Crash-consistent applications

Key takeaways

1. One logical update becomes several physical writes, and the kernel persists them lazily and in any order. A crash can expose an on-disk state your code never meant to write.

2. Atomicity comes from funneling every write through one commit point: an atomic rename of a complete new file, or a commit record in a write-ahead log.

3. Database recovery restores the database's invariants after a crash. Your index's invariants are yours: recovery is running the update again, which is safe only if rerunning changes nothing.

Common misunderstandings of crash consistency

Which claim would you defend right now? Choose before the evidence.

“If close() returns, the bytes are safely on stable storage.”

“The disk persists my writes in the order my program issued them.”

“After a crash, rerunning an append-only update safely finishes the job.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Your program will be killed mid-write

Design for the kill, not the happy path

A kill signal, an OOM kill, a power cut: the process never gets to finish. Kill an in-place updater halfway through and the index is left in a state your code never intended to write. The next command reads it.

A crash lands in the middle of your write

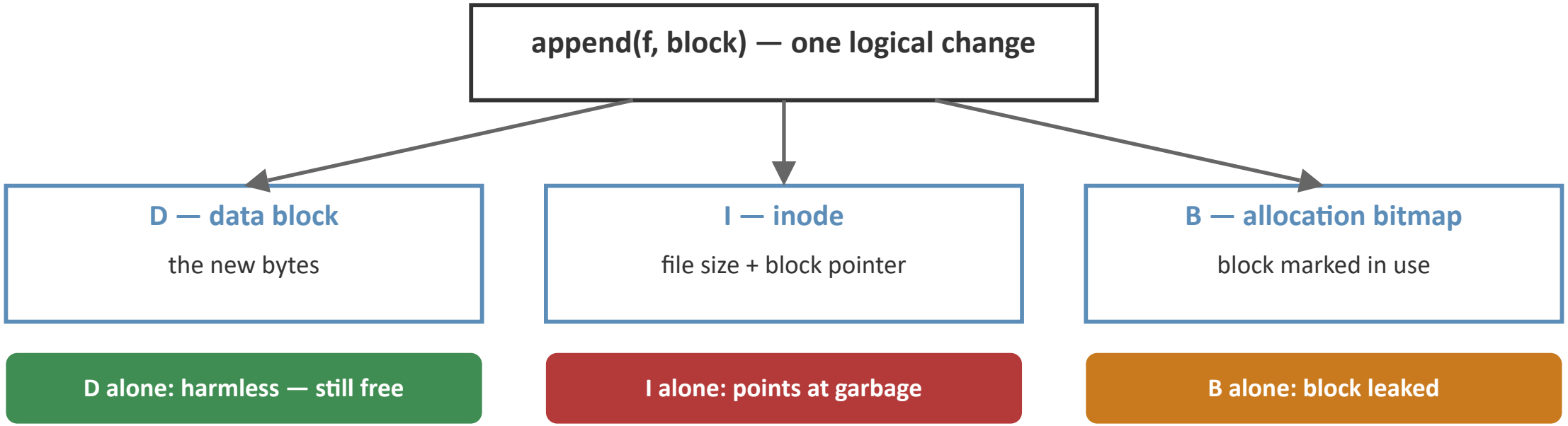
```
$ mindex update --db idx.db mail/    # writing 900 new postings...
[ kill -9 / OOM / power cut ]        # dies after 450

$ mindex search --db idx.db "cxl"    # what does this read now?
#  -> half the postings present? a torn row? an empty file?
```

The crash is the common case. Your on-disk format has to survive it.

One logical update is many physical writes

Even the filesystem has this problem

Appending one block to a file changes three on-disk structures. The kernel writes them back lazily, in any order — a crash persists an arbitrary subset.



Crash consistency: every state a crash can expose must be recoverable to a correct one.

Where a crash lands

An atomic, idempotent update turns every CORRUPT and DUPLICATED point into SAFE.

Instapoll

Instapoll · Review · Sep 10 A

A program writes a new index file, close() returns, and the power fails one second later. What is on disk after the reboot?

- A. The complete new file, because close() does not return until the bytes are written
- B. Possibly none of the new bytes, because close() can leave them in the kernel’s cache
- C. The old file, because the filesystem rolls back any write that did not finish
- D. The new file’s bytes but no directory entry, because renames are what get lost

close() is not durability

close() promises the file descriptor is gone, not that your bytes are on the disk. They can sit in the kernel page cache for seconds. Only fsync forces them to stable storage.

Looks saved — bytes may be RAM-only

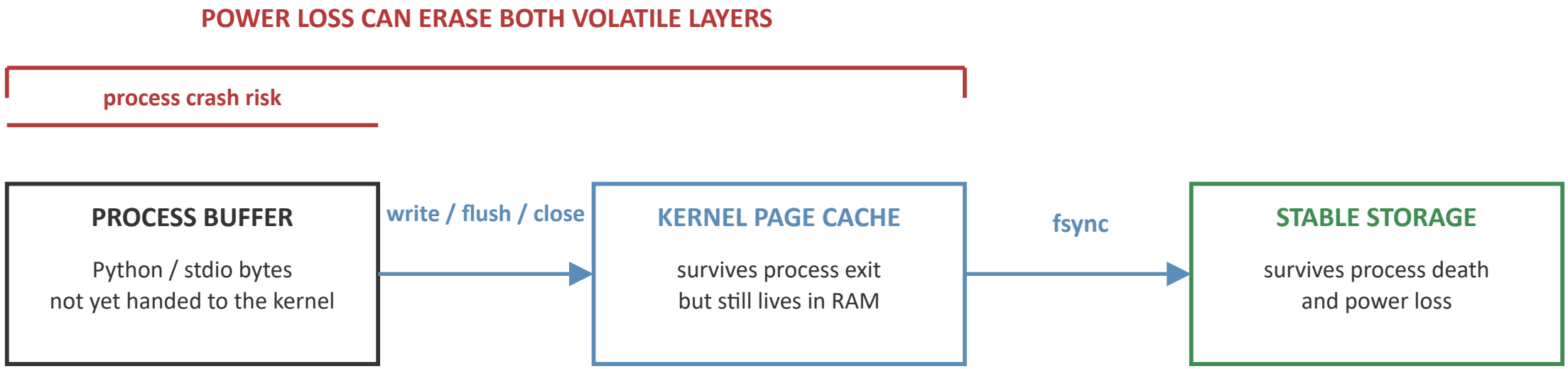
```
with open("idx.db","wb") as f:
    f.write(new_bytes)
# close() returned;
# a power cut here loses it
```

Actually durable

```
with open("idx.db","wb") as f:
    f.write(new_bytes)
    f.flush()
    os.fsync(f.fileno())
# now the SSD has it
```

Two failures erase different layers

close() changes ownership; fsync() changes survival



close() can move bytes out of the process; only fsync moves them beyond the power-loss boundary.

Two ways to buy back atomicity

Funnel many writes into ONE commit point: before it all-old, after it all-new

Copy-on-write: publish by rename

```
write_all("idx.db.tmp")
fsync(tmp_fd)
os.replace(tmp, "idx.db")
#   ^ THE commit point
fsync(dir_fd)
```

Write-ahead log: publish by commit record

```
log.append(intents...)
log.append(COMMIT)
fsync(log)
#   ^ THE commit point
apply_in_place(intents)
# crash? replay the log if it
# ends in a COMMIT that verifies
```

os.replace() issues the rename(2) system call: the kernel is what makes the swap atomic, not Python.

COMMIT counts only once fsync returns. A crash during the append can tear the last record: new bytes over half of it, stale bytes over the rest. So every record carries a checksum, and replay stops at the last record that verifies. Filesystems journal their own metadata exactly this way (ext4, XFS, NTFS).

The durable-write recipe for files

1. write tmp → **2. fsync(tmp)** → **3. rename(2)** → **4. fsync(dir)**

1. write tmp	2. fsync(tmp)	3. rename(2)	4. fsync(dir)
new bytes → idx.db.tmp	force tmp to the SSD	os.replace(tmp, idx.db)	force the rename itself
durable: nothing yet — but idx.db stays intact	durable: the new contents survive a crash	durable: readers see all-old OR all-new	durable: the swap survives a crash too

```
tmp = "idx.db.tmp"
write_index(tmp); os.fsync(tmp_fd)   # 1-2  contents on the SSD
os.replace(tmp, "idx.db")            # 3    rename(2): all-or-nothing
os.fsync(dir_fd)                     # 4    the rename itself is durable
```

The live file is only ever swapped in whole. A crash can’t leave it torn.

A database does this for you

SQLite provides transactions. A transaction starts at BEGIN and tries to commit at COMMIT; SQLite uses a write-ahead log to make it atomic. Database recovery is what runs at the next open after a crash: it keeps every committed transaction, discards every uncommitted one, and so restores every invariant of the database.

ref_mindex.py: all mutations in ONE transaction; crash before COMMIT loses only the batch

```
db = sqlite3.connect("idx.db")
db.execute("BEGIN")              # one transaction
apply_all_changes(db)            # inserts + deletes, many rows
#  <-- crash here: recovery at the next open discards the batch
db.commit()                      # COMMIT: the single durable instant
```

Recovery restores the database’s invariants. Your index’s invariants are yours to restore.

Idempotent updates recover safely

- Your recovery is running update again; a log replay is the same rerun of apply.
- A full rebuild is correct but punishes every crash. Reconcile instead: update-twice == update-once.
- Append blindly and the rerun double-indexes what the crash already wrote.

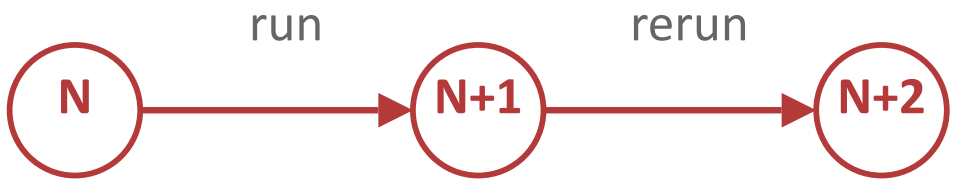
Append-only — rerun duplicates

```
def update(corpus):
    for m in corpus:
        insert_postings(m)
# rerun: every posting twice
```

Reconcile — idempotent

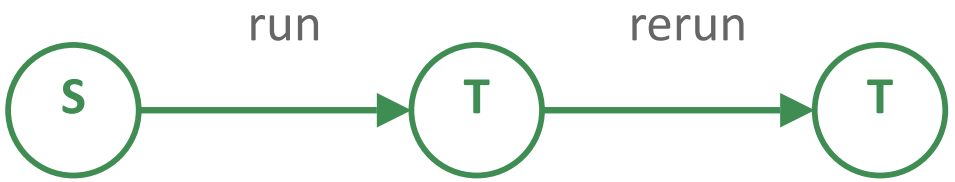
```
def update(corpus):
    add, gone, chg = diff(now)
    delete(gone + chg)
    insert(add + chg)
# rerun: diff empty, a no-op
```

APPEND: REPLAY CHANGES STATE AGAIN



f(f(N)) != f(N)

RECONCILE: REPLAY REACHES THE SAME TARGET



f(f(S)) = f(S)

Key Takeaways

1. One logical update is many physical writes, so a crash exposes the states in between. Design the on-disk format for the crash, not for the happy path; the crash is the common case.

2. Buy atomicity with one commit point: write a complete copy and publish it by atomic rename, or log the intent plus a commit record and apply afterward. How SQLite's log does this is the Sep 24 lecture.

3. Database recovery restores the database's invariants after a crash; your index's invariants are yours. Recovery is running the update again, so rerunning it must change nothing once it has succeeded.

Design the update for the kill

10 min

- Work in pairs. Copy the four colored boxes and write your answers on paper.
- Put each event in every colored box whose mechanism handles it. Some need two.
- Pick one design column: name its worst crash instant and one cost. Swap with another pair.

1 Events

Add 900 messages to a 2 GB index while searches run.

A. Power cut after close()
B. Killed halfway through
C. Search runs mid-update
D. Two updates at once
E. Rerun after a crash
F. Power cut after rename
G. Killed mid-apply of log
H. Leftover idx.db.tmp

One commit point
Event letters: _____

fsync to storage
Event letters: _____

Idempotent rerun
Event letters: _____

Serialized writers
Event letters: _____

2 Designs

Choose one whole column and justify it.

Swap a copy	Log, then apply
Copy idx.db to tmp, apply changes there	Append each change to idx.journal
fsync tmp, rename over idx.db, fsync dir	Append COMMIT, fsync journal
Recover: delete tmp, run update again	Recover: replay log if it ends in COMMIT

Your answer

Design: ___; worst instant: ___; cost: ___