Referential Integrity

Every pointer must point to something that exists

What is referential integrity?

A system has referential integrity when every reference points to something that actually exists.

- A note references image files — those files must exist
- A foreign key references a row — that row must exist
- A symlink references a path — that path must exist

Dangling reference
Target was deleted while something still points to it

Premature reference
Pointer was created before the target exists

Two rules: (1) delete data only after removing all pointers to it,
(2) create data before creating pointers to it.

Notes depend on images

Both Notebook (SQLite) and FireNote (Firestore) store notes that reference image files.
A broken reference = broken UI.

Notebook (SQLite)

```
data class Note(val id: Long,
    val text: String,
    val timestamp: String,
    val imageList: List<Image>)

// Images live in a separate table
// linked by foreign key
```

FireNote (Firestore)

```
data class Note(
    var text: String = "",
    var pictureUUIDs: List<String>
        = listOf(),
    @ServerTimestamp
    val timeStamp: Timestamp? = null,
    @DocumentId
    var firestoreID: String = ""
)
```

In both cases, the note contains references to images.
If an image doesn't exist, the app crashes or shows a broken view.

The delete problem — SQLite

Delete a note → its image rows must also be deleted, or they become dangling references.

Two layers of cleanup: CASCADE for DB rows, delFile() for disk

```
// Image.kt — foreign key with cascade
"FOREIGN KEY(note_table_id) REFERENCES notes(id) ON DELETE CASCADE"

// DatabaseHelper.kt — delete a note
fun deleteNote(note: Note) {
    for (img in getImage(db, note.id)) { delFile(img) }  // disk files
    db.delete(Note.TABLE_NAME, ...)  // CASCADE deletes image rows
}
```

- CASCADE automatically deletes image rows when the parent note is deleted
- But the physical .jpg files on disk are not in the database — manual delFile()

SQLite disables foreign keys by default! Add PRAGMA foreign_keys=ON in onOpen.

The delete problem — Firestore

Firestore has no foreign keys, no CASCADE. Your application code must handle everything.

MainViewModel.kt — manual cleanup required

```
fun removeNoteAt(position: Int) {
    val note = getNote(position)
    note.pictureUUIDs.forEach { storage.deleteImage(it) }
    dbHelp.removeNote(note) { _notesList.value = it }
}
```

SQLite: CASCADE
DB enforces integrity automatically

Firestore: manual loop
Crash mid-loop = orphaned images

NoSQL tradeoff: flexibility and scalability,
but every integrity invariant becomes application code.

The create problem — Firestore

Deleting isn’t the only danger. Creating a note can also violate referential integrity.

The naive approach — broken!

```
storage.upload(imageFile)        // async; takes
time
val note = Note(text = "...",
    pictureUUIDs = listOf("ffjsjsjsj-sjfjfj"))
db.collection("allNotes").add(note)  // writes
immediately
```

What becomes visible

```
note.pictureUUIDs = [ffjsjsjsj-sjfjfj]
storage/images/ffjsjsjsj-sjfjfj does not yet
exist!
Violates referential integrity!
```

- Note write completes before image upload finishes
- LiveData fires — every client tries to display the image
- Result: broken image in the UI, crash or error in the log

The fix: write data before pointers

Wait for the image upload to succeed, then write the note that references it.

Before (broken)

```
storage.upload(imageFile)
// Upload still in progress...
val note = Note(
    text = "...",
    pictureUUIDs = listOf(uuid)
)
db.collection("allNotes").add(note)
// note is visible, image is not!
```

After (correct)

```
storage.upload(imageFile)
    .addOnSuccessListener {
        deleteLocalFile()
        val note = Note(
            text = "...",
            pictureUUIDs = listOf(uuid)
        )
        db.collection("allNotes").add(note)
        // image exists!
    }
```

User-visible delay: the picture won’t appear until upload completes.
We accept this delay — a broken note visible to all users is worse.

The same principle in the filesystem

Hard links, symbolic links, and dangling pointers

Hard links vs symbolic links

Two ways to give a file multiple names — with very different integrity properties.

Hard link

- Another directory entry for the same inode
- Both names are equally real
- Data survives until the last link is removed
- Cannot cross filesystem boundaries

Symbolic link (symlink)

- A small file whose content is a path
- Indirection: name → path → inode
- Target deleted means symlink breaks
- Can cross filesystems and link to directories

Hard link = shared ownership (reference counting). Symbolic link = a pointer that can dangle.

Hard links in action

Same inode, multiple names, and the OS keeps a link count.

Git subcommands are hard links — same inode, same binary

```
$ ls -li /usr/bin/git /usr/bin/git-receive-pack /usr/bin/git-upload-pack
11529215003125709l5 -rwxr-xr-x 78 root wheel 118928 /usr/bin/git
11529215003125709l5 -rwxr-xr-x 78 root wheel 118928 /usr/bin/git-receive-pack
11529215003125709l5 -rwxr-xr-x 78 root wheel 118928 /usr/bin/git-upload-pack
```

- Same inode (11529215003125709l5) — all three names are the same file
- Link count is 78 — the OS tracks how many names reference this inode
- Delete any name and the data survives via the others
- Data is only freed when the link count drops to zero

Homebrew: gzip and gunzip share one binary via hard link

```
$ ls -li /opt/homebrew/Cellar/libdeflate/1.25/bin/
486994948 -r-xr-xr-x 2 witchel admin 106736 libdeflate-gunzip
486994948 -r-xr-xr-x 2 witchel admin 106736 libdeflate-gzip
```

Symbolic links dangle

A symbolic link is a pointer — and pointers can outlive their targets.

Dotfile symlinks — working, targets exist

```
$ ls -l ~/.zshrc ~/.bashrc ~/.tmux.conf
lrwxr-xr-x 1 witchel staff 30 .bashrc → /Users/witchel/dotfiles/.bashrc
lrwxr-xr-x 1 witchel staff 34 .tmux.conf → /Users/witchel/dotfiles/.tmux.conf
lrwxr-xr-x 1 witchel staff 30 .zshrc → /Users/witchel/dotfiles/.zshrc
```

A real dangling symlink — target was deleted, pointer remains

```
$ ls -l ~/.claude/debug/latest
lrwxrwxr-x 1 witchel staff 69 latest → .../e71c5e64-...7e80.txt
$ cat ~/.claude/debug/latest
cat: .../latest: No such file or directory
```

A dangling symlink is a referential integrity violation —
the pointer outlives its target, just like Firestore.

Key takeaways

Referential integrity is universal
Databases, filesystems, cloud storage — any system with pointers can have dangling references

Two rules
Delete data only after removing all pointers. Create data before creating pointers to it

Correctness costs latency
Waiting for an upload before writing the note causes a user-visible delay — and that is the right tradeoff

Hard link
can’t dangle (ref counting)

CASCADE
DB enforces (declarative)

Symlink / Firestore
can dangle (app must enforce)