TECHNICAL WHITE PAPER

Crash-Safe Change Data Capture

A change-data-capture pipeline is only as trustworthy as the moment it decides "this change is safely delivered — I can move the checkpoint forward." Move it too early and a crash silently drops a committed change. faucet-stream advances the replication checkpoint only from a durable bookmark, only after the sink has flushed — never from decoded WAL or a keepalive. This paper describes that rule, proves it for Postgres, and is honest about what the guarantee is (no loss) and is not (no duplicates, by default).

faucet-stream · ~13 min read · Grounded in code & crash-resume tests

Online version: https://faucet-hq.github.io/papers/crash-safe-cdc

---

Abstract

Change data capture turns a database's write-ahead log into a stream of row changes. The hard part is not reading the log — it is knowing when it is safe to tell the database "you may forget everything up to here." That acknowledgement is a promise: the changes below this position are durably delivered. If the pipeline makes that promise before the destination has actually stored the data — because it decoded the change, or because the database sent a keepalive — a crash in the gap loses a committed change, silently and permanently. faucet-stream makes the promise in exactly one place: after the sink flushes a page, it persists a durable bookmark, and only that bookmark is ever fed back to the source. This paper walks through the ordering rule, its strongest expression in the Postgres connector, its shape in MySQL, MongoDB, and SQL Server, and the precise delivery guarantee that results — at-least-once by default, effectively-once when the sink cooperates. It deliberately does not claim exactly-once.

1. The problem: acknowledging too early

Every CDC source has a resume position: a Postgres LSN, a MySQL binlog offset or GTID, a MongoDB resume token, a SQL Server log sequence number. On restart, the pipeline asks the source to continue from that position. So the position is a durability claim — "I have everything before this." The failure mode that matters is advancing it too early:

- **Advancing on decode.** The pipeline reads and parses a change, marks the position, then crashes before the destination stores it. On restart the source resumes *after* the change. It is gone — never written, never re-read.
- **Advancing on a keepalive.** Postgres periodically asks "still there?" and reports its latest WAL position. Echoing that position back tells Postgres it may recycle WAL for changes the consumer has not persisted. A crash then loses them.

Both are easy mistakes — they make the stream look healthy and the lag graph look great — and both corrupt the destination silently. Avoiding them is the whole game.

2. The rule: flush, then persist, then acknowledge

faucet-stream fixes the ordering once, centrally, for every pipeline — not per connector. A page of changes carries a `bookmark`: the resume position that becomes safe *after* this page is durably written. The engine always does the same three steps in the same order:

1. **Write** the page to the sink.
2. **Flush** the sink — block until it confirms the batch is durable.
3. **Persist** the bookmark to a durable state store — and only now is that position eligible to be acknowledged upstream.

> The invariant, stated in the code and fixed by an architecture decision record: **the state store is always at or behind the sink — never ahead.** The only crash window is between "sink committed" and "bookmark persisted." A crash there re-delivers the page on restart. It never loses it.

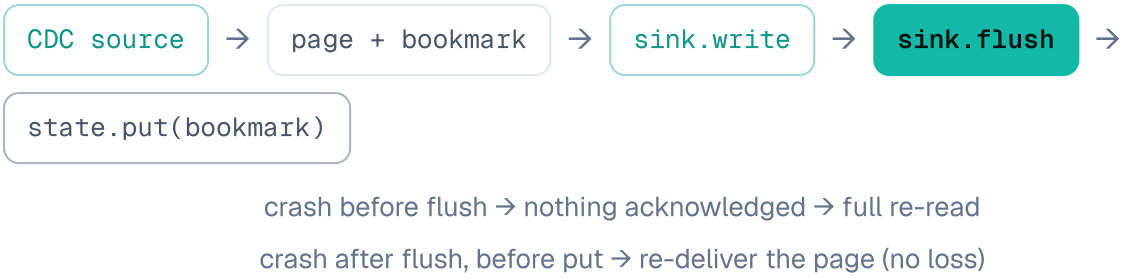


Figure 1 — the bookmark is persisted only after the sink flush; the acknowledgement upstream is derived only from the persisted bookmark. Identical on the row and columnar paths.

3. Postgres: the rule made explicit

Postgres is the one source that acknowledges a position *back* to the database (it must, so the server can recycle WAL and advance the replication slot). That makes it the strictest test of the rule — and the connector enforces it in two deliberate refusals:

A keepalive never advances the checkpoint

When Postgres sends a keepalive with its latest WAL position, the connector absorbs it and pointedly does *not* report that position as applied — doing so would authorize WAL recycling for changes never persisted.

Decoding never advances the checkpoint

Parsing a `COMMIT` record does not move the confirmed position. The comment in the source is blunt: the only durable signal is the bookmark the pipeline persists after the sink flush, which arrives back via the resume path.

So the LSN fed back to Postgres — the `confirmed_flush_lsn` that advances the replication slot before streaming resumes — originates *only* from a durable bookmark. With no persisted bookmark, the slot is not advanced at all. The connector emits one page per committed transaction, carrying that transaction's end-LSN, and lets the central rule in §2 decide when it becomes real.

4. MySQL, MongoDB, SQL Server

The other three sources never acknowledge a position upstream — MySQL and MongoDB stream by pulling, SQL Server by polling — so the "never from a keepalive"

hazard cannot even arise. Their resume position lives solely in the durable state store and advances only through the same post-flush persist:

SOURCE	RESUME UNIT	ACKS UPSTREAM?	MATURITY
Postgres	LSN (logical replication slot, pgoutput)	Yes — confirmed_flush_lsn	Tier 1
MySQL	binlog file+pos, or GTID set	No — pull-based	Tier 1
MongoDB	change-stream resume token	No — pull-based	Tier 1
SQL Server	per-capture-instance LSN map	No — poll-based	Tier 1

- **MySQL** — one page per commit (an empty commit still yields, to advance the bookmark). Even under GTID mode the persisted bookmark records file+position, so resume is unambiguous.
- **MongoDB** — the resume token advances per flushed batch; nothing is persisted until a page yields, so a dropped in-flight buffer is simply re-fetched from the last token.
- **SQL Server** — a map of capture-instance → last-committed LSN; on resume each poll starts at the LSN *after* the bookmark, so a committed change is never re-read within a run.

5. The durable bookmark store

"Persist the bookmark" is only as strong as the store behind it. faucet-stream's state store is a small key/value contract whose `put` must be durable before it returns:

File state store

Writes to a temp file, `fsync`s it, atomically renames it into place, then `fsync`s the parent directory. Genuinely crash-durable.

Postgres state store

A single `INSERT … ON CONFLICT DO UPDATE` — durable per Postgres commit.

Redis state store

A plain `SET`. **Caveat:** durability is only as strong as the Redis server's own persistence configuration — there is no synchronous fsync wait. Prefer the file or Postgres store where bookmark durability is critical.

6. The honest guarantee: at-least-once, upgradable

Here is where many CDC systems overclaim. faucet-stream does not. The crash window in §2 — sink committed, bookmark not yet persisted — *re-delivers* the page on restart. That means the default guarantee is:

> **At-least-once: no committed change is ever lost, but a change may be delivered more than once across a crash.** The codebase does not offer a distributed "exactly-once" — it names the ceiling *effectively-once* and says so in the type system.

Duplicates are removed by pairing a CDC source with a sink that can absorb a replay. All four CDC sources declare deterministic replay, so opting into effectively-once selects one of two mechanisms:

- **Atomic watermark** — an idempotent sink plus a durable state store. The sink's committed token embeds the resume bookmark; on restart the pipeline reads that token and re-anchors the source, skipping pages already committed.
- **Keyed upsert** — the sink deduplicates by primary key, so a replayed row overwrites rather than duplicates.

If neither is available, the engine refuses to pretend: requesting exactly-once without an idempotent or keyed sink is a typed error that names the limiting side, rather than a silent downgrade.

7. Proven, not just asserted

The no-loss claim is checked in CI, not merely documented. The Postgres connector has a test that simulates a crash *before* the flush-and-persist step and asserts the row is **re-delivered, not lost**; companion tests assert that resuming from a bookmark skips already-consumed changes. Every CDC source has a bookmark round-trip conformance test and a "capture, then resume without replay" integration test, and the pipeline itself is tested to persist the bookmark only after the sink succeeds — and to *not* persist it when the sink fails.

8. Limitations & honest scope

- **No double-delivery is *not* the default.** Default delivery is at-least-once; a crash can re-deliver a page. Duplicate-free requires opting into effectively-once with an idempotent or keyed sink and a durable state store.
- **The "never from a keepalive/decode" rule is enforced, non-trivially, only for Postgres** — the only source that acks upstream. For MySQL, MongoDB, and SQL Server the claim holds trivially, because they never acknowledge a position at all.
- **SQL Server has one upstream-retention gap.** If the database's own CDC cleanup job purges change rows before they are read (the resume point falls below the retained minimum), a data gap is possible. This is an upstream retention limit, not a checkpoint-ordering flaw — the connector warns and resumes from the earliest retained change.
- **Redis state is not synchronously durable.** Use the file or Postgres state store when the bookmark must survive a hard crash of the state layer itself.

9. Conclusion

Crash safety in CDC comes down to one discipline, applied without exception: never acknowledge a position you have not durably delivered. faucet-stream keeps that discipline in a single place — flush the sink, persist the bookmark, and only then let that bookmark be acknowledged upstream — so the state store is always at or behind the destination, and the worst a crash can do is make the pipeline repeat itself, never skip. Postgres proves the rule against the hardest case, an upstream that must be told when to forget; the other sources inherit it for free. The result is a

guarantee stated precisely rather than generously: no committed change is lost, duplicates are removed when the sink can help, and the word "exactly-once" is left unsaid because it cannot honestly be earned here.

Was this useful?

React with 👍, ❤️ or 🚀 to vote, or leave a comment. You sign in with GitHub inside the box below; this site never sees your account.

Loading comments…