The Delete That Did Not Delete

A post-mortem on nexus_db: how an obvious flush optimization resurrected deleted keys, and why only compaction is allowed to drop a tombstone.

Abir Deol · University of Alberta, Computing Science · July 2026

Abstract. nexus_db is an embedded log-structured merge-tree key-value store written in C++17, modelled on LevelDB and RocksDB, with a write-ahead log, a skip list memtable, and immutable SSTables on disk. This is a post-mortem on a deletion bug: under a specific and entirely ordinary sequence of operations, deleted keys came back. The cause was an optimization in the flush path that discarded tombstones it judged redundant, using a definition of redundant that only considered memory and ignored every record already written to disk.

1. Deletion in an LSM tree

An LSM tree never modifies data in place. Writes go to a write-ahead log for durability and into an in-memory skip list. When that memtable passes a size threshold it is frozen, a fresh one takes over, and the frozen one is written out as an immutable SSTable sorted by key. Files on disk are never edited afterward, only merged and replaced during compaction.

That immutability creates a problem for deletion. You cannot go and remove the record, because the file holding it is read only, and it may sit under several newer files. So a delete is not a removal. It is an insertion: a marker called a tombstone, written at that key, saying this key is dead as of this sequence number. Reads walk the levels newest to oldest and stop at the first thing they find. If that is a tombstone, the answer is not found, and the older record underneath is shadowed but still physically present.

The consequence, which is the entire subject of this document: a tombstone is not metadata about an operation. It is *data*, and it has to survive to disk with the same care as any value, because the thing it is protecting you from is also on disk.

2. The symptom

A deleted key returned its old value. Not always, which is what made it interesting: delete a key and read it back immediately and the delete held perfectly. The bug needed a particular ordering to show up.

```
PUT   user:1042  ->  {...}      # lands in memtable
<flush>                         # memtable frozen, written to 000031.sst
DEL   user:1042                 # tombstone into the NEW memtable
GET   user:1042  ->  not found  # correct, tombstone is in memory
<flush>                         # this memtable is written out
GET   user:1042  ->  {...}      # the old value is back
```

Reads served from memory were right. Reads served after the second flush were wrong. That narrows the fault to the flush path with no further work, because the tombstone demonstrably existed and behaved

correctly right up until it was written to disk.

3. The hunt

The useful move was to stop reasoning about it and read the SSTable. The files are my own format, so dumping every key and its type flag from the file written by that second flush took a few minutes to write and answered the question outright: user:1042 was not in it. Not as a tombstone, not as anything. The record had entered the flush and had not come out the other side.

At that point the search space is one function, and the offending condition was easy to find once I knew what I was looking for.

4. Root cause

The flush loop iterated the frozen memtable and, for each entry, decided whether it was worth writing. For a tombstone it asked whether there was a live value for that key anywhere in the memtable it was currently flushing. If not, it concluded the tombstone had nothing to shadow and dropped it to save the space.

```
// the bug
for (const auto& e : frozen) {
    if (e.type == kTombstone && !frozen.contains_value(e.key)) {
        continue;              // "nothing to delete", skip it
    }
    writer.Add(e.key, e.value, e.type);
}
```

The condition is a correct test of the wrong question. It asks whether the tombstone shadows anything in *this memtable*. What matters is whether it shadows anything in the entire tree, and the overwhelming majority of that tree is on disk, in files this loop never consults.

So the failure needs exactly the ordering above. Write a key and let it reach disk. Delete it while the tombstone is alone in a fresh memtable, with no accompanying value because the value is in an SSTable now. The flush looks around the memtable, sees no value for that key, decides the tombstone is meaningless, and throws away the only thing standing between the reader and a record that was supposed to be gone. Delete and rewrite in the same memtable and the tombstone gets written, which is why casual testing never caught it.

5. The fix

The flush path is unconditional. Every entry in the memtable is written, tombstones included, without any judgement about redundancy.

```
for (const auto& e : frozen) {
    writer.Add(e.key, e.value, e.type);   // tombstones included, always
}
```

Tombstones are still collected, but not here. Compaction is the only stage that can safely do it, because compaction is the only stage that knows what is underneath. A tombstone may be discarded when the

merge that consumes it also consumes every older record for that key, which is provable only when the compaction reaches the bottom level and no older file can contain the key. Anywhere above that, dropping it is a guess.

This is the rule I should have started from, and it generalizes past this codebase: the component that deletes information must be the component that can see all of it. Flush sees one memtable. It was never entitled to that decision.

6. What it changed

The optimization that caused this was worth close to nothing. Tombstones are a key and a type flag with no value payload, so the space saved was negligible, and I traded correctness of deletes for it. I wrote it because it felt obviously safe, and it felt obviously safe because I was picturing the memtable rather than the tree.

- A skipped write is a write. Any branch that decides not to persist something is part of the durability path and deserves the same scrutiny as the code that writes.
- In a layered store, correctness arguments have to name the layer. "This record is redundant" is meaningless without saying redundant with respect to what.
- Deletion is the operation most likely to be under-tested, because the natural test is delete and read back, and that path is served from memory.

The test suite grew a category it did not have before: sequences that deliberately straddle a flush boundary. Write, flush, delete, flush, read. Write, delete, flush, read. Delete a key that never existed, flush, read. These are the orderings where the memory and disk halves of the store have to agree with each other, and they are exactly the orderings that a quick manual test never produces.