READ MODE · ANSWER FIRST

How Apache Pinot full upserts work internally

Explore Pinot full upsert internals: comparison columns, validDocIds, deletes, compaction, snapshots, TTL, and consistency modes.

CHEAT SHEET · 25 ESSENTIAL IDEAS

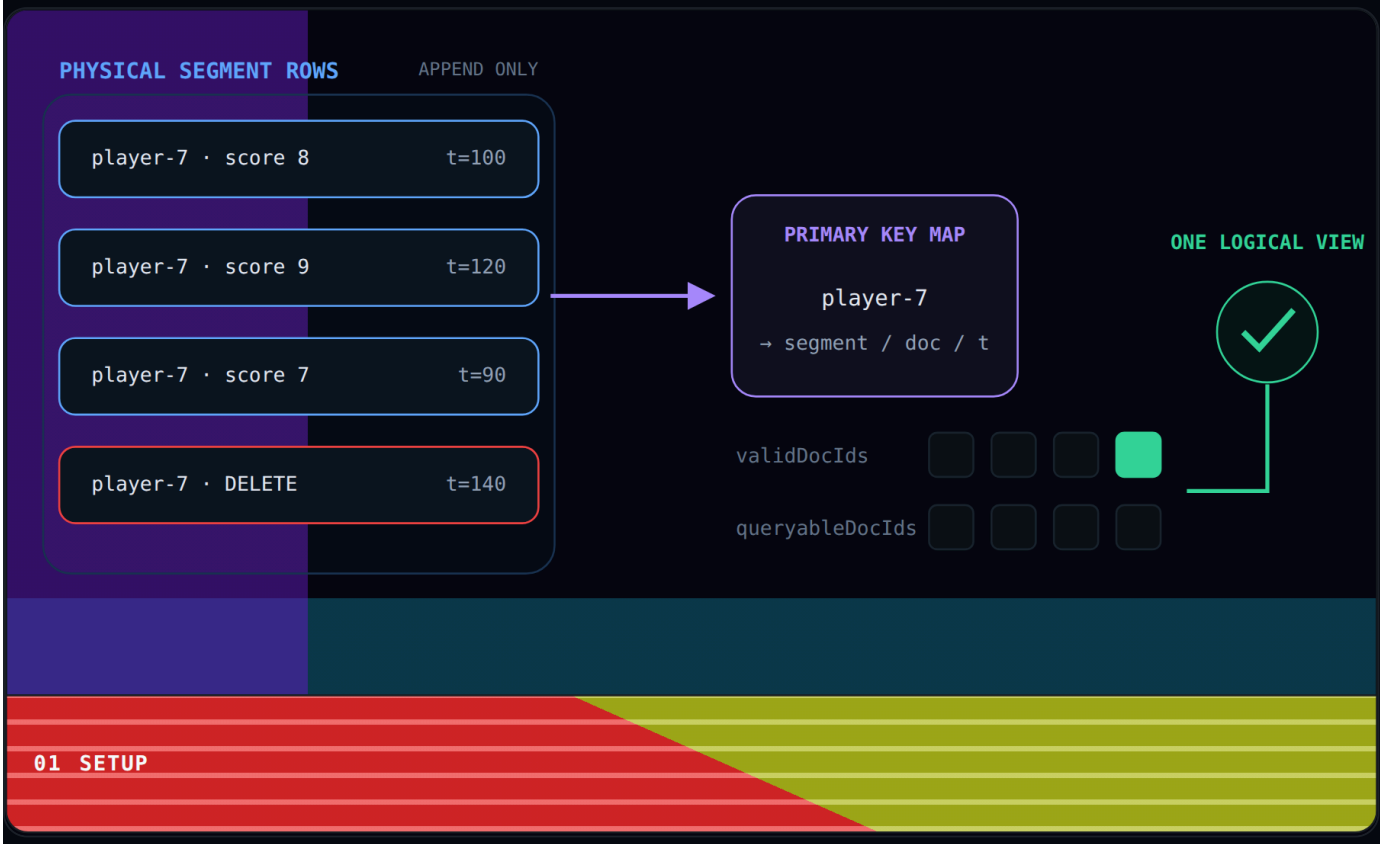
The whole story in 25 lines

Pinot full upsert is a partition-local winner map whose bitmaps define query truth while maintenance features preserve that truth over time.

1. Pinot appends every accepted version physically while bitmaps expose one logical winner per primary key.
2. Every key must stay in one partition because each partition owns an independent upsert metadata map.
3. RecordInfo carries the primary key, doc id, comparison value, and delete flag into the winner decision.
4. The concurrent map points each hashed primary key at one segment, doc id, and comparison value.
5. A greater comparison value wins, a smaller value is stale, and an equal value needs a deterministic tie rule.
6. Within one segment, a newer row appends while the valid bit moves from the old doc id to the new one.
7. Across segments, Pinot removes the old valid bit and adds the new valid bit as one metadata handoff.
8. Queries intersect segment rows with validDocIds so obsolete physical versions never enter normal results.
9. A tombstone can remain the valid winner while queryableDocIds hides it from ordinary queries.
10. A stale row is normally stored physically but receives no valid bit, so it cannot replace the winner.
11. NONE mode can keep, drop, or mark stale rows, but drop and mark are unavailable with consistent views.
12. Loading sealed segments scans RecordInfo values and reconstructs the same partition map and bitmaps.
13. Commit replacement remaps mutable doc ids to immutable doc ids without changing the logical winner.
14. Equal comparisons use LLC sequence, authoritative segment time, and uploaded naming for deterministic placement.
15. Commit-time and minion compaction remove invalid physical versions while preserving the same logical answer.
16. Snapshots persist valid and queryable bitmaps before persisting the TTL watermark that depends on them.
17. Preload rebuilds metadata from snapshot-selected rows instead of comparing every physical row at startup.
18. metadataTTL removes old primary-key locations below largestSeen minus TTL and requires one comparison column.
19. deletedKeysTTL keeps tombstones long enough to block older values before their metadata can be forgotten.
20. Consistent delete cleanup waits until a deleted key occurs in no more than one remaining segment.
21. NONE consistency maximizes write freedom but a query can observe bitmap changes from different moments.
22. SYNC holds a writer lock across old-bit removal and new-bit addition while queries take the reader lock.
23. SNAPSHOT copies changed bitmaps and atomically swaps a query view, trading bounded freshness for lock-free reads.
24. Servers temporarily add newly created segments as optional query inputs while broker routing catches up.
25. PROTECTED force-commit handling remembers previous locations and restores keys missing from the committed copy.

STAGE 1 OF 27

Setup



Welcome to our tour of Pinot full upsert. Let’s imagine a live scoreboard receiving repeated updates for one player. Pinot stores every accepted event in append-oriented segments, but viewers should see only the newest score as if the older row had been overwritten.

Before we trace that trick, let’s learn four labels on the screen. A primary key names the player, a comparison value orders versions, a doc id locates a row, and the two bitmaps decide authority and query visibility.

Choosing one winner is only the beginning. That answer must survive deletes, late events, segment replacement, compaction, restarts, and queries arriving during updates. How can all these mechanisms protect the same truth?

PAUSE AND PREDICT

Which layer can preserve one answer while segment rows remain append-only?

- **Winner metadata and bitmaps**
- **In-place row rewrites**
- **Broker result caching**

Skip this check

In this course, we’ll follow one primary key into its winner map, through bitmap visibility and physical cleanup, then into recovery, retention, and consistent query views. Let’s begin with the central trick: store every version physically while exposing one version logically.

STAGE 2 OF 27

Append Storage, Overwrite View

Let’s start with the puzzle that makes Pinot upsert interesting. Pinot writes each accepted update as another physical row because its segments are append oriented, yet users expect one current value.

An ordinary database might overwrite the old row in place, but an immutable Pinot segment cannot do that. How can a later event still behave like an overwrite?

PAUSE AND PREDICT

How can a later event behave like an overwrite inside immutable segments?

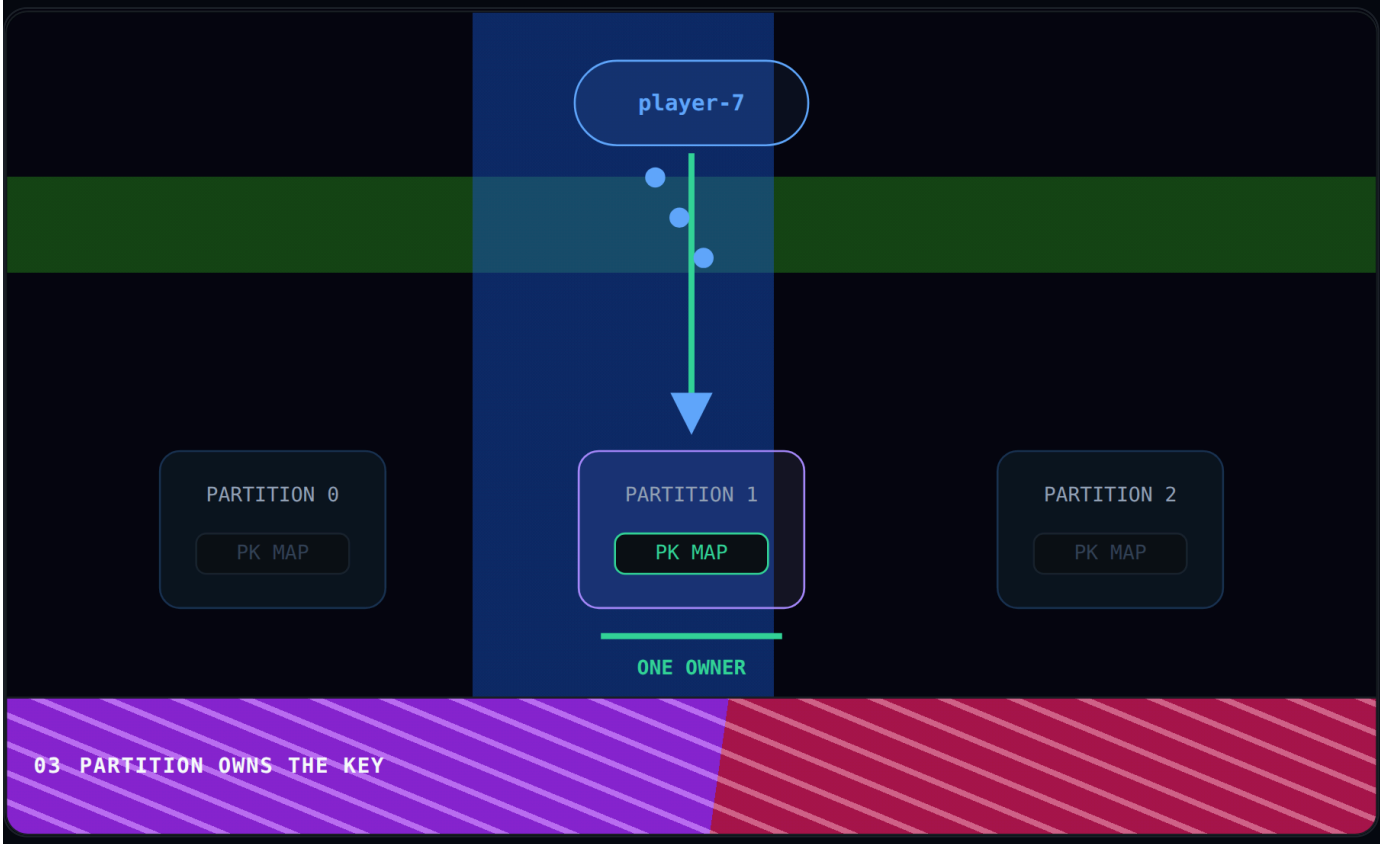
- **Rewrite the old row**
- **Move the winner metadata**
- **Return every version**

Skip this check

Each accepted version extends the physical history, while the winner pointer moves to the greatest comparison value. validDocIds then admits only that row into normal queries. The row bytes never move, so the query view changes without mutating a sealed file.

Here’s the key idea: Pinot keeps the physical history and changes only the logical view through metadata. That design keeps segment writes sequential and pushes overwrite semantics into compact selection data. Keep that split in mind because next we need to decide which manager is allowed to choose the winner.

Partition Owns the Key



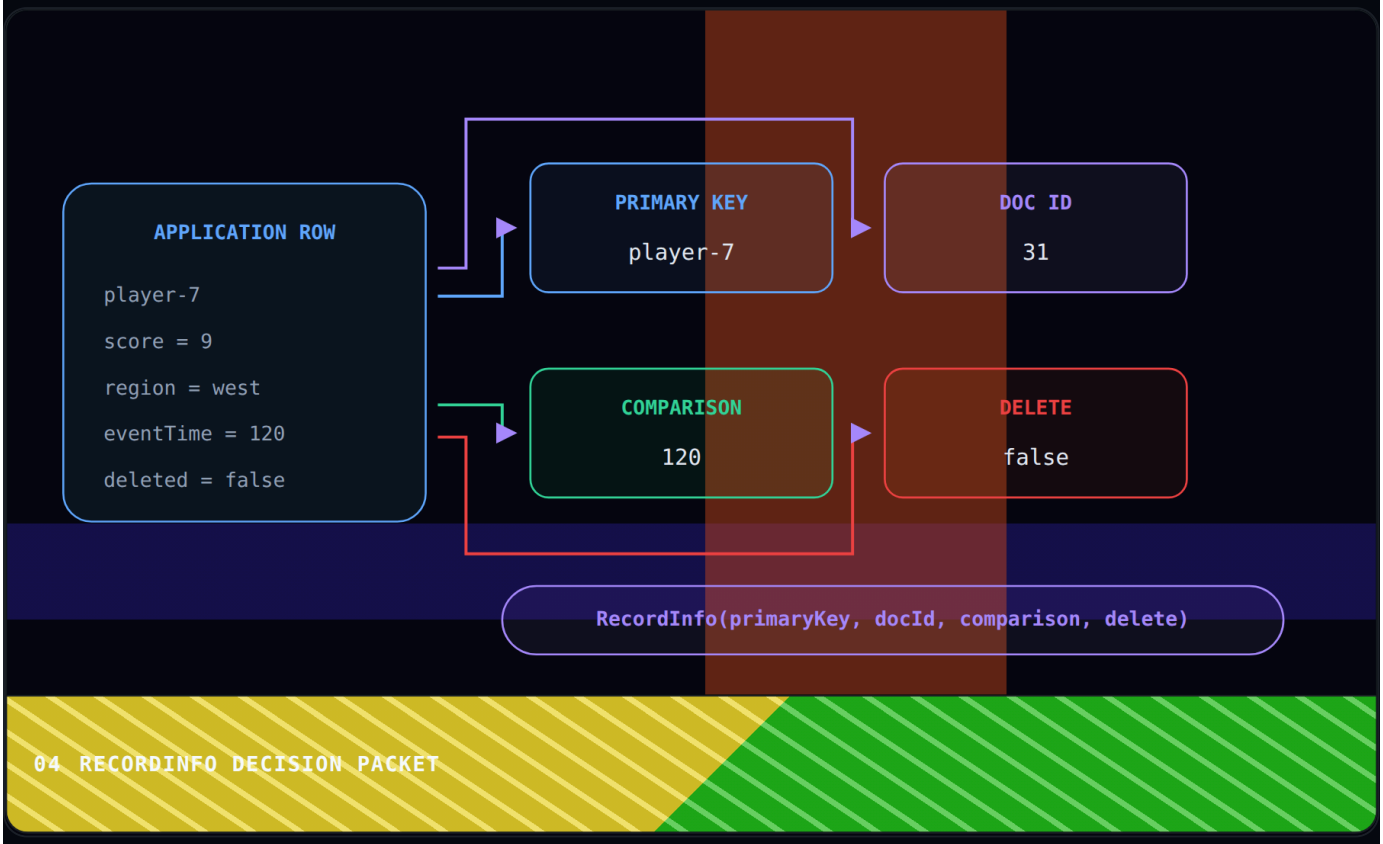
Now let’s ask who gets to move that winner pointer. Every version of one primary key must reach the same partition metadata manager because each manager owns a separate map and does not negotiate with the others.

Imagine version 10 landing in partition zero while version 12 lands in partition one. Both independent maps could correctly declare a local winner and still violate one-key-one-winner correctness.

The producer routes every version by the same primary key, so the record always lands in one ownership lane. A split route creates two owners before Pinot has any way to repair the mistake.

So partitioning is not just a speed trick here. It is part of correctness because it gives each key one decision maker. Pinot validates this assumption instead of coordinating winner maps across partitions. One manager can now make every later decision locally. Next we’ll open the small packet that carries everything this manager needs.

RecordInfo Decision Packet



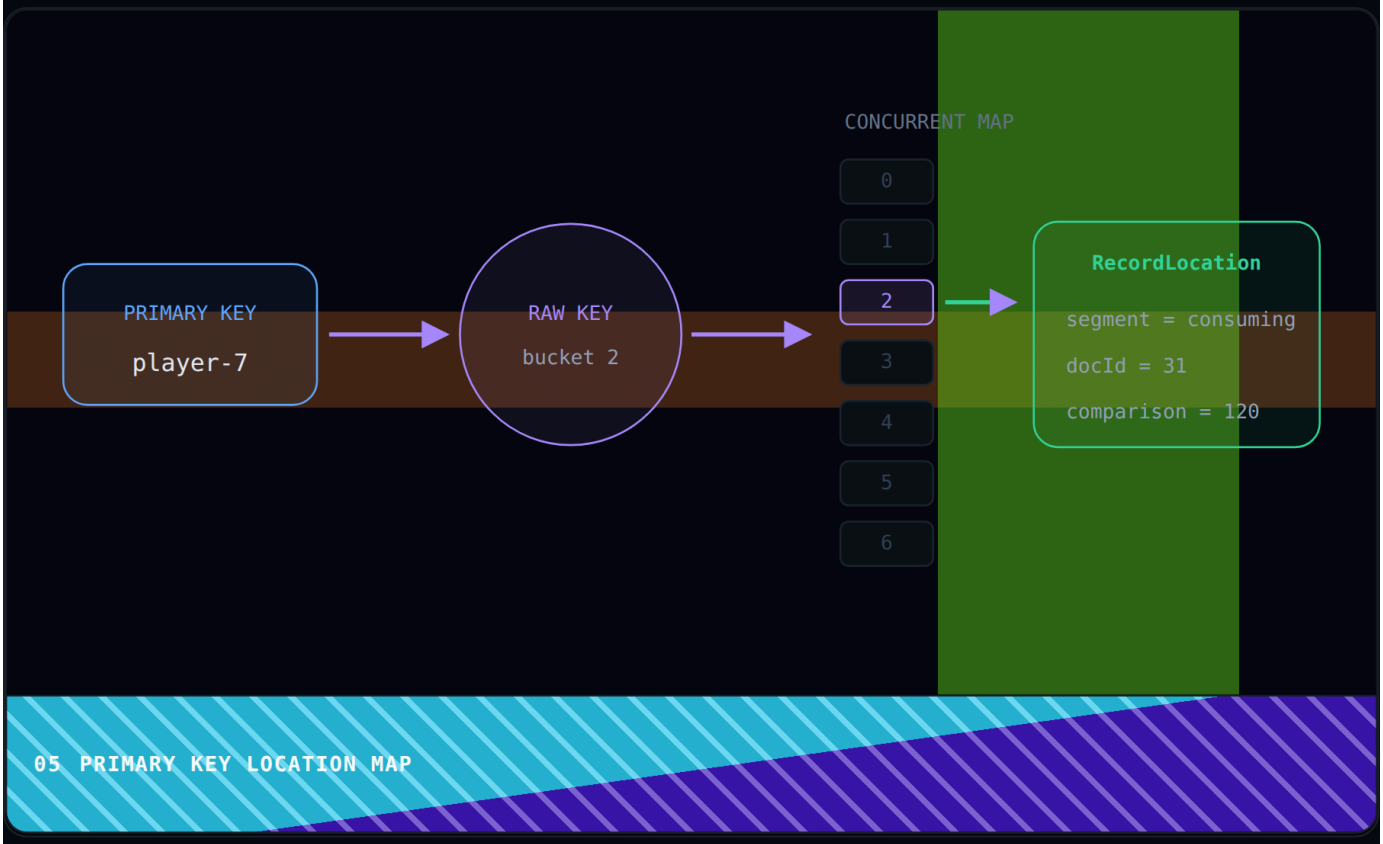
We now have one manager for the key, so let’s unpack what that manager actually receives. MutableSegmentImpl turns the large application row into a smaller RecordInfo packet containing only the facts needed for the upsert decision.

The application row may contain dozens of columns, but only four facts determine ownership and visibility. Those facts must survive when the row reaches the metadata manager.

RecordInfo carries the primary key, physical doc id, comparison value, and delete flag. Those four facts identify the entity, locate this version, order it against the current winner, and tell queries whether the winner is a tombstone.

Think of RecordInfo as the label on a package: it carries enough information to route and judge the row without copying the contents. The same packet shape supports live ingestion and sealed-segment replay. That compact boundary keeps ingestion and metadata concerns clean. Next the manager uses its primary key to find one entry in the winner map.

Primary Key Location Map



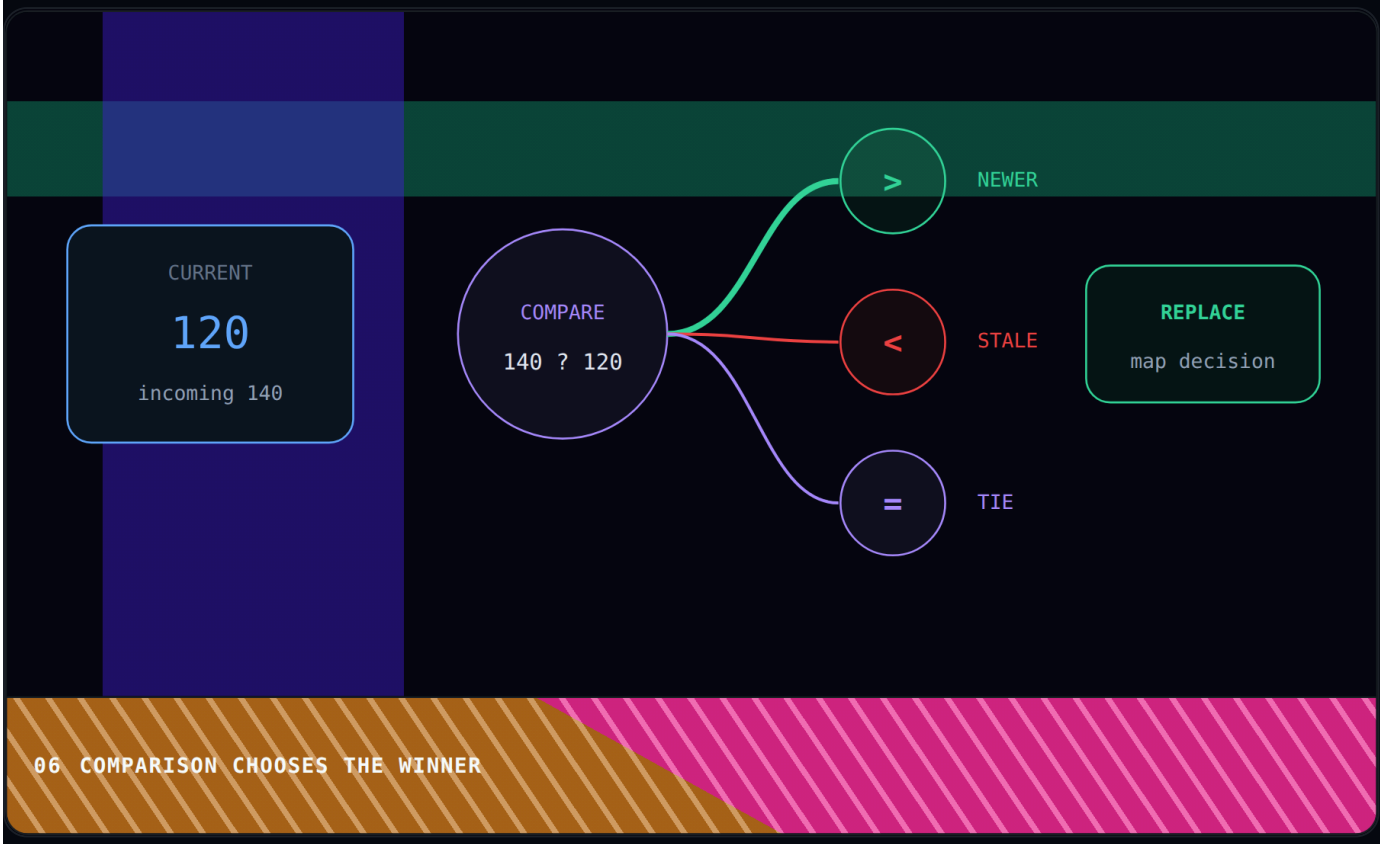
Let’s follow that RecordInfo packet into the map. Pinot can use the original primary key or turn it into a fixed-size MURMUR3 hash, but either form leads to one concurrent map entry.

The physical row already lives inside a segment, so copying it into the upsert map would waste memory. A compact locator is enough to identify and compare the winner.

The map key can be the primary key or its fixed-size hash, while the stored value stays a compact record location. That location carries the segment, doc id, and comparison value needed next.

Here’s the useful mental model: the map is an index card pointing to the winning row, not a second copy of the row itself. That separation keeps metadata proportional to keys rather than full row width. The segment remains the only owner of full row data. Next we’ll see when the comparison gate replaces that card.

Comparison Chooses the Winner



Now we reach the actual winner decision. Pinot compares the incoming RecordInfo with the current map entry inside one atomic operation, so the new arrival either becomes logical truth or stays only as physical history.

A greater value is newer and a smaller value is stale, but equality still needs a deliberate rule. The three branches therefore cannot all move the pointer in the same way.

Greater values replace the locator, smaller values divert to stale handling, and equal values accept the later physical record. This rule lets loading and replacement progress deterministically.

Switch the Arrival control through Newer, Stale, and Tie. Compare which map location survives, because the same gate can replace, reject, or deterministically advance the winner.

The important takeaway is that comparison creates logical order without rewriting stored rows. Next we'll follow an accepted update inside one segment and move the valid bit from the older doc id to the newer one.

STAGE 7 OF 27

Replace Inside One Segment

Let’s keep both versions inside one consuming segment for a moment. Their doc ids are simply row positions, so the newer version is appended at the tail instead of replacing the older bytes.

Appending the row preserves the segment write path, but a query must never return both versions. One bitmap can make the later doc id replace the earlier one logically.

The new row appears at the tail while validDocIds clears the old cell and sets the new cell. The physical strip keeps both rows, but the single green bit follows the map pointer to the accepted winner.

So an update inside one segment is really a row append plus a local bitmap move. No file rewrite is needed because query authority follows the bit. That local move is the simplest form of winner transfer. Next we’ll keep the same one-winner rule while handing that bit across two different segment objects.

Handoff Across Segments

Now let’s make the cross-segment handoff a little harder. The previous winner sits in an immutable segment while the new row arrives in the consuming segment, so Pinot must update metadata owned by two different segment objects.

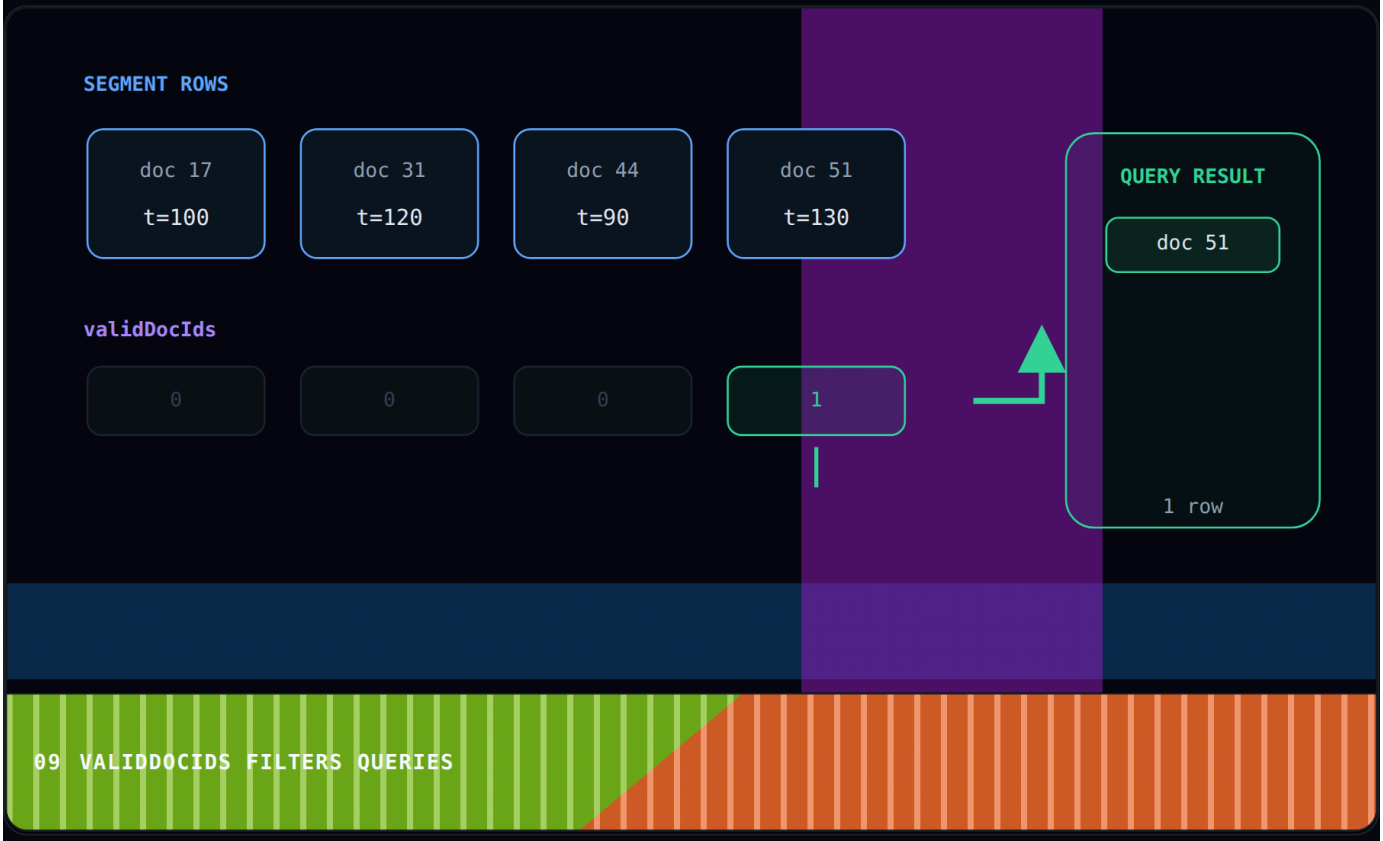
The old bitmap must lose its winner while the consuming bitmap gains one. Pinot must complete that handoff without leaving the key valid in both segments or in neither segment.

The map pointer crosses to the consuming segment as the old valid cell empties and the new cell fills. The important invariant is not where the row lives, but that exactly one segment contributes a valid version for this key.

Keep this two-bitmap handoff in mind because the consistency modes later exist to protect it from overlapping queries. The map and both segment bitmaps must agree after the move. First, let’s trace how validDocIds turns those physical rows into one normal query result.

validDocIds Filters Queries

★ If you remember one thing · Pinot keeps every physical version while validDocIds exposes exactly one logical winner.



We now have one green valid position, so let’s connect it to query execution. Each bitmap position lines up with one physical row, and Pinot intersects query candidates with validDocIds before obsolete versions can enter the result.

The old rows still exist and can be useful for diagnosis, even though ordinary users should see one current value. What changes when a query deliberately bypasses upsert filtering?

PAUSE AND PREDICT

What will an ordinary upsert query return when four physical versions exist?

- One winning version
- All four versions
- No rows until compaction

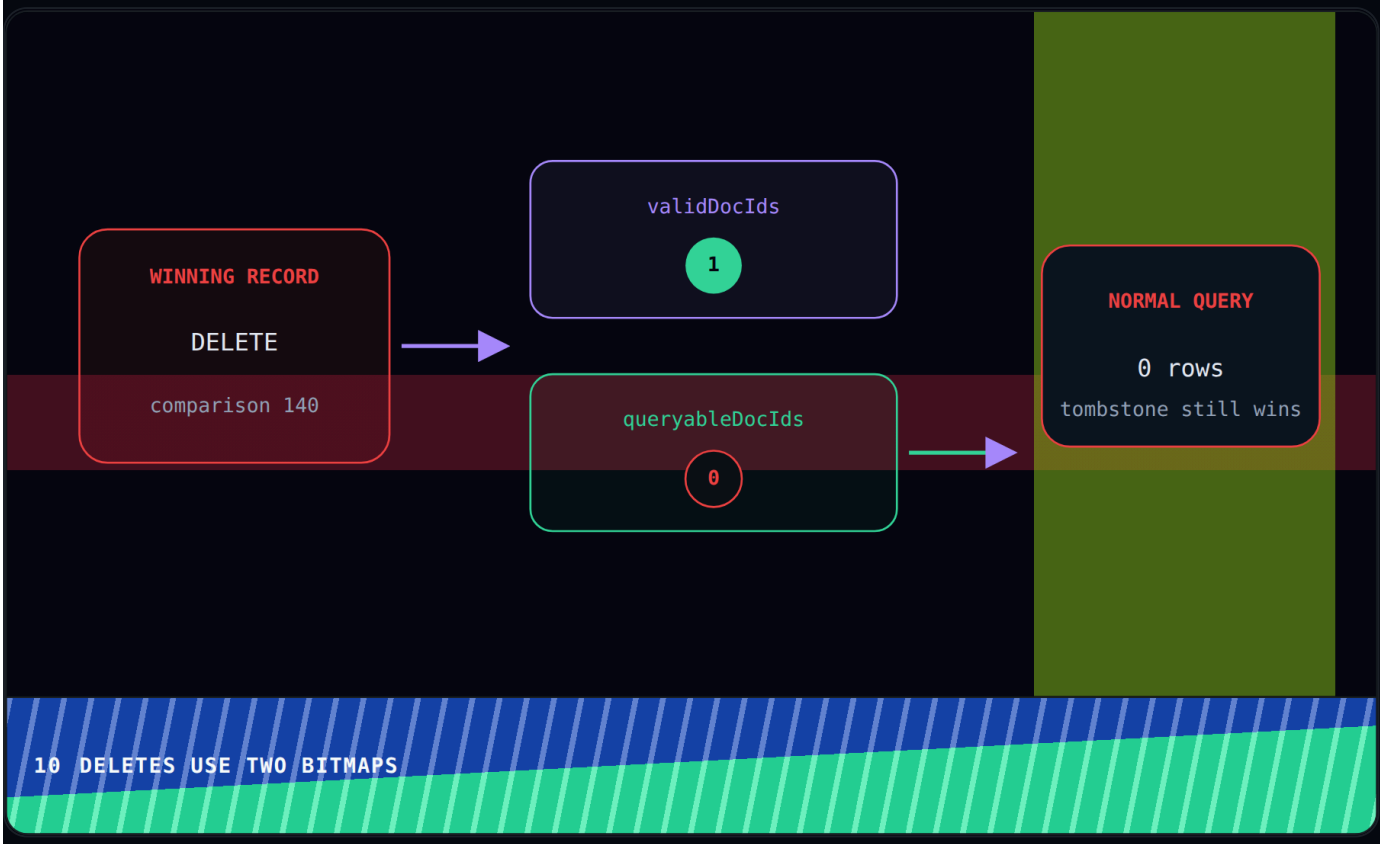
Skip this check

The ordinary Upsert path intersects row positions with validDocIds and emits one winner. skipUpsert bypasses that logical view and exposes every physical version for diagnosis.

Switch the Query view control between Upsert and skipUpsert. Compare the result card, where the logical path returns one winner while the bypass returns every stored version.

That is the bridge from append-only storage to overwrite-like query behavior: old rows remain stored, but validDocIds keeps them out of the answer. Next we need a second bitmap for a winner that represents deletion rather than a live value.

Deletes Use Two Bitmaps



Now let’s handle a delete, which is a surprisingly important edge case. The validDocIds bitmap can say the tombstone wins, but Pinot still needs a separate answer for whether that winner should appear in normal query results.

If Pinot removed a winning tombstone from every piece of metadata, an older value could look like the best remaining record. Deletion authority must therefore survive outside the normal query result.

Both live values and tombstones keep their valid bit because both remain authoritative. The tombstone alone loses its queryableDocIds bit, so ordinary queries return no row for that key.

Switch the Incoming record control between Value and Delete. Compare queryableDocIds and the result card, while validDocIds keeps either winner authoritative.

Here’s the distinction to remember: validDocIds records authority while queryableDocIds controls ordinary visibility. Next we’ll send an older value through the same comparison gate and find out what Pinot stores when that row loses.

Stale Arrival by Default

Let’s say an event arrives late to our server. It is physically new because Pinot just received it, but its smaller comparison value makes it logically older than the winner already stored in the map.

The comparison gate rejects that event as a winner, but ingestion has already created a row position for it. Stale metadata does not necessarily require the mutable segment to discard the physical row.

By default, the stale row remains in the physical strip but receives no valid bit. The current winner, its bitmap position, and the primary-key map entry all stay unchanged because the smaller comparison value cannot replace them.

So “stale” describes the row’s logical status, not whether its bytes exist. The winner comparison and physical retention policy are separate decisions. Physical retention can support diagnosis without changing query truth. Next we’ll compare Pinot’s three choices for those bytes: keep the row, drop it, or mark it for diagnosis.

Drop or Mark Stale Rows

Now let’s compare those three stale-row policies. In consistency mode NONE, Pinot can keep the physical row, drop it, or store it with an outOfOrderRecordColumn flag, while the same newer record remains the winner.

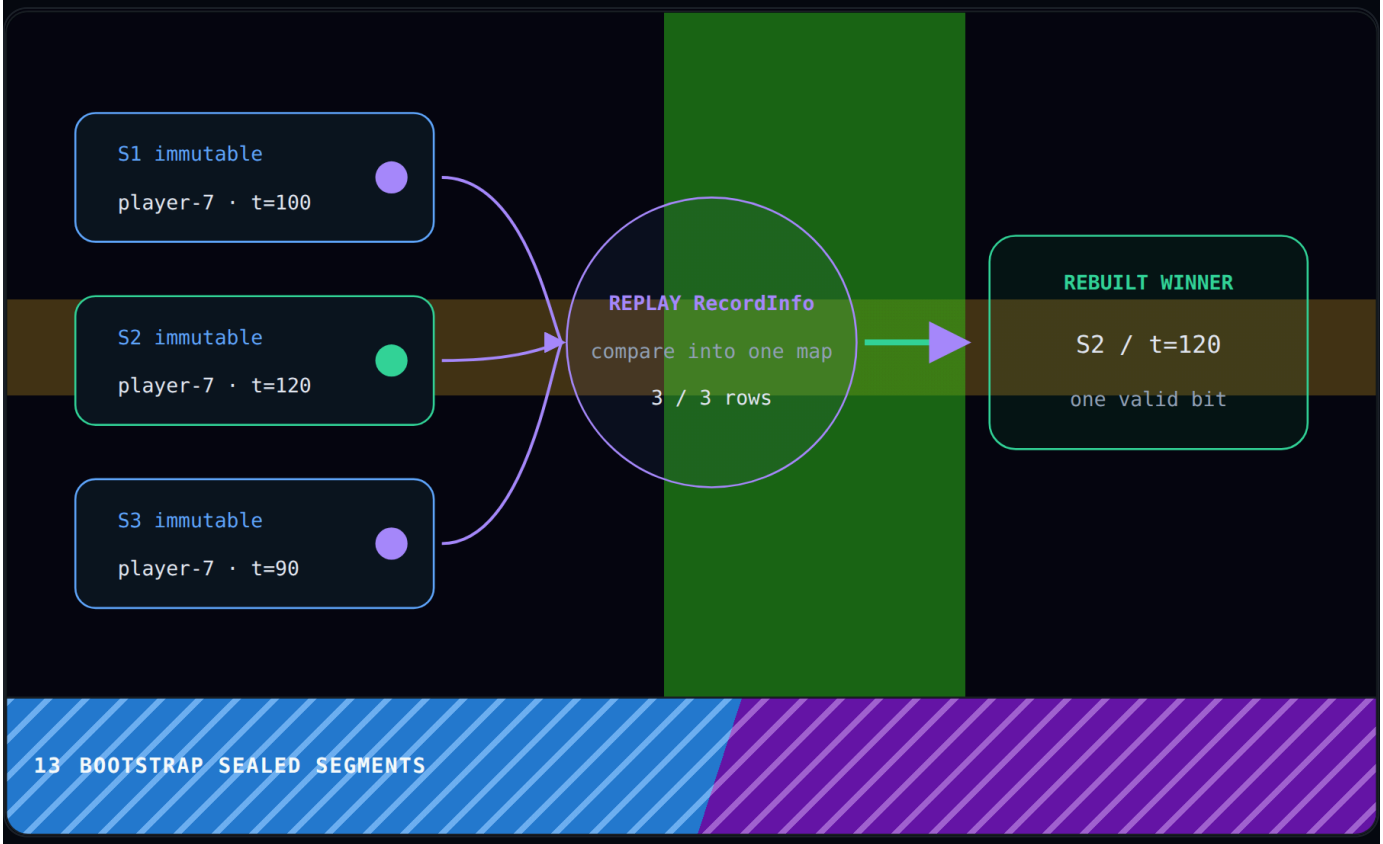
Keep, Drop, and Mark are identical at the winner map because none replaces the current record. Their physical evidence differs, ranging from no stale row to a retained diagnostic flag.

Keep stores an ordinary invalid row, Drop prevents that row from surviving, and Mark stores it with an out-of-order flag. Downstream diagnosis can therefore distinguish the marked case.

Switch the Policy control through Keep, Drop, and Mark. Compare whether the stale row remains, disappears, or carries a diagnostic flag while the logical winner stays fixed.

The catch is that Drop and Mark require metadata to be checked before Pinot finishes its other indexes. Remember that ordering constraint because SYNC and SNAPSHOT later disable these options to protect a different promise: a consistent query view.

Bootstrap Sealed Segments



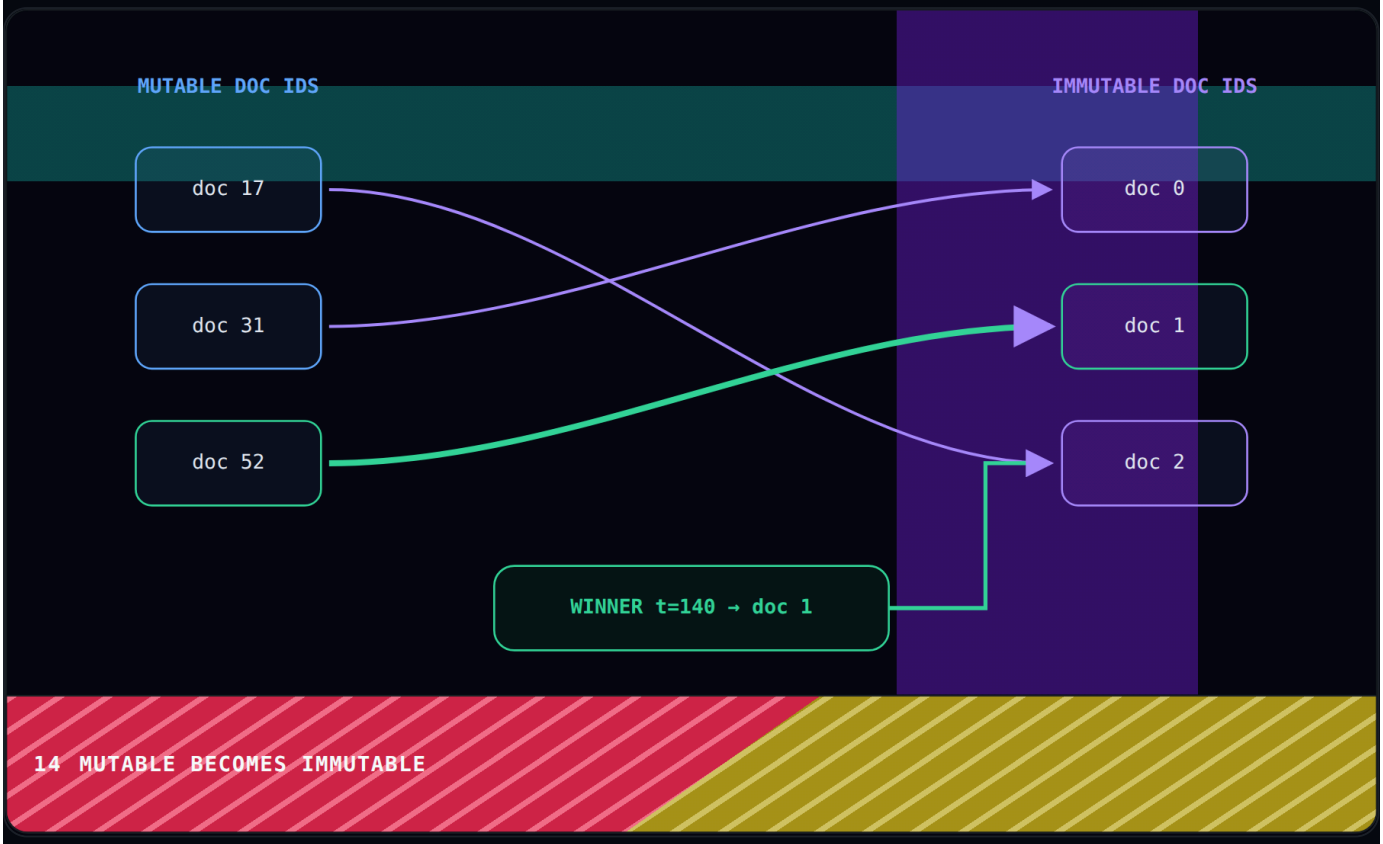
So far we have followed live rows, but a restarted server begins with segment files and no in-memory winner map. Let’s rebuild the same truth from sealed segments already stored on disk.

Several sealed segments may contain different versions of the same primary key, and their loading order is not the logical order. Replay must recover one deterministic winner from those rows.

Each segment cursor reconstructs RecordInfo values and feeds them into the shared comparison map. Every accepted winner sets its own valid bit and clears the losing location until exactly one segment owns the key.

The reassuring part is that startup replays the same RecordInfo decisions used during live ingestion, so it rebuilds the same winner rules. Segment load order cannot replace comparison order as the source of truth. Rebuilding bitmaps alongside the map restores both location and authority. Next a realtime commit replaces a mutable segment with an immutable copy whose doc ids may change.

Mutable Becomes Immutable



Now picture a realtime segment finishing its commit. Pinot replaces the mutable version with an immutable copy, but sorting or compaction may give the same logical rows completely different doc ids.

A map entry points at the old mutable segment and one of its doc ids, so simply swapping segment objects would leave a broken locator. Replacement must remap the winner into the committed copy.

Pinot maps accepted rows from mutable positions into immutable positions, then moves the primary-key location and valid bitmap to the matching committed row. The physical identity changes while the comparison value and logical winner remain the same.

So the physical address changes while the logical winner stays the same. The remap must finish before the old mutable locator disappears. That ordering prevents a locator from pointing into a retired segment. Next we will use stable segment facts to make every replica choose the same location when comparison values tie.

Deterministic Tie Resolution

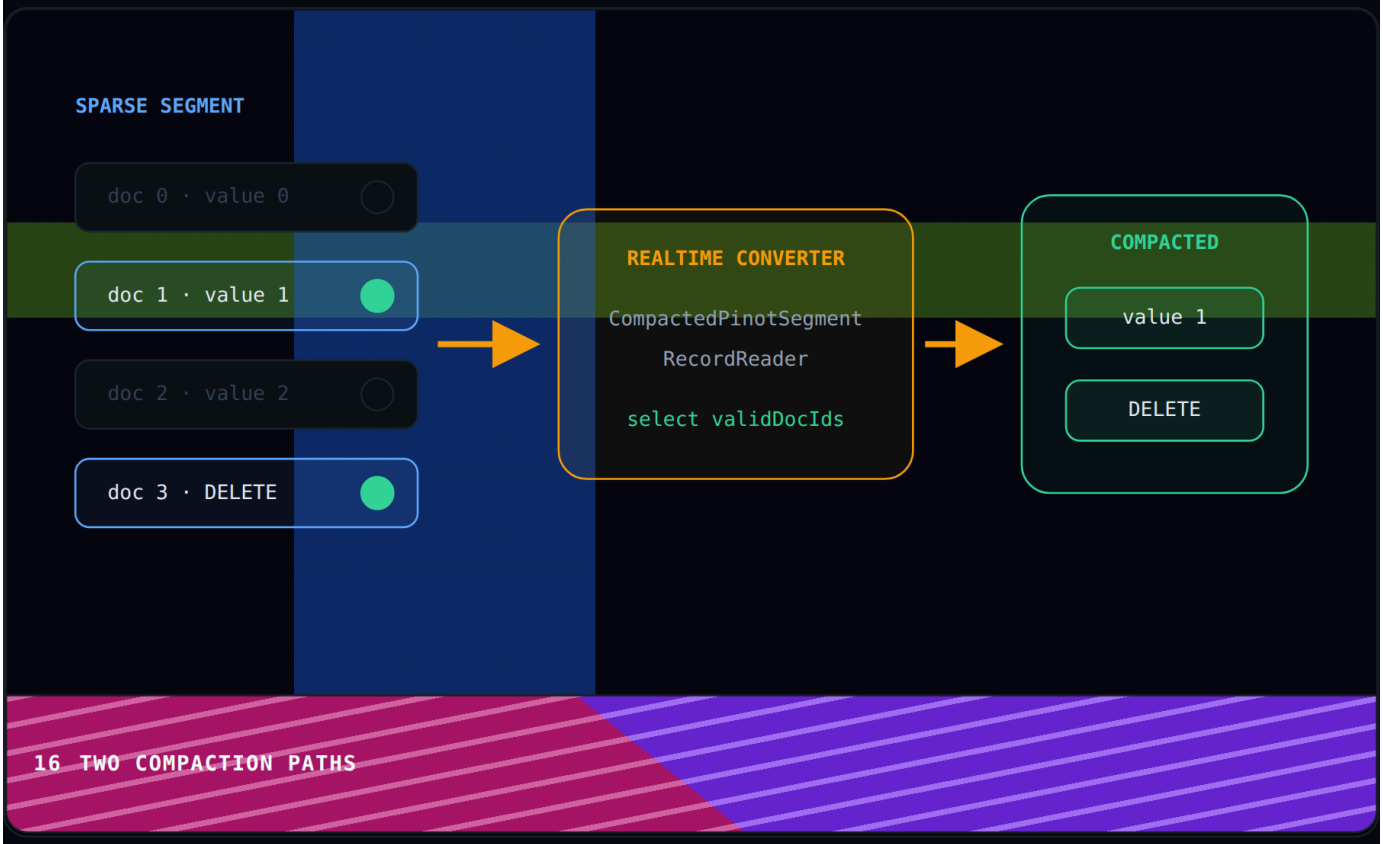
Let’s focus on that equal-value case. The comparison column cannot choose between old and new physical copies, and simply accepting whichever segment loads first could make two replicas disagree.

When two records carry equal comparison values, accepting whichever segment loaded first would make replica state depend on timing. A stable segment identity must break the tie instead.

LLC segments use their sequence relationship, while uploaded segments compare authoritative creation time and then segment naming. Those source-backed rules let every replica select the same physical winner.

The point of these tie breakers is not to find a newer business value. Their job is to make physical placement deterministic on every replica. Equal business time still needs one reproducible storage address. Replica agreement therefore comes from source-backed segment ordering, not arrival luck. This matters whenever equal values appear during load or replacement. Next we’ll use that agreed valid bitmap to remove losing rows through compaction.

Two Compaction Paths



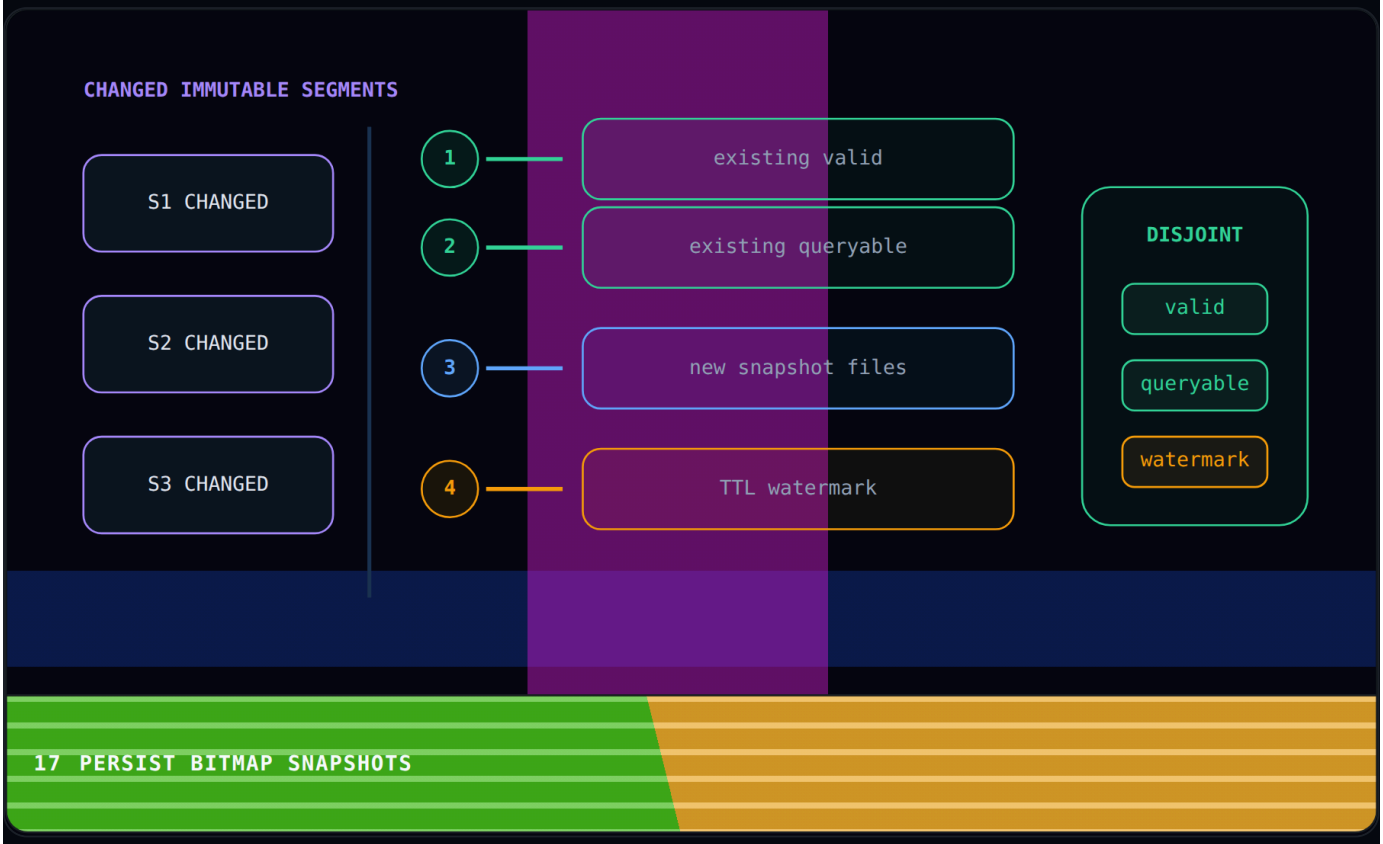
We have hidden old versions from queries, but they still consume disk. Think of compaction as taking the rows selected by validDocIds and packing them into a cleaner segment without changing the logical answer users already receive.

Pinot can compact while committing a consuming segment or let a later Minion task rewrite an immutable segment. Each path must obtain validDocIds evidence at its own execution time.

Commit-time compaction filters rows during realtime conversion, while the Minion path later consumes persisted validDocIds snapshots. Both paths produce the same sparse selection from different lifecycle points.

Both paths remove physical waste while preserving winners and authoritative tombstones. Commit time uses the live consuming context, while Minion work depends on persisted evidence. The logical answer stays stable throughout either cleanup path. Next we'll save the bitmap evidence itself because Minion compaction and fast recovery both need a trustworthy record of which positions were valid.

Persist Bitmap Snapshots



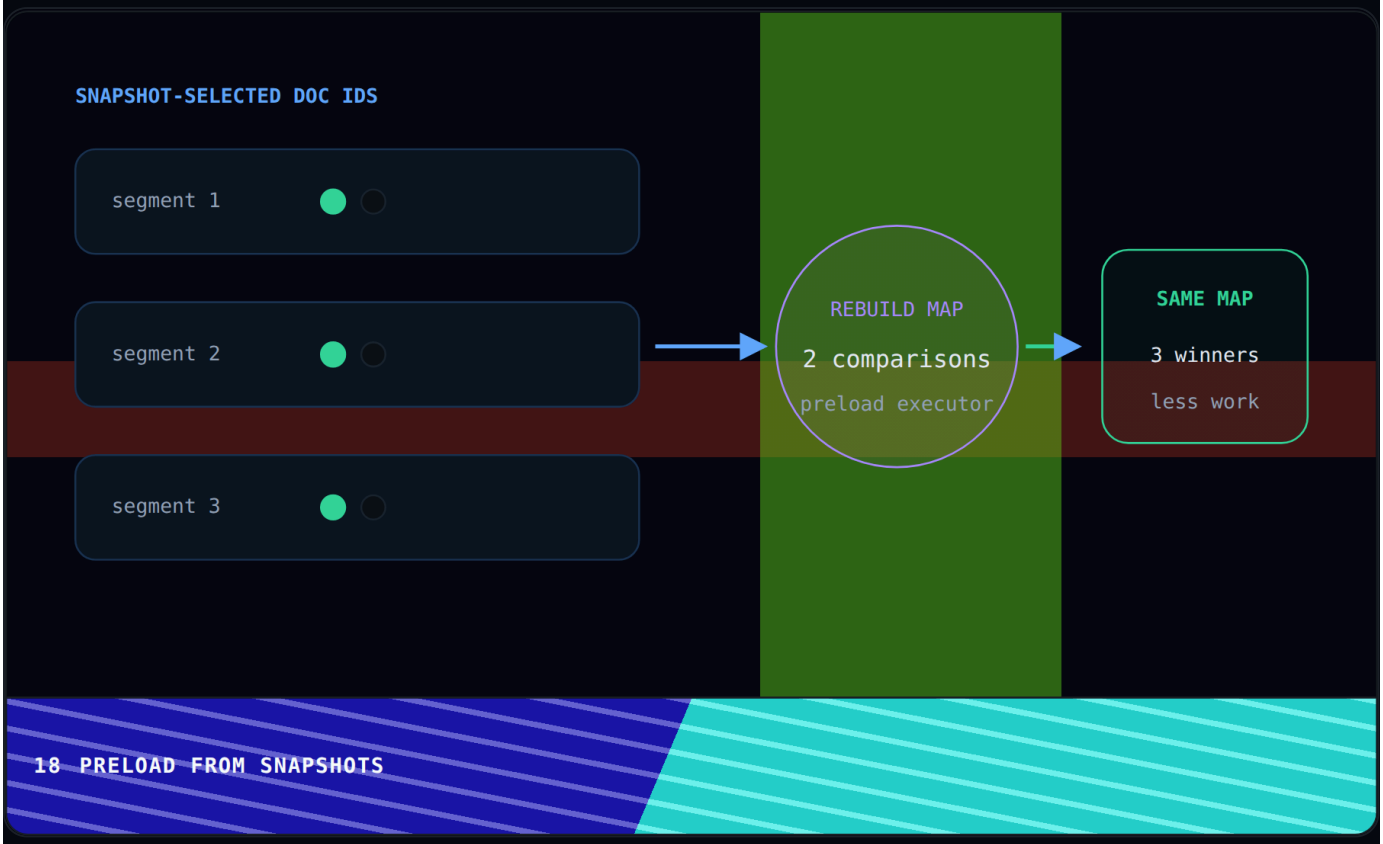
Now let’s save validDocIds and queryableDocIds beside the immutable segment data. Pinot cannot write every snapshot file as one giant transaction, so the order of those smaller writes must prevent a half-finished generation from looking complete.

Some segments already have snapshot files while newly immutable segments do not, and any write can fail. The write ordering must prevent two incomplete generations from looking like one complete set.

Pinot updates snapshot files that already exist before creating files for newly immutable segments. Only after both bitmap sets succeed does it persist the TTL watermark, because that watermark claims the snapshots safely cover a newer comparison range.

Here’s why the order matters: after a failure, Pinot should find either the older coherent snapshot or the complete newer one, not a misleading mixture. The watermark certifies the completed generation. Recovery can then trust that saved boundary after a restart. Next preload uses those selected positions to skip obsolete rows during startup.

Preload from Snapshots



We now have snapshot files that point to the rows known to be valid in a coherent saved view. Let’s use them as a shortcut instead of comparing every obsolete physical version again during startup.

A full scan still works and remains the fallback, but large upsert tables may contain many invalid historical rows. Snapshot-selected positions can eliminate most repeated comparison work.

Both recovery paths reconstruct the same primary-key locations. Preload cursors visit only rows selected by snapshots, while full-scan cursors walk every physical row before finding those same winners.

Switch the Recovery control between Full scan and Preload. Compare the rows entering the rebuild map, because both paths recover the same winners with very different comparison work.

So preload spends snapshot work earlier to save comparison work during restart, and it needs both those files and a preload executor. Next we’ll use the saved watermark to decide how long old winner entries should stay in memory.

Metadata TTL Watermark

Preload helps with restart time, but it does not solve memory growth. Imagine a long-lived table with millions of keys that never update again. Their winner entries can remain in memory forever unless Pinot has a safe expiration rule.

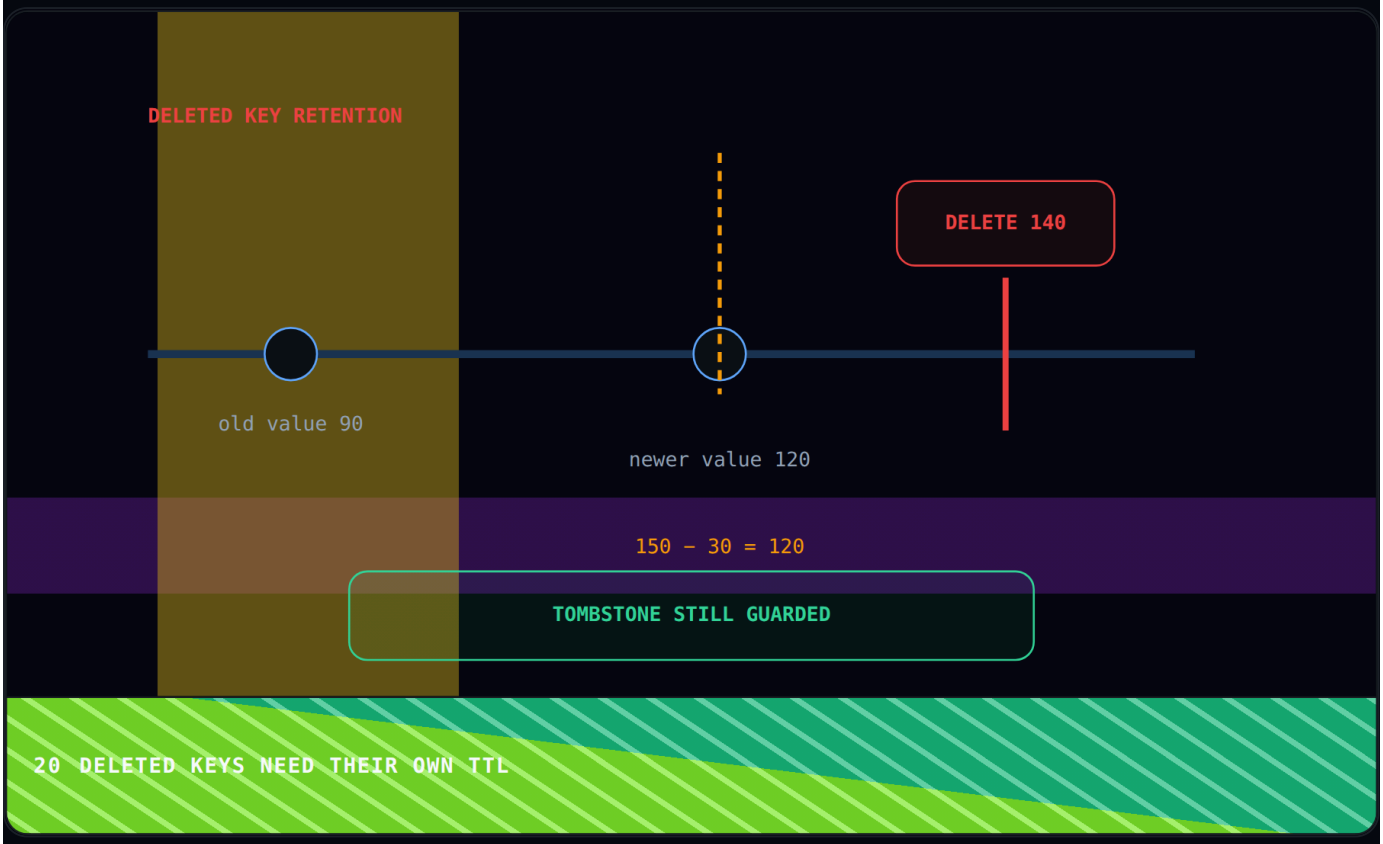
Pinot tracks the largest comparison value observed and treats it as the moving edge of event time. The expiration line sits one configured window behind that edge.

The cutoff equals the largest seen value minus metadataTTL. Winner locations older than that threshold leave the map, which bounds memory by assuming those keys will not receive another relevant version.

Adjust the Metadata TTL control and follow the cutoff across the winner locations. Notice which keys leave the map as the retained comparison-value window shrinks.

Treat this as a correctness promise, not ordinary cache cleanup: you are saying events older than the window no longer need upsert protection. Pinot therefore requires snapshots and one comparison column, and next we'll give deleted keys their own safer window.

Deleted Keys Need Their Own TTL



A deleted key needs extra care because its tombstone is actively suppressing older live values that may still exist in segments. If Pinot forgets that tombstone too early, restart can mistake one of those older rows for the current value.

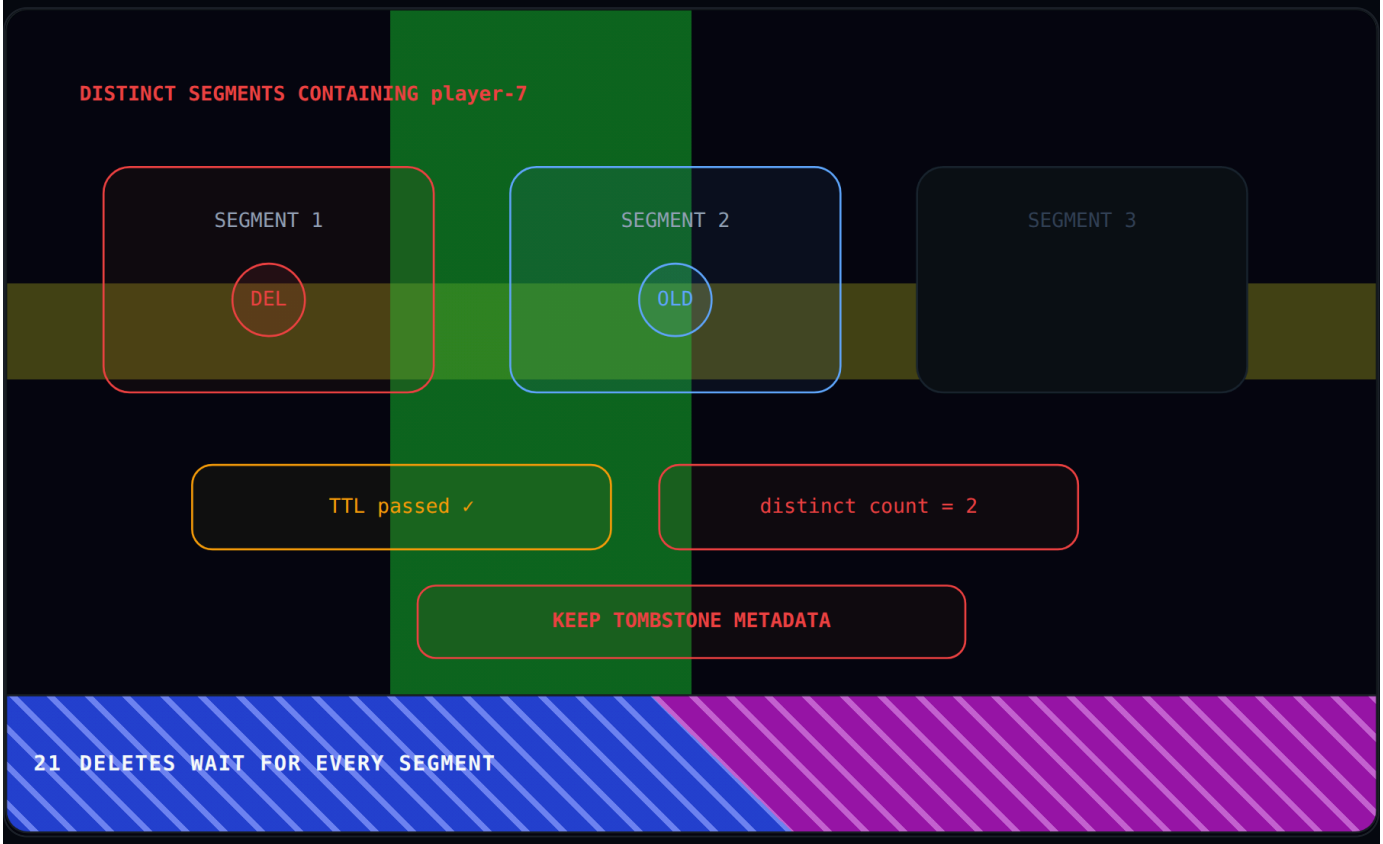
The tombstone comparison value gives Pinot a time boundary, but safe retention depends on how late older copies can still appear. Deletion authority needs its own configurable protection window.

The protected interval grows or shrinks independently from metadataTTL. The tombstone stays in the primary-key map until its comparison value falls behind this delete-specific cutoff.

Adjust the Deleted keys TTL control across its cutoff. Compare when the tombstone stays guarded and when time alone makes its metadata eligible for cleanup.

So deletedKeysTTL lets you keep deletion authority longer than ordinary winner metadata. But time is only half the safety check because compaction runs gradually, so next we’ll count how many segments still contain the key.

Deletes Wait for Every Segment



Let’s slow down at a dangerous moment. The delete TTL has expired, one compacted segment has already dropped the tombstone, and another segment still contains an older live value for the same key.

If Pinot removed deletion metadata at that moment, a restart could scan the older segment and elect its live row as winner. The manager therefore needs evidence about remaining physical copies.

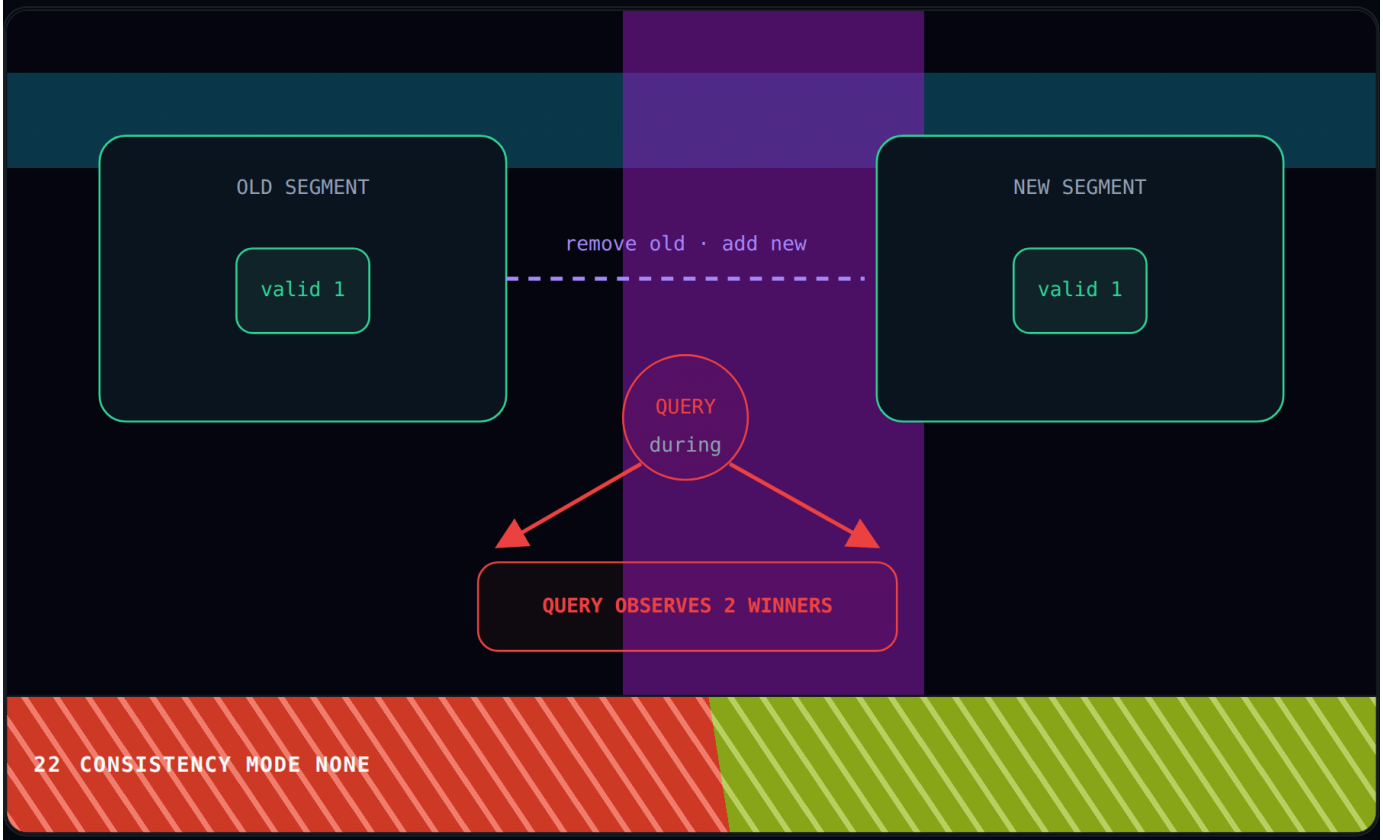
The tombstone remains guarded while multiple distinct segments still contain the key. Cleanup becomes eligible only after TTL has passed and no more than one remaining segment can contribute that key.

Change the Copies control from three toward one. Compare the cleanup verdict, which stays blocked until no older live copy can survive in another segment.

The safety rule is simple to say: time must pass and extra physical copies must disappear. With cleanup covered, let’s move from old files to live queries that can overlap the cross-segment bitmap handoff we saw earlier.

Consistency Mode NONE

Try it in Watch mode. Make the query observe one coherent winner without changing consistency mode.



Recall the cross-segment handoff where Pinot clears the old valid bit and sets the new one. A live query can arrive between those two in-memory changes, so now we need to decide what kind of consistency that query receives.

Consistency mode NONE lets queries read those live bitmaps without a coordinating view manager. One query can therefore capture segment contexts from opposite sides of the handoff.

A query before the handoff sees one stable old winner. A query during the handoff can assemble contexts from different moments and observe a duplicate or missing logical row.

Switch the Interleaving control between Before and During. Compare the query verdict, then solve the challenge by choosing the timing that keeps one coherent winner.

Here’s the tradeoff in NONE: writes stay flexible and Drop or Mark remain available, but queries do not get one atomic bitmap moment. Next SYNC adds a read-write lock so readers receive either the complete old view or the complete new view.

SYNC Locks the Handoff

Let’s fix that mixed-time query with SYNC. The writer takes an exclusive lock across both bitmap changes while each query takes the shared side of the same lock before collecting its segment contexts.

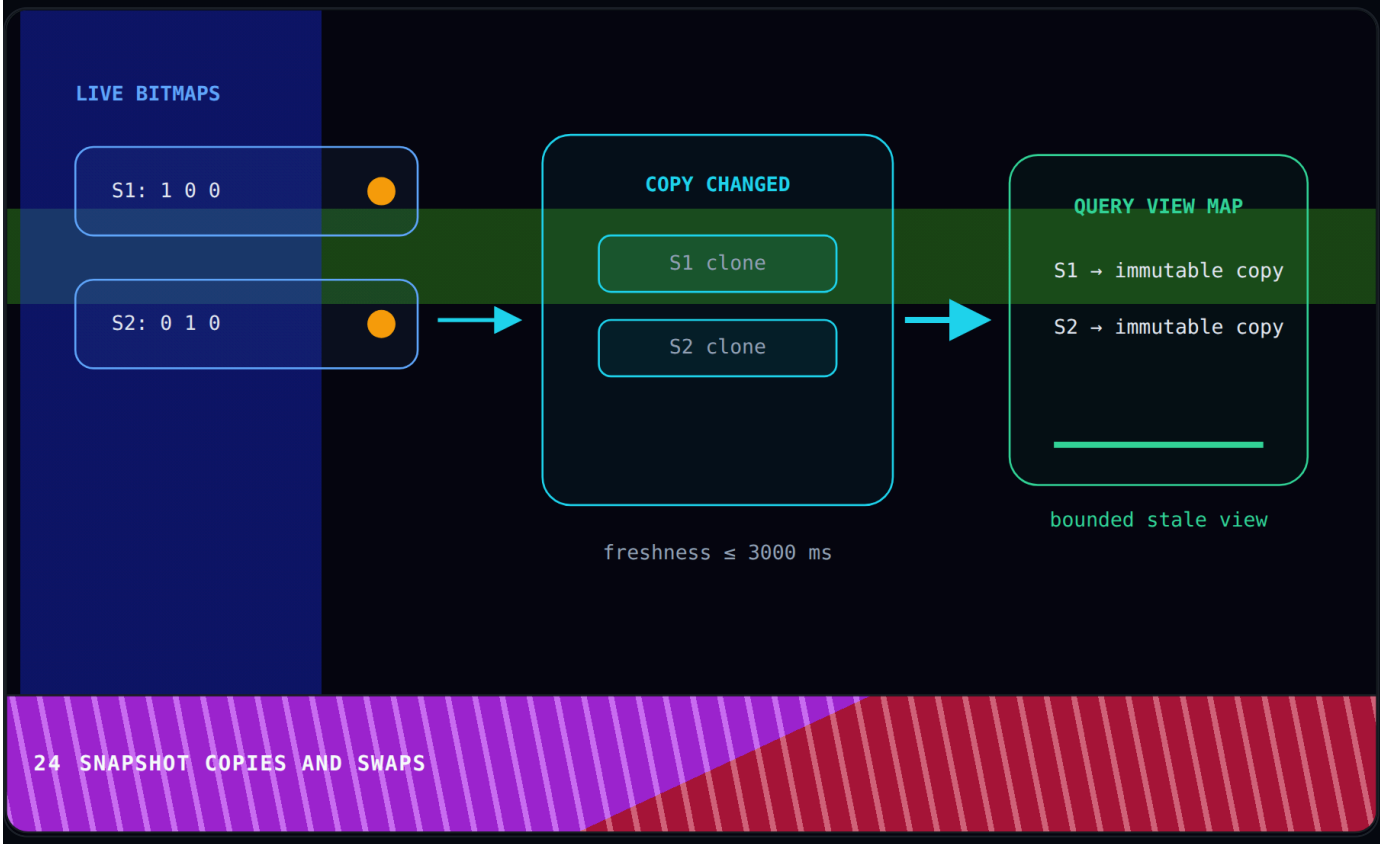
A query arriving before the writer can capture the old coherent view, while one arriving afterward captures the new view. A request during the critical section must wait.

The middle query waits outside the writer lock instead of collecting half-updated bitmaps. Every returned context set therefore represents one complete side of the handoff.

Switch the Query timing control among Before write, During write, and After write. Compare the returned view and the waiting state around the writer lock.

So SYNC gives every query a fresh, atomic view, but an overlapping reader may have to wait. Next SNAPSHOT chooses a different tradeoff: lock-free reads in exchange for copied bitmaps and a controlled amount of staleness.

SNAPSHOT Copies and Swaps



Now let’s remove that reader lock from queries. SNAPSHOT lets ingestion keep changing live bitmaps while queries read an immutable map of copied bitmaps from the most recently published view.

Copying each changed segment at an arbitrary time could still assemble a mixed generation. Pinot must publish the lock-free view as one internally consistent object.

Changed live bitmaps copy into a new map before one atomic pointer swap publishes the complete query view. Readers either retain the previous immutable map or receive the new map, never a partial container.

Adjust the Freshness control down to zero and back. Compare when a query forces new bitmap copies against when it can reuse the bounded-stale immutable view.

Here’s the SNAPSHOT bargain: queries avoid locks, but the view can be slightly old and Pinot must pay to copy changed bitmaps. Next we’ll handle a different problem, where a coherent view is missing a brand-new segment entirely.

STAGE 25 OF 27

Track Newly Added Segments

A perfectly consistent view can still be wrong if it is missing an input. When a server creates a new consuming or uploaded segment, the broker may briefly keep routing requests with an older segment list.

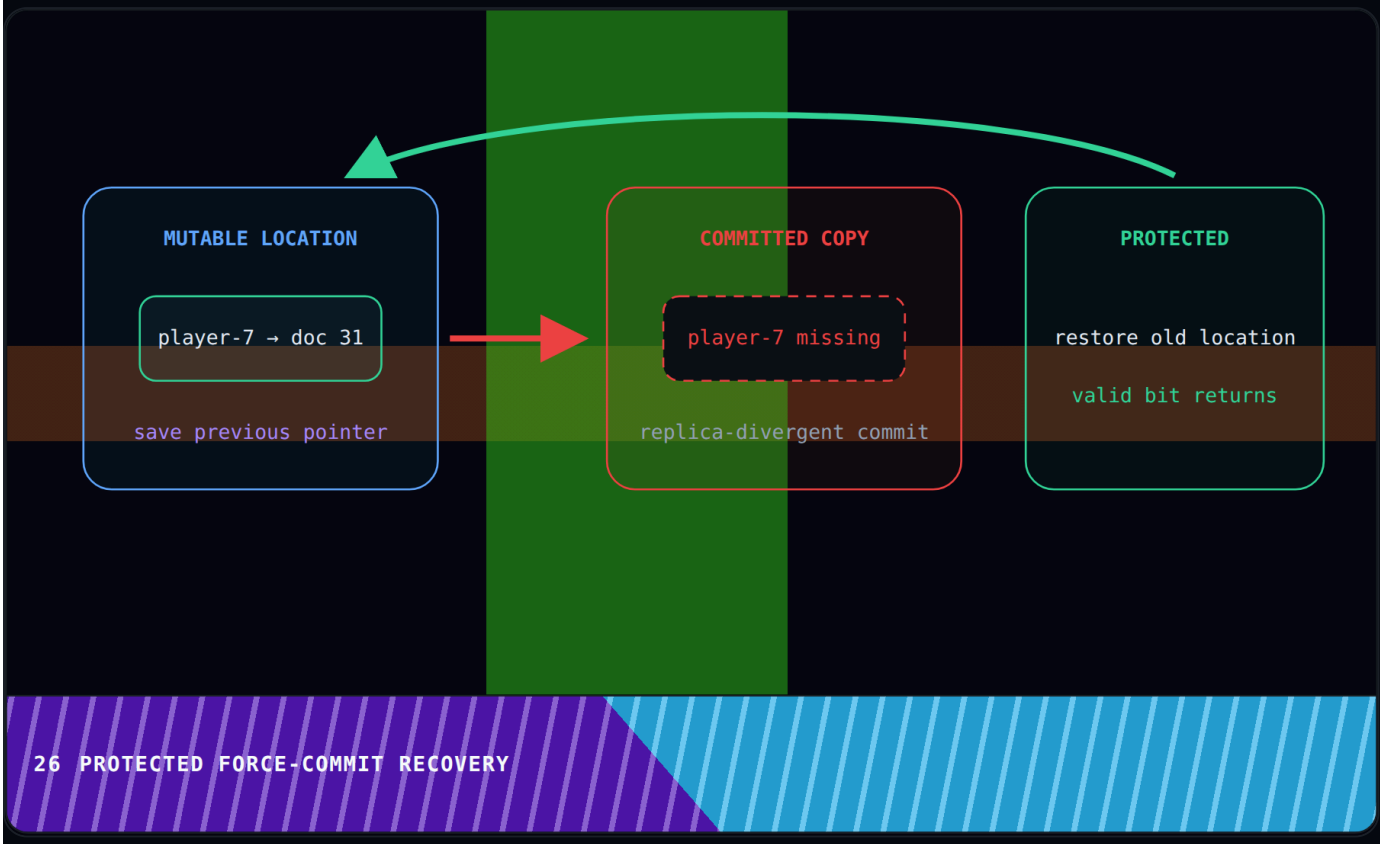
The new segment may already contain the latest winner while the broker request names only older routed segments. The server can repair that temporary routing gap with local knowledge.

When broker routing lags, the server adds its tracked newly created segment as an optional query input. That local addition restores the newest winner until routing names the segment normally.

Switch the Broker routing control between Caught up and Lagging. Compare whether the broker names the new segment or the server supplies it as an optional input.

So the server temporarily fills the routing gap with local knowledge until the broker catches up. In our final technical stage, we'll handle another transition failure: a force-committed segment that is missing keys from the mutable copy it replaces.

PROTECTED Force-Commit Recovery



Let’s finish with a force-commit edge case. Replicas may have handled stale rows differently. The committed segment uploaded by one replica can therefore omit a key that another replica’s mutable segment used to replace an older location.

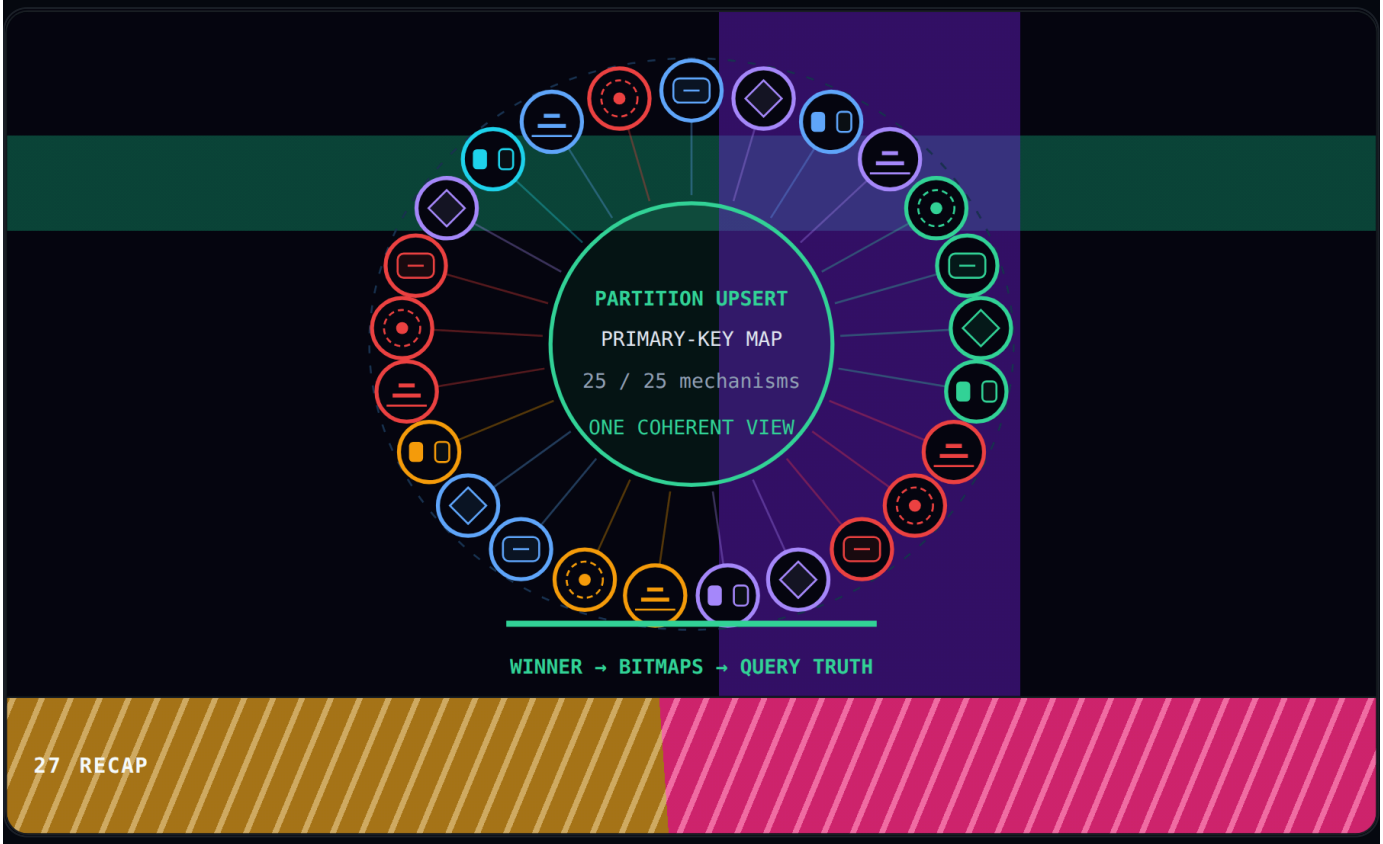
Ordinary replacement moves winners into the committed segment, but it cannot move a row absent from that copy. Recovery therefore depends on remembering the previous valid location.

PROTECTED mode retains previous record locations during replacement. For every key missing from the committed copy, the saved pointer reverses the handoff and restores the old segment valid bit.

Switch the Cluster mode control between DEFAULT and PROTECTED. Compare whether the missing key stays lost or the saved previous location restores its valid bit.

That reverse handoff is what makes PROTECTED mode valuable: replacement can recover instead of silently losing the key. We have now covered selection, visibility, cleanup, recovery, retention, and query consistency, so let’s connect the whole story in the recap.

Recap



Let’s start our recap with the foundation: Pinot appends every accepted version physically while bitmaps expose one logical winner per primary key. That separation lets every later feature change metadata without pretending the old bytes disappeared.

Now connect that idea to the next piece: Every key must stay in one partition because each partition owns an independent upsert metadata map. One owner is what makes a single winner decision possible instead of requiring coordination between partition maps.

Here is the rule to carry forward: RecordInfo carries the primary key, doc id, comparison value, and delete flag into the winner decision. This packet is small enough for the metadata path but complete enough to drive every later branch.

Next, remember this: The concurrent map points each hashed primary key at one segment, doc id, and comparison value. Keeping only a locator prevents the in-memory winner map from becoming a duplicate copy of segment storage.

Now connect that idea to the next piece: A greater comparison value wins, a smaller value is stale, and an equal value needs a deterministic tie rule. The comparison gate is the common ordering rule behind live ingestion, segment loading, replacement, and recovery.

Here is the rule to carry forward: Within one segment, a newer row appends while the valid bit moves from the old doc id to the new one. The physical append and logical bitmap move are two views of the same accepted update.

Next, remember this: Across segments, Pinot removes the old valid bit and adds the new valid bit as one metadata handoff. Cross-segment handoff preserves the one-winner invariant even when the winning row changes storage owners.

Now connect that idea to the next piece: Queries intersect segment rows with validDocIds so obsolete physical versions never enter normal results. This intersection is the exact point where append-only history becomes an overwrite-like query result.

Here is the rule to carry forward: A tombstone can remain the valid winner while queryableDocIds hides it from ordinary queries. Separate authority and queryability are why a delete can suppress older values without appearing as a live row.

Next, remember this: A stale row is normally stored physically but receives no valid bit, so it cannot replace the winner. Rejecting a locator is different from deleting bytes, which explains why stale rows can remain available for diagnosis.

Now connect that idea to the next piece: NONE mode can keep, drop, or mark stale rows, but drop and mark are unavailable with consistent views. Those storage policies are useful, but their metadata-first order is incompatible with the stronger consistent-view modes.

Here is the rule to carry forward: Loading sealed segments scans RecordInfo values and reconstructs the same partition map and bitmaps. Replaying the same RecordInfo decisions means startup reconstructs the same truth as live ingestion.

Next, remember this: Commit replacement remaps mutable doc ids to immutable doc ids without changing the logical winner. Replacement preserves the logical entity even when sorting or compaction gives its row a different physical address.

Now connect that idea to the next piece: Equal comparisons use LLC sequence, authoritative segment time, and uploaded naming for deterministic placement. Stable tie breakers prevent replicas from choosing different locations merely because they loaded segments in another order.

Here is the rule to carry forward: Commit-time and minion compaction remove invalid physical versions while preserving the same logical answer. Compaction changes physical cost while leaving the bitmap-defined answer unchanged for users.

Next, remember this: Snapshots persist valid and queryable bitmaps before persisting the TTL watermark that depends on them. The watermark comes last because it may advance only after the bitmap files it depends on are safely published.

Now connect that idea to the next piece: Preload rebuilds metadata from snapshot-selected rows instead of comparing every physical row at startup. Preload uses prior snapshot work to skip obsolete rows while rebuilding the identical winner map.

Here is the rule to carry forward: metadataTTL removes old primary-key locations below largestSeen minus TTL and requires one comparison column. That memory bound is safe only when the configured event-time window matches the application’s late-arrival behavior.

Next, remember this: deletedKeysTTL keeps tombstones long enough to block older values before their metadata can be forgotten. Delete retention deserves its own window because forgetting authority can make an older live value visible again.

Now connect that idea to the next piece: Consistent delete cleanup waits until a deleted key occurs in no more than one remaining segment. Checking both time and remaining segments keeps staggered compaction from turning deletion into resurrection.

Here is the rule to carry forward: NONE consistency maximizes write freedom but a query can observe bitmap changes from different moments. The cost of NONE is that a query may combine segment bitmaps captured on opposite sides of one handoff.

Next, remember this: SYNC holds a writer lock across old-bit removal and new-bit addition while queries take the reader lock. SYNC removes that mixed moment by making readers wait until the writer publishes the entire update.

Now connect that idea to the next piece: SNAPSHOT copies changed bitmaps and atomically swaps a query view, trading bounded freshness for lock-free reads. SNAPSHOT removes reader blocking by publishing immutable bitmap maps, with copy cost and bounded staleness as the tradeoff.

Here is the rule to carry forward: Servers temporarily add newly created segments as optional query inputs while broker routing catches up. Server-local segment tracking preserves completeness during the short window before broker routing catches up.

Next, remember this: PROTECTED force-commit handling remembers previous locations and restores keys missing from the committed copy. PROTECTED closes the lifecycle by making a risky force-commit replacement reversible when its committed rows diverge.

Step back and look at the complete system: one partition map chooses each winner, while bitmaps control visibility and snapshots preserve recoverable evidence. Compaction, TTL, and consistency modes then protect that same logical choice as physical rows move, disappear, restart, and meet live queries over time.