READ MODE · ANSWER FIRST

Iceberg Snapshots, Branches, and Time Travel

An interactive, visual explanation of Iceberg Snapshots, Branches, and Time Travel, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

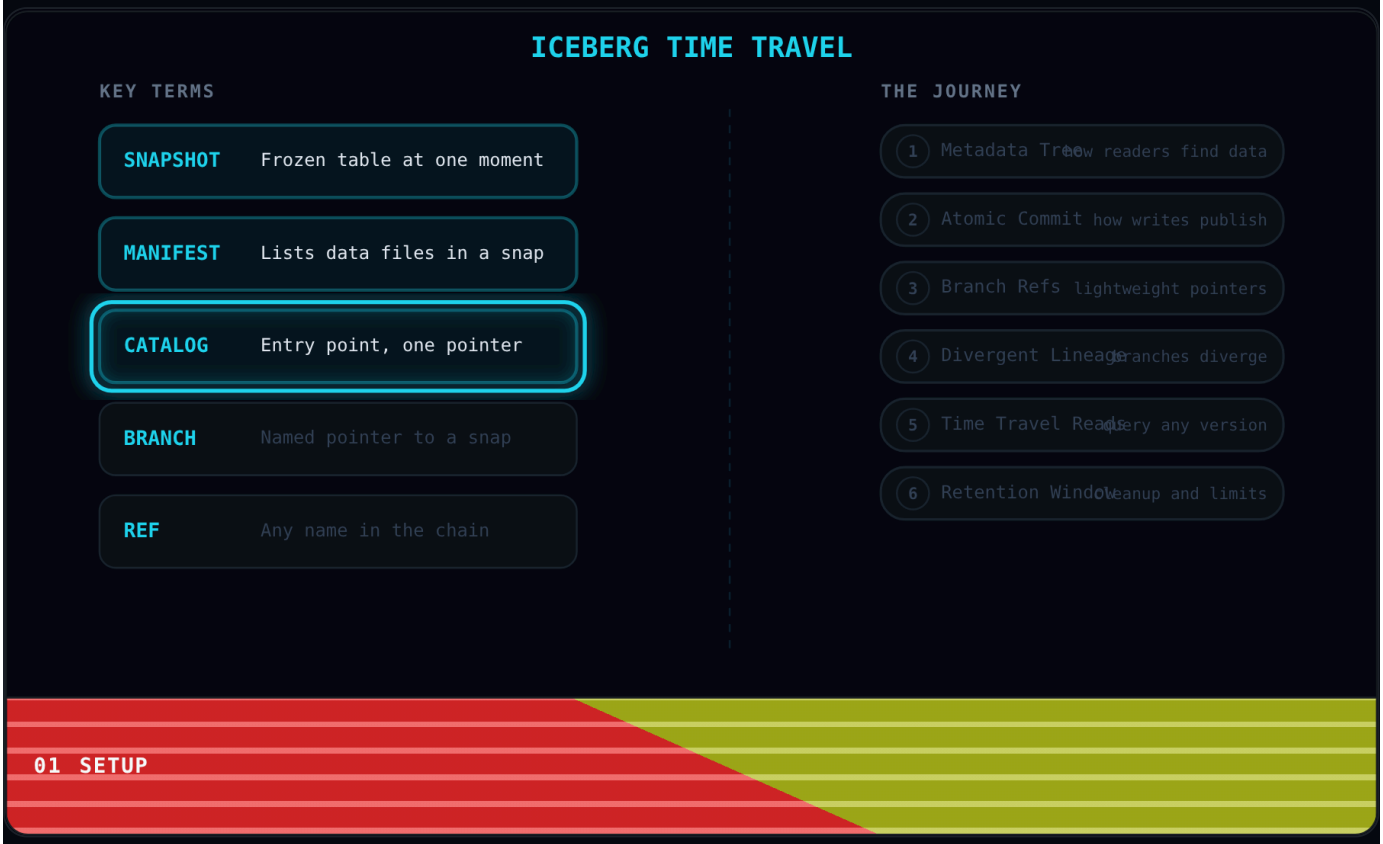
The whole story in 6 lines

Iceberg Snapshots, Branches, and Time Travel becomes easier to reason about when every stage is connected as one system.

1. Readers never discover data files directly. They resolve a ref, load the current metadata file, then walk snapshots and manifests.
2. Iceberg commits build candidate metadata off to the side. The live table changes only when the final pointer swap succeeds.
3. Creating a branch copies no data. It only stores another name that points at a chosen snapshot in the lineage.
4. Main and dev can expose different table views while still referencing much of the same underlying storage.
5. The read path always locks onto a concrete snapshot first. Only then does it plan manifests and files, which is why time travel stays...
6. Time travel survives only as long as snapshots stay reachable. Retention is metadata pruning first, physical file deletion second.

STAGE 1 OF 8

Setup



Welcome to Iceberg time travel. Think of a table that remembers every version of itself, like a document with infinite undo history. That is what Iceberg gives you.

A snapshot is a frozen picture of the entire table at one moment in time. Every write creates a new snapshot without touching the old one.

A manifest is a file that lists which data files belong to a snapshot. It is like a table of contents for one chapter of the table.

The catalog is the single entry point. It stores one pointer to the current metadata file for each table name.

A branch is a named pointer to a snapshot. Main always points at the latest production state. Dev branches let you experiment without affecting main.

Now let us start with the most fundamental idea: how the metadata tree connects all these pieces together.

STAGE 2 OF 8

Metadata Tree

02 METADATA TREE

Here is the core question this stage answers: how does a reader find the actual data files for a table? The answer starts at the catalog, which holds just one pointer to the current metadata file.

That metadata file carries the table schema plus the full snapshot log. Think of it as the spine of a book that connects every chapter.

The active snapshot is the frozen picture we learned about in the Setup. It points at the table state at one exact moment.

The snapshot opens its manifest list, which fans out into individual manifests. Try changing the Manifests control in the sidebar to see how the fanout grows.

Each manifest lists concrete data files on storage. Switch the Partitioning control to see how daily, region, or bucket groupings change the file names.

That is the metadata tree in one picture: catalog to metadata to snapshot to manifests to files. No file is ever mutated in place. Next, we will see how a new write publishes itself through this chain.

STAGE 3 OF 8

Atomic Commit

Now that we understand the metadata tree, let us see how a new write enters it. The writer reads the current snapshot and plans changes against it.

New data files are staged on storage without touching the live head. Think of this as writing a draft next to the published book. Switch the Operation control to see how append, rewrite, and delete each look different.

The writer builds a candidate snapshot that references the new files plus any unchanged manifests from the parent. This is the new chapter being assembled.

A new metadata file is produced with this candidate snapshot as the current version. Notice how the parent link connects back to the old head.

Now comes the critical moment. The catalog attempts one atomic pointer swap, a compare-and-swap. If no other writer changed the pointer first, this commit wins.

Switch the Outcome control to Conflict to see what happens when two writers race. The loser finds a stale base and must retry from the new head. That retry loop is the price of lock-free commits. Next, let us see how branches give different writers their own ref to advance.

STAGE 4 OF 8

Branch Refs

04 BRANCH REFS

We just saw how a commit uses the atomic swap from Stage 2. But what if you want to experiment without affecting the production table? That is what branches are for.

Main is just a named ref that points at the latest snapshot. Remember the catalog pointer from Stage 1? Branches work the same way, one level down.

Creating a dev branch is instant. No data is copied. The new ref just points at a chosen snapshot. Try switching the Base control to History to see the branch start from an older point.

Tags work like branches that never move. Toggle the Tag control in the sidebar to see an audit tag pinned to a specific snapshot. Tags are useful for compliance checkpoints.

Notice the shared storage panel below. Both refs reach the same underlying manifests and data files. Creating refs is metadata only.

When a new commit lands on dev, only the dev ref moves. Main stays put. That independence is what we will explore next: how two branches can diverge while still sharing files.

STAGE 5 OF 8

Divergent Lineage

05 DIVERGENT LINEAGE

In Stage 3 we created branches. Now watch what happens when both branches start writing. Main and dev begin from the same base snapshot.

Main lands new commits and advances its ref. Each commit uses the atomic swap we learned about in Stage 2, but only main's pointer moves.

Dev writes its own changes independently. Switch the Experiment control to see how append, rewrite, or delete operations create different file patterns on the dev side.

Toggle the Reuse control in the sidebar. Notice the shared files panel. Files from the base snapshot are still referenced by both branches until one of them rewrites or deletes them.

Both heads now expose entirely different table states, even though they share much of the same underlying storage. This is the power of immutable metadata plus lightweight refs.

Nothing becomes visible on the other branch until a later publish or merge moves that ref. That isolation is exactly what makes Iceberg branches safe for experimentation. Next, we will see how a reader picks which view to query.

STAGE 6 OF 8

Time Travel Reads

06 TIME TRAVEL READS

We have seen how branches diverge in Stage 4. Now a reader arrives with a SQL query. How does it know which version of the table to see? That depends on the selector it provides.

Switch the Selector control between Branch, Snapshot, and Time. Each gives a different way to name the exact table state you want. All three collapse to one concrete snapshot before the scan begins.

The resolved snapshot becomes the pinned read view. Even if main advances while this query runs, the read stays locked to the snapshot chosen at the start. That is the consistency guarantee.

Iceberg loads only the manifests reachable from that snapshot. Remember the manifest list from Stage 1? The same fanout applies here, but scoped to the resolved snapshot.

Partition pruning narrows the scan to only the matching data files. Files outside the query predicate are skipped entirely.

Returned rows always match the resolved snapshot. That one-snapshot-at-a-time rule is why time travel queries are consistent. Now let us step back and see what happens when old snapshots need to be cleaned up.

STAGE 7 OF 8

Retention Window

07 RETENTION WINDOW

We have built up snapshots across all the previous stages. But snapshots accumulate forever unless something cleans them up. That is what the retention window does.

Toggle the Publish control to see how moving main to the dev head (a merge) changes which snapshots are reachable. Live refs define the boundary of what stays alive.

Adjust the Retain stepper to keep 2, 3, or 4 snapshots. The retention window keeps only the newest reachable snapshots under the live refs. Everything older becomes a candidate for expiry.

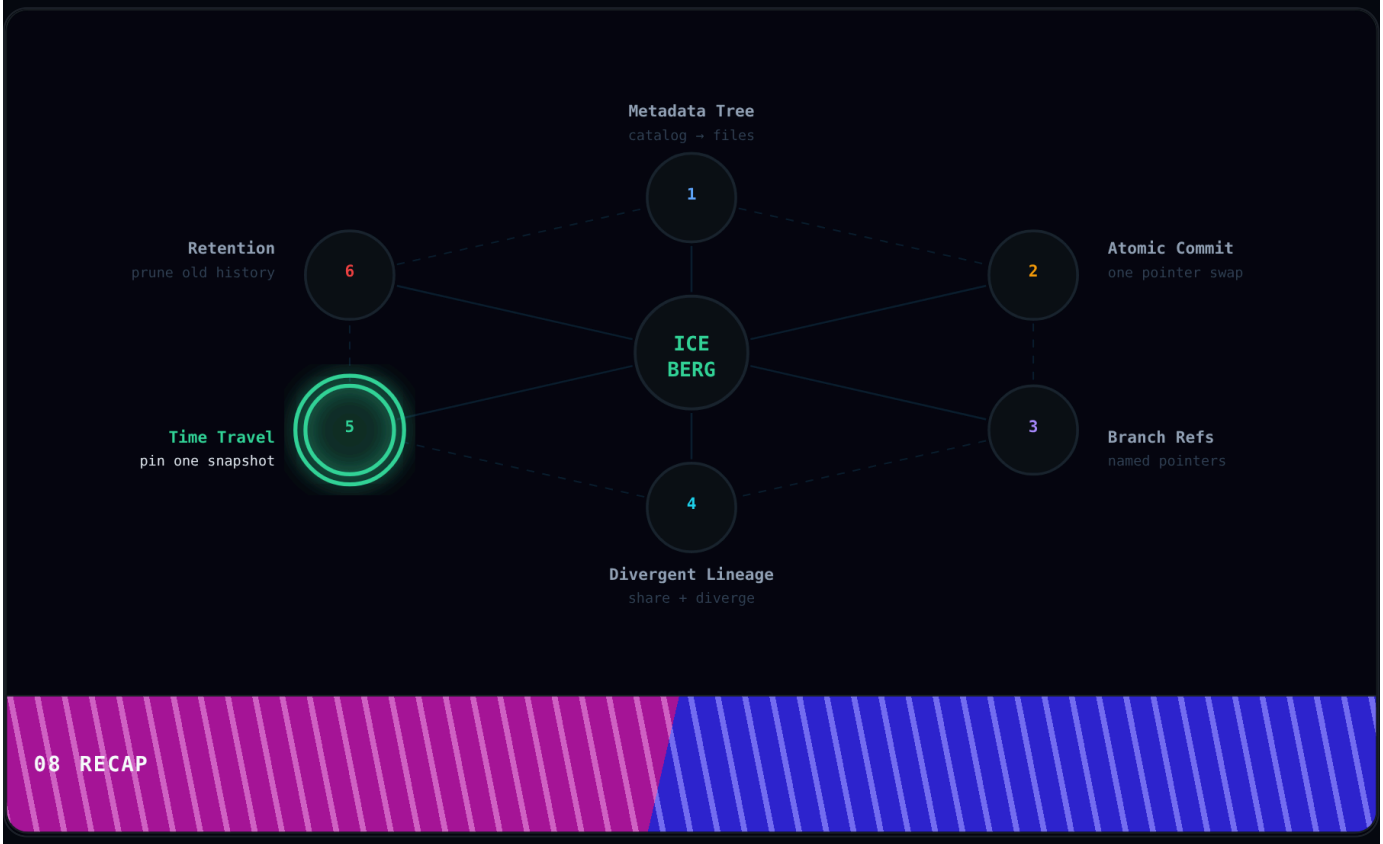
Expired snapshots and manifest lists are dropped from metadata first. This is the same metadata tree from Stage 1, but now parts of it are being pruned instead of grown.

Only after metadata releases them do orphan data files become safe to delete. Toggle the Orphans control to see the difference. This two-phase cleanup prevents accidental data loss.

The table stays compact, but time travel now exists only inside the retained window. That tradeoff between history depth and storage cost is the final piece of the Iceberg puzzle. Now let us step back and see the whole picture together.

STAGE 8 OF 8

Recap



08 RECAP

We started with the metadata tree: catalog to metadata to snapshot to manifests to files. That chain is the foundation everything else builds on.

Then we learned that writes never touch the live head. They build a candidate off to the side and publish with one atomic swap. That is how Iceberg stays consistent without locks.

Branches are just named pointers into the snapshot chain. Creating one copies no data. It is metadata only, instant and free.

When branches advance independently, they can diverge while still sharing files from their common ancestry. That reuse is what makes branches practical at scale.

Time travel reads pin to one snapshot before scanning. Branch names, snapshot ids, and timestamps all collapse to one frozen view. That is the consistency guarantee.

Retention cleans up old snapshots in two phases: metadata first, then orphan files. It keeps the table compact while preserving a configurable window of history.

All six concepts connect into one system. Immutable metadata enables atomic commits. Atomic commits enable branches. Branches enable divergent lineage. Time travel reads the result. Retention keeps it sustainable. That is Iceberg.