurrent writers there - need to recheck
- For all the eviction sides
    - Either early returns
    - Or removing the page without persisting anything
    - > That’s synchronised with at least the step-down setting since step-down holds explicit eviction lock.
    - > For the checkpoint pick-up side it might be fine since all the checkpointed-view btrees are READONLY from their creation so not much changes with having an OUTDATED flag
        - > Recheck what actually changes here
- __wt_checkpoint_get_handles - always holds a checkpoint lock - so it is serialised
- Cursor reopen - ???

Aug 28, 2026

Outdated logic

AI answer:

> *OUTDATED = "the metadata this handle was opened from has been superseded: stop handing it to new users, let current users finish, never write its content back, and retire it promptly."*

Outdated synchronisation (for checkpoint pickup)

Reader	Site	Locks held when reading	Synced with writers?
Session cache find	`session_dhandle.c:152`	none (session-private list, shared dhandle)	No
Connection find	`conn_dhandle.c:296` and `:320`	handle-list read lock (writer holds the same read lock)	No
Session sweep	`session_dhandle.c:863`	none (per comment at :854-856)	No
Connection sweep expire	`conn_sweep.c:241` and `:249`	none (lockless list walk)	No
Checkpoint skip	`checkpoint_txn.c:524`	checkpoint lock (whole checkpoint) + dhandle held	**Yes — transitively via checkpoint lock**
Eviction page gates	`evict_page.c:477` and `:544`	none on the dhandle (page refs/hazard pointers)	No (step-down mitigates via `evict_file_exclusive_on`)
Eviction walk gate	`evict_walk.c:1270`	none on the dhandle	No
Split gates	`bt_split.c:2360`, `:2487`, `:2529`	dhandle read lock (cursor op)	No (writer doesn't take it)
CAN_REOPEN at cursor close	`cur_file.c:752`, `:811`; `cur_layered.c:2285`, `:2340`	dhandle read lock (until release)	No

- **Session/connection find** — **benign ???** - retired handle handed out once; reader gets a consistent old generation; tolerated by design ("older btree… ingest pinned longer")..
- **Session + connection sweeps** — **benign** - retirement delayed one sweep interval.
- **Checkpoint skip** — **unreachable:** reader and both writers hold the checkpoint lock.
- **Eviction gates** — **benign.** one page takes the normal path. Pickup-marked trees are readonly (no dirty pages possible ⇒ gate is pure acceleration, benign).
  Step-down-marked trees: contained by `evict_file_exclusive_on` during marking + txn-level step-down write refusal.
- **Split gates** — ??? - one in-memory split on a tree about to be discarded; triggering write already refused at txn level. Contained.

- **CAN_REOPEN at cursor close — Benign????**. handle stays open one extra cycle; next open/sweep retires it.

Checkpoint pickup vs layered cursors:

```
None
__disagg_pickup_checkpoint() -> holds ckpt lock
	__disagg_adopt_checkpoint_meta() -> holds schema lock
		__wt_metadata_update
		__wti_conn_handle_outdated(old_ckpt) // not holding dhandle lock
	__disagg_finalize_checkpoint_data()
		store_release(last_checkpoint_meta_lsn)
		__wti_layered_iterate_ingest_tables_for_gc_pruning -> set prune ts =
N + 1


__clayered_update_stable()
	conn_lsn = load_acquire(last_checkpoint_meta_lsn) -> returned ckpt n
	if (clayered->lsn != conn_lsn)
		__clayered_open_stable_follower
			__wt_meta_last_ckpt_name -> metadata search
			__clayered_open_stable_int -> holds ckpt lock + schema lock
(only for the first dhandle opening)
```

- acquire/release around `last_checkpoint_meta_lsn` guarantee that if a layered cursor has seen conn_lsn for checkpoint N, it'll be able to observe all the events (metadata stores, flags) that were done for this checkpoint pickup before the `last_checkpoint_meta_lsn` was updated.
    - But it does not protect us from observing newer checkpoints that were picked up later, so if we see conn_lsn for checkpoint N, we can see the metadata for checkpoint N+1, but we can actually already outdated checkpoint N+1 and it might have an outdated flag since we are in the middle of picking up checkpoint N+2 (and many other things could happen as well)
- However, opening a stable cursor holds both checkpoint and schema lock, so it's guaranteed that it'll be serialised with any checkpoint pickup
    - If we pickup checkpoint N, but checkpoint N+1 has already come so checkpoint N is already outdated, what will the cursor open return? Will it return EBUSY?
        - Yes - it will return EBUSY and it's guaranteed to see a newer checkpoint next time because of locks synchronisation.

- The interesting part is that we hold these locks only for the first dhandle opening, so if one cursor opened a dhandle successfully (while it didn't have the OUTDATED flag, others might do that even when we picked up a newer checkpoint and marked it as outdated - at least for some time)
    - Yes, and it could happening but it's fine since we skip OUTDATED dhandles in `__wt_conn_dhandle_find` only if dhandle->session_inuse != 0

Summary

WT_DHANDLE_OUTDATED — the model + newly discovered holes

The meaning in one sentence: OUTDATED marks a dhandle whose metadata has been superseded — don't hand it to new users, let current users finish, never write its content back, retire it promptly.

The interesting part is that we never clear the flag once it's set.

Where the flag is set

- On step-down: OUTDATED + READONLY on all open writable disaggregated btrees (conn_layered.c:1399-1408).
- On checkpoint pickup:
    - the previous checkpoint's handle (conn_layered_checkpoint_pick_up.c:718);
    - the live stable handle (conn_layered_checkpoint_pick_up.c:729);
    - the shared metadata table's old checkpoint (conn_layered_checkpoint_pick_up.c:224).
    - **TODO: It’s not clear why do we outdate live dhandles on a checkpoint pickup**
- On startup/reconfigure: the shared metadata URI (conn_layered.c:1017).

What OUTDATED means for different read places

- **Sweep server**: this dhandle is a candidate for being discarded.
    - In disaggregated mode it is the *only* thing swept below the handle-count floor.
    - Outdated stable-checkpoint handles get a short grace period: the prune-timestamp walk and spanning readers still reference them.
    - If we miss the setting — nothing wrong happens; we just need to read it eventually, and once we read it, it's all good, since it's never going to be cleared.
    - Known race here: none as a reader; but sweep is a *writer* of the same flags word in the expire/close path — see problem 2.
- **Dhandle find / session cache**: if it is a live btree or session_inuse == 0, don't return this dhandle, try to open a new one.
    - Exception: an in-use stable-checkpoint handle is still shared — that's intended, it feeds the prune accounting.
    - Opening a new one takes checkpoint + schema + dhandle locks, so it's fully serialised with a pickup; pulling from the session cache is not.
    - **Known race here:** the check (find) and the pin (session_inuse++) are not atomic and nothing revalidates after the open — see problem 1 (reproduced).
    - **TODO: Check whether there are more races in different places which call this.**

- **Eviction / splits**: either early returns (EBUSY) or removing the page without persisting anything.
    - Synchronised with the step-down setting since step-down holds the explicit eviction lock per tree (evict_file_exclusive_on).
    - For the checkpoint pickup side it's fine since all checkpointed-view btrees are READONLY from their creation, so they can never have dirty pages — the gates are pure acceleration.
    - On the step-down side the gates do real work: a tree that was writable in the leader era can still hold dirty pages at marking time, and the follower has to discard them instead of reconciling.
    - No known race.
- **Checkpoint handle gather** (the checkpoint-time skip):
    - Always holds the checkpoint lock, and both writers hold it too — fully serialised.
    - No known race.
- **Cursor reopen**: guarded not by the flag but by other machinery:
    - the role-change generation (WT_GEN_DISAGG_ROLE);
    - the checkpoint-generation pin + deferred adoption (WT_GEN_DISAGG_CKPT);
    - the EBUSY-retry when the resolved checkpoint was superseded mid-open.
    - **TODO: This one needs further investigation!!!**

When it could lead to problems

1. **Pickup, miss at find (WT-18532, reproduced, 5/5)**:
    - The OUTDATED check (find) and the pin (session_inuse++) are not one atomic step in get_dhandle(), and nothing revalidates after the open.
    - A reader suspended between them across a whole pickup opens checkpoint N while the prune walk has already advanced the ingest prune timestamp to N+1 — because we never change checkpointed views on stable — and gets a stale read.
    - Timestamped/non-transactional readers only; snapshot readers pin the checkpoint generation and defer the adoption.
        - **TODO: if non-timestamped readers block checkpoint adoption - how do we adopt new checkpoints? are non-timestamped readers very rare? What's the mechanism there**
2. **Writer-vs-writer (WT-18452 for the pickup callsite, WT-18334 for the step-down marking loop, never observed)**:
    - The plain F_SET races the close/sweep read-modify-write of the same flags word under locks the outdate writer doesn't share.
    - Lost OUTDATED is self-healing; a resurrected OPEN/DEAD is crash-class, not silent.

Potential solutions

1. **Separate the flag from the shared flags word and make check-and-pin atomic.**

    - Move OUTDATED into its own field (e.g. a `dynamic_flags` word) guarded by a dedicated read-write mutex.
    - Writers take the write side; `__wt_session_get_dhandle` holds the read side from the find until `session_inuse` is incremented — a reader can no longer slip between the check and the pin (fixes hole 1).
    - OUTDATED no longer shares a word with `DEAD`/`OPEN`/`DISCARD`, whose writers are already under the dhandle lock (fixes hole 2).
    - **TODO: check whether the lock span (find → cursor open) works for the other readers and writers without new lock-ordering problems.**
    - **TODO: check what the right thing is for step-down specifically — there eviction is already synchronized (the per-tree eviction lock is held during marking), so it may need nothing more than the new write lock, or arguably no change at all.**
2. **Rework the prune_timestamp condition instead.**
    - Advance the prune timestamp only when the checkpoint's dhandle does not exist at all (rather than exists-but-idle) — build GC pacing on dhandle existence instead of `session_inuse`.
    - An idle-but-present handle would hold pruning back, so a late reader resuming onto it always finds its ingest cover intact.
    - Harder: couples pruning progress to sweep timing, changes GC lag semantics, and needs more investigation into how quickly handles actually disappear.
3. **Check the AI proposed idea about** validate-after-open:
    - Acquire-load `last_checkpoint_meta_lsn` after binding the stable dhandle and retry the bind if it advanced past the resolved checkpoint (same pattern as `__txn_snapshot_validate_disagg`, `txn.c:275`).
    - Independent of the WT-18452 flag-atomicity fix; that fix alone does not close this hole.

Proposed next steps

1. **2-3 days** — compile the full picture and understand whether there might be more issues connected to it (finish the cursor-reopen TODO above; audit remaining plain-flags-word writers for the same torn-RMW pattern).
2. **2-5 days** — try to implement the proposed separate-field + mutex approach (solution 1), including checking the find → open lock span against all readers/writers and deciding what step-down needs given its eviction lock.
3. **Testing this class of issues**:
    - **TODO: Think how we can test this kind of issues???**
    - *Torn RMW on the flags word* (problem 2): TSan should catch it directly (concurrent pickup vs sweep/close under a TSan build, e.g. layered follower tests or `format` with a disagg config); check the existing TSan suppressions don't already hide it.

- *Atomicity/ordering gaps* (problem 1): TSan does not help — this is not a data race but a missing higher-level invariant. It needs a deterministic interleaving test: a suspension/timing-stress hook at the exact gap (as in the repro test), plus the diagnostic asserts as tripwires. For a committable version, convert the env-var hook into a proper `timing_stress_for_test` point.

DHANDLE_OUTDATED accesses

DHANDLE_OUTDATED accesses - TBC

WRITES

1. Checkpoint pick-up

Outdate old checkpoints (to protect them from being reopened? - **WT-18532**?)

It also outdates live btrees - why??? - TODO check - **the only functional consumer of pickup-time outdating is the shared HS,** but the full picture is that it marks:

- Shared HS live handle (open on every follower) — needed, load-bearing.
- Shared metadata table — needed as insurance only.
- User tables, closed leftover from a refused open (step-down race loser, direct API attempt) — not needed for correctness.
- User tables, leader-era handle closed before step-down (the step-down gap) — not needed for correctness
- User tables, open live stable handle on a follower

2. `__disagg_mark_btrees_readonly_then_step_down`

Marking all the stable dhandles OUTDATED and all the stable btrees - readonly

It does mark only OPEN dhandles, but it is fine, because `__disagg_step_down` and newly opened dhandles will see the new role + flags set on currently open ones.

READS

1. `__wt_conn_dhandle_find()`

Why is it needed? Why can't we just always skip the dhandle as soon as we see a flag? TODO - ask Thomas

Initial commit : https://github.com/wiredtiger/wiredtiger/pull/13074

```
C/C++
          if (F_ISSET(dhandle, WT_DHANDLE_OUTDATED)) {
               /*
```

```
                * An outdated read-only stable handle is only safe to reuse in
its checkpoint-view
                * form (a "...wt_stable/WiredTigerCheckpoint.N" name): those
checkpoint handles
                * span eras and sweep keeps them alive for readers and the
checkpoint tracking
                * logic. A live stable handle must be reopened fresh instead --
draining through an
                * outdated one hits resident inserts or unresolved prepared
updates left by the
                * previous leader era.
                */
               if (WT_DHANDLE_BTREE(dhandle) &&
                 F_ISSET_ATOMIC_32((WT_BTREE *)dhandle->handle,
WT_BTREE_READONLY)) {
                   if (__wt_atomic_load_int32_acquire(&dhandle->session_inuse) ==
0 ||
                     !WT_URI_IS_STABLE_CHECKPOINT(dhandle->name))
                       continue;
               } else
                   continue;
           }
```

2. __wt_btree_is_outdated_disagg

Is used for all the eviction accesses. Checks READONLY first.

- It's not clear why it is DISAGGREGATED || READONLY - it is a bug, confirmed by Thomas.
    - https://github.com/wiredtiger/wiredtiger/pull/14588 - Would it be enough to check only one of these flags? - TODO - talk with Thomas and Chenhao
- READONLY has become atomic 2 weeks ago (https://github.com/wiredtiger/wiredtiger/pull/14416) It seems like READONLY was made atomic because of other places and this one is expected to be synchronised by other sync (btree creation, step-down exclusive eviction lock) so it should be fine to use relaxed load there
    - But we do have https://jira.mongodb.org/browse/WT-18533 that's happening solely because of READONLY so maybe this synchronisation is not enough. TODO: Follow up with Thomas

```
None
static WT_INLINE bool
__wt_btree_is_outdated_disagg(WT_SESSION_IMPL *session)
{
    return ((F_ISSET(S2BT(session), WT_BTREE_DISAGGREGATED) ||
              F_ISSET_ATOMIC_32(S2BT(session), WT_BTREE_READONLY)) &&
      F_ISSET(session->dhandle, WT_DHANDLE_OUTDATED));
}
```

3. `__sweep_expire`
4. `__wt_conn_dhandle_find` and `__session_find_dhandle`
5. `WT_DHANDLE_CAN_REOPEN`
6. `__wt_conn_btree_apply`
7. `__wt_checkpoint_get_handles`

Holds a checkpoint lock so never happens simultaneously with any writes

Alex

DHandle Read Lock

`__wt_session_get_dhandle` may return success while keeping the dhandle's read lock held. So active readers hold the dhandle's read lock for their lifetime.

C/C++

Here's the full diagram with every exit point that returns 0 holding the read lock marked:

```
session.open_cursor(uri)
  └─ __session_open_cursor (session_api.c)
      └─ __wt_curfile_open (cursor/cur_file.c)          [or the layered equivalent]
          └─ __wt_session_get_dhandle(session, uri, NULL, cfg, 0)        session_dhandle.c:956
              │
              ├─ __session_get_dhandle(session, uri, checkpoint)       :966  (find only, no lock)
              │     → session cache hit or connection-list find + ACQUIRE reference
              │
              └─ __wt_session_lock_dhandle(session, flags=0, &is_dead) :970
              │     │
              │     ├─ already exclusively owned by this session (:213-216)
              │     │     → excl_ref++, return 0            [write lock held from before]
              │     │
              │     ├─ handle DEAD (:229)                   → return 0, no lock, is_deadp
              │     ├─ btree SPECIAL_FLAGS (:237)           → return EBUSY, no lock
              │     │
              │     ├─ handle OPEN, ordinary wanted (:248):
              │     │     __session_dhandle_readlock()      :249   ← read lock taken
              │     │     ├─ DEAD under it (:250)           → readunlock :252, return 0, no lock
              │     │     └─ open && !exclusive (:257-258):
              │     │           return (0)  ★ READ LOCK HELD — direct ordinary-open exit
              │     │
              │     └─ try_writelock (:268):
              │           ├─ DEAD under it (:270)           → writeunlock :272, return 0, no lock
              │           ├─ opened meanwhile, ordinary wanted (:279-282):
              │           │     writeunlock :281, continue ──loops back to :249──┐
              │           │                                                      │
              │           │          (reaches :257-258 on the next pass)  ★ READ LOCK HELD
```

```
              │           │
              │           └─ exclusive wanted / closed handle (:286-291):
              │                 EXCLUSIVE set, excl_session=session, return 0  [WRITE lock held]
              │
              ▼  back in __wt_session_get_dhandle
              │
              ├─ is_dead → SKIP_OPEN ? EBUSY : continue (retry find)          :971-975
              ├─ SKIP_OPEN && !OPEN (:982-986):
              │     __session_dhandle_exclusive_unlock :984 → return EBUSY, no lock
              │
              ├─ open in wanted mode (:988-991): break
              │     → asserts :1049-1054 → return 0 :1059   ★ READ LOCK HELD (preserved from
              │                                                __wt_session_lock_dhandle :257-258)
              │
              └─ not open (we hold it EXCLUSIVE to open for real):
                    ├─ no schema lock held (:1003):
                    │     __session_dhandle_exclusive_unlock :1004  ← write lock dropped
                    │     recurse __wt_session_get_dhandle under checkpoi

nt+schema locks :1027-1032
                    │     → recursive call lands on the ordinary path
                    │       → returns 0 :1034                    ★ READ LOCK HELD (via recursion)
                    │
                    └─ schema lock held: __wt_conn_dhandle_open :1037
                          ├─ EXCLUSIVE wanted: break            [WRITE lock held] :1038-1039
                          └─ ordinary wanted (:1041-1046):
                                __session_dhandle_exclusive_unlock :1045  ← write lock dropped
                                continue ──retries──► :970 → :249 → :257-258
                                    → break :989 → return 0 :1059  ★ READ LOCK HELD
```

So the complete set of "returns 0 with the read lock held" sites:

session_dhandle.c:257-258 (in __wt_session_lock_dhandle) — the single point where the read lock is taken-and-returned. Reached three ways: directly on an open handle, after the :279-282 write-lock downgrade, and after the :1045 exclusive-unlock retry.

session_dhandle.c:989-991 → :1059 (in __wt_session_get_dhandle) — the outer exit that preserves it (ordinary mode).

session_dhandle.c:1027-1034 — the schema/checkpoint-lock recursion, which hands back the same read-locked state from the inner call.

```
The mirror image — the held read lock's only release point for cursor lifetime:
cursor.close()
  └─ __curfile_close (cur_file.c:639)
      └─ __wt_session_release_dhandle(session)            :704 / :726
          └─ __wt_session_release_dhandle_v2              session_dhandle.c:314
              └─ __session_dhandle_readunlock             :384   ← the read lock finally drops
                 (or via the cache path: __curfile_cache, cur_file.c:726)
```

Flag setting without exclusive lock

C/C++

Write-side audit (every F_SET/F_CLR on dhandle->flags on master)

Violations — no dhandle lock at all, only handle-list read lock + a references refcount:

Site	Flag
__wti_conn_dhandle_outdated, conn_dhandle.c:363	WT_DHANDLE_OUTDATED
Step-down marking loop, conn_layered.c:1408	WT_DHANDLE_OUTDATED

Compliant (write lock held, or pre-publication):

- __wt_conn_dhandle_open :705 (OPEN), __wt_conn_dhandle_close :539 (DEAD), :555 (OPEN clear) — all under the dhandle write lock.
- session_dhandle.c:286/358/369/75 (EXCLUSIVE, DISCARD* transitions) — these are the lock/unlock paths, write lock held.
- conn_dhandle.c:889 (DROPPED) and meta_track.c:194, session_dhandle.c:1089, schema_drop.c:386 (DISCARD) — drop/overwrite paths, dhandle held exclusively (schema lock + exclusive dhandle lock; the comments say so explicitly, e.g. "We lock checkpoint handles that we are overwriting").
- conn_dhandle.c:242 (IS_METADATA), bt_handle.c:672,678 (HS, DISAGG_META) — set at handle creation/open, before the handle is visible to concurrent readers.
- Sweep's close path honors it: __sweep_expire_one takes the dhandle try-write-lock specifically to comply (conn_sweep.c:213), deliberately without setting the EXCLUSIVE bit.

Read-side: the corollary is broadly not honored (by design)

Unlocked readers are everywhere: the session-cache sweep reads flags with "no locks in place, other than the data handle reference" (session_dhandle.c:853-857, an acknowledged unlocked read); __session_find_dhandle (:150-157), the sweep server's lock-free list walk (conn_sweep.c:241-308), and __wt_conn_dhandle_find (handle-list lock only, no dhandle lock). Eviction/checkpoint readers do hold the dhandle read lock — but that only excludes writers who actually take the write lock, which the two violating sites don't.

The reason this held together for decades: with the write rule intact, all writers serialize on the write lock, so no two read-modify-writes can interleave; plain unlocked reads against locked plain writes are technically UB but practically benign on x86 (no tearing on a uint16). The two OUTDATED sites break the writer-writer exclusion the whole pattern depends on — that's the lost-bit mechanism behind WT-18452/18334. And the fix direction has precedent in the same file: WT-18015 already converted the concurrently-modified WT_BTREE_* flags (the >0xfff range sharing this word's combined usage) to F_SET_ATOMIC_32 — note the loop's own F_SET_ATOMIC_32(btree, WT_BTREE_READONLY) two lines above the violating store.

Negative Proof

Hypothesis: WT-18452 causes AF-19791

Requires:

1. __wti_conn_dhandle_outdated() or a similar unprotected dhandle flag setting can race with sweep.
2. The dhandle flags determine whether a dhandle is safe to be reopened.
3. This race can result in a state where the dhandle's memory is freed, but flags indicate it can still be opened.
4. If the flags are in such a state, then we can mistakenly open a freed dhandle and hit the AF-19791 segmentation fault.

Negative Proof: Show that 1 and 2 are true, but 3 and 4 are false. Therefore WT-18452 cannot cause AF-19791.

1. An unprotected dhandle flag write can race with sweep closing a dhandle.

The table below shows all code locations where dhandle flags are set and cleared. The bolded entries are **__wt_conn_dhandle_close**, the function called by the sweep server when closing a dhandle, and the two locations where the **OUTDATED** flag is set. We can see that these are the *only* two locations where the dhandle flag is modified without a lock.

Therefore, these are the only two locations that could possibly race with sweep, setting a dhandle's flag while it is being closed. All other locations are mutually excluded.

Calling Function	Write	Site	Synchronization
__wt_conn_dhandle_alloc	init (IS_METADATA)	conn_dhandle.c:242	handle not yet visible
__wt_conn_dhandle_close	**F_SET(DEAD)**	**conn_dhandle.c:541**	**dhandle write lock + close_lock**
__wt_conn_dhandle_close	**F_CLR(OPEN)**	**conn_dhandle.c:557**	**dhandle write lock + close_lock**
__wt_conn_dhandle_open	F_SET(OPEN)	conn_dhandle.c:707	dhandle exclusive lock

__wt_btree_open	F_SET(HS), F_SET(DISAGG_META)	bt_handle.c:678/:684	dhandle exclusive lock
__wt_session_lock_dhandle	F_SET(EXCLUSIVE)	session_dhandle.c:286	dhandle write lock
__wt_session_release_dhandle_v2, __session_dhandle_exclusive_unlock	F_CLR(EXCLUSIVE)	session_dhandle.c:369/:75	dhandle write lock
__wt_session_release_dhandle_v2	F_CLR(DISCARD|DISCARD_KILL)	session_dhandle.c:358	dhandle exclusive lock
__conn_dhandle_close_one	F_SET(DROPPED)	conn_dhandle.c:891	dhandle write lock, exclusive
__wt_session_lock_checkpoint, __meta_track_unroll, __wt_schema_drop	F_SET(DISCARD)	session_dhandle.c:1089, meta_track.c:194, schema_drop.c:386	dhandle write lock, exclusive
__wti_conn_dhandle_outdated	**F_SET(OUTDATED)**	**conn_dhandle.c:365**	**none**
__disagg_mark_btrees_readonly_then_step_down	**F_SET(OUTDATED)**	**conn_layered.c:1408**	**none**

2. The dhandle flags determine whether a dhandle is safe to be reopened.

What flag state is required for the dhandle to be considered (re)openable?

Calling function	Requires !DEAD	Requires !OUTDATED	Requires OPEN	Notes

__wt_conn_dhandle_find (live lookup), conn_dhandle.c:294-312	**yes**	**yes**	no	Carve-out (WT-17817): an outdated readonly stable checkpoint-view handle may still be reused while session_inuse > 0 - and close requires session_inuse == 0, so the carve-out can never bind a closed btree
__wt_conn_dhandle_find (checkpoint-name lookup), conn_dhandle.c:320	**yes**	**yes**	no	
__session_find_dhandle (session cache), session_dhandle.c:152	**yes**	**yes**	yes, or EXCLUSIVE	Discards INACTIVE || OUTDATED entries (INACTIVE = DEAD || !(EXCLUSIVE | OPEN)); metadata exempt
__wt_session_dhandle_sweep, session_dhandle.c:858-873	**yes**	**yes**	yes, or EXCLUSIVE	Also discards on idle timeout; prunes the session cache only
__wt_session_get_dhandle, session_dhandle.c:959/:970/:978	**yes**	no (but gated upstream by find/cache)	yes, to use as-is	SKIP_OPEN && !OPEN → EBUSY; otherwise reopens under the exclusive lock
__curfile_reopen_int (cached file cursor), cur_file.c:752	**yes**	**yes**	**yes**	CAN_REOPEN requires exactly OPEN among {DEAD, DROPPED, OPEN, OUTDATED} (dhandle.h:64), so DROPPED is excluded too; failure releases the handle and fails the cached cursor
__curfile_reopen (sweep-check only), cur_file.c:811	**yes**	**yes**	**yes**	!CAN_REOPEN → the cached cursor is swept
__clayered_reopen_int / __clayered_reopen, cur_layered.c:2285/:2340	**yes**	**yes**	**yes**	CAN_REOPEN on the stable dhandle, same semantics

__wt_conn_btree_apply walks (checkpoint gather), conn_dhandle.c:840/:859	**yes**	**yes**	**yes**	The gather skips anything else

So we always require both !DEAD and !OUTDATED before reusing or reopening a dhandle.

3. The race can result in a state where the dhandle's btree is closed but flags indicate it can still be opened.

Even if __wti_conn_dhandle_outdated or __disagg_mark_btrees_readonly_then_step_down does race with __wt_conn_dhandle_close, the resulting corrupted flags still are not put into a state that allows the dhandle to be reused/reopened.

Calling function	Last store (wins the race)	Resulting flags always...
__wti_conn_dhandle_outdated, __disagg_mark_btrees_readonly_then_step_down	F_SET(OUTDATED)	has OUTDATED
__wt_conn_dhandle_close (expire, mark_dead=true)	F_SET(DEAD)	has DEAD
__wt_conn_dhandle_close (discard, mark_dead=false)	F_CLR(OPEN)	lacks OPEN

All possible orderings still leave the dhandle marked as either DEAD or OUTDATED, and both are required to be false.

Race	Vs expire close (F_SET(DEAD))	Vs discard close (F_CLR(OPEN))

Mark stores last	S|OUTDATED - close's DEAD lost	S|OUTDATED - close's OPEN-clear lost (OPEN resurrected)
Close stores last	S|DEAD - mark's OUTDATED lost (moot; sweep reprocesses)	S\OPEN - mark's OUTDATED lost (closed, goes away)

4. If the flags are in such a state, then we can mistakenly open a closed dhandle and hit the AF-19791 segmentation fault.

The first part of this statement is true, but even this does not lead to the AF-19791 strstr segmentation fault.

If we artificially corrupt the flags to a state where the dhandle appears OPEN, !DEAD and !OUTDATED, allowing it to pass the WT_DHANDLE_CAN_REOPEN check, then we get a different segmentation fault when a reader tries to read via the closed handle:

Core Dump:

```
#0 __wt_row_search (leaf=0x0)   src/btree/row_srch.c:434   if (page->type != WT_PAGE_ROW_INT)
#1 __cursor_row_search          src/btree/bt_cursor.c:512
#2 __cursor_search              src/btree/bt_cursor.c:528
#3 __wt_btcur_search            src/btree/bt_cursor.c:927
#4 __curfile_search             src/cursor/cur_file.c:325
```

This actual strstr callsites are enumerated below, with justifications for why a flag corruption could not trigger the AF in each case:

Calling function / site	strstr argument	Segfault downstream of flag corruption?

__wt_conn_dhandle_find carve-out (conn_dhandle.c:308), __sweep_expire (conn_sweep.c:244) - WT_URI_IS_STABLE_CHECKPOINT	dhandle->name	No - name is allocated at __wt_conn_dhandle_alloc and freed only at __conn_dhandle_destroy, which is gated by the atomic references count, not the flags word; a referenced handle's name outlives any flags state
__wt_btree_open (bt_handle.c:610), __prepared_discover_open_ingest_cursor (prepared_discover_walk.c:38), ingest/layered asserts (conn_layered_ingest.c:331, cur_layered.c:1240) - WT_URI_IS_STABLE*	dhandle->name	No - same
__wt_btree_shared_base_name (btree_inline.h:331)	name: dhandle->name (bt_handle.c:96, conn_dhandle.c:56) or a metadata filename (meta_ckpt.c:857)	No - freshly read metadata buffers are caller-controlled
__wt_session_get_dhandle checkpoint-lock test (session_dhandle.c:979)	uri - the URI being opened; in the layered stats path, built into a scratch buffer from layered->stable_uri first	No via flags - the scratch buffer is always valid; a NULL stable_uri (freed layered field) faults earlier, in the __wt_buf_fmt - a different frame

hs_verify.c:185/:226, bt_vrfy.c:270/:581, schema_create.c:89, btree_inline.h:306, conn_layered.c:1610, prepared_discover_walk.c:20 - WT_URI_IS_STABLE*	metadata values, caller URIs, directory entries	No - all freshly read or local buffers; only indirectly if a corrupted metadata dhandle served garbage content (second-order, the stale-read class)
live_restore_fs.c:1800, err.c:935	live-restore paths, build-path formatting	No - unrelated to dhandles