READ MODE · ANSWER FIRST

Apache Flink Exactly-Once Ingestion

An interactive, visual explanation of Apache Flink Exactly-Once Ingestion, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

The whole story in 6 lines

Apache Flink Exactly-Once Ingestion becomes easier to reason about when every stage is connected as one system.

1. Establish the baseline: records can be in flight, state is mutable, and sink output is still private.
2. Teach that barriers divide pre-checkpoint data from post-checkpoint data without stopping the world.
3. Show alignment clearly: the early channel blocks while the late channel catches up, or unaligned mode snapshots the in-flight buffers.
4. Pair state handles with sink prepare work so the learner sees both halves of exactly-once.
5. Make the global completion signal and sink commit boundary explicit so learners understand why duplicates are avoided.
6. Close the loop: source offsets, restored state, and aborted pending transactions explain exactly-once under failure.

STAGE 1 OF 8

Setup

Welcome. Before we start, let us cover the big picture. Flink processes streams of data in real time and guarantees that every record affects the final output exactly one time, even if something crashes. Think of it like a relay race where every handoff is recorded on camera so nothing gets lost or counted twice.

A checkpoint is a saved snapshot of the entire job at one specific moment. It captures where every source was reading, what every operator had stored, and what the sink was about to publish. If anything goes wrong, Flink rewinds to the last checkpoint and picks up right there.

A barrier is a special marker that Flink injects into the data stream. It does not carry any real data. Instead, it acts like a dividing line: everything before the barrier belongs to this checkpoint, and everything after belongs to the next one.

Exactly-once means every record is processed and reflected in the output exactly one time. Not zero times (lost), not two times (duplicated). Flink achieves this by tying checkpoints to external commits so nothing publishes unless the full checkpoint succeeds.

The state backend is where each operator keeps its working memory, like a student keeping notes in a notebook. A two-phase commit means the sink first prepares its output, then only commits it after the coordinator says the checkpoint is globally complete. A savepoint is a checkpoint you trigger manually, usually before an upgrade.

Over the next six stages we will build up the complete picture: how live data flows, how barriers cut the stream, how multi-input operators align, how state gets frozen, how commits happen globally, and how crash recovery rolls everything back safely. Let us begin.

STAGE 2 OF 8

Live Stream + State

Before any checkpoint begins, Flink is already pulling live records from Kafka partitions. Try changing the Partitions stepper to see how the source fans out.

Each source subtask reads records and pushes them downstream immediately. The records are in flight, not durable yet.

The keyed operator hashes each record by key and updates mutable keyed state. You can flip the Load control to Hot key to watch one key receive most of the traffic.

Source offsets and keyed state advance together inside the running task. We now have two things that need protection: offsets and state.

The sink writes rows into an open transaction, but the outside world cannot see them yet. Think of it like writing answers in pencil before the teacher collects the test.

So the baseline is clear: records flow, state mutates, and sink output stays private. Next we will see how Flink cuts a checkpoint barrier through this live stream to start making things durable.

STAGE 3 OF 8

Barrier Cut

We just saw records flowing through source, operator, and sink. Now Flink needs to draw a line through that live stream so it can snapshot a consistent moment. That line is a checkpoint barrier.

The checkpoint coordinator triggers checkpoint 17 and tells every source channel to inject a barrier. Try changing the Partitions stepper to see more or fewer channels receive the barrier at the same time.

Each source inserts the barrier into its output stream. Records already ahead of the barrier keep moving forward. The barrier does not rewrite history.

You can think of the barrier like a bookmark slipped between pages. Everything before the bookmark belongs to this checkpoint. Everything after belongs to the next one.

Switch the Cadence control to Wide to see more data sitting behind the barrier. Post barrier records cannot overtake the barrier, so the cut stays clean.

The takeaway: barriers divide the stream without stopping it. Every channel now carries a precise boundary for checkpoint 17. Next we will see what happens when a multi input operator receives barriers from different channels at different times.

STAGE 4 OF 8

Alignment Gate

04 ALIGNMENT GATE

Barriers are now flowing through the stream, but what happens when an operator has two inputs? One channel will almost always receive barrier 17 before the other. Watch the fast lane and slow lane to see this skew.

Barrier 17 reaches the fast lane first. The operator now has a partial checkpoint boundary because it has seen the barrier on one input but not the other.

Switch the Mode control between Aligned and Unaligned to see the difference. In aligned mode, Flink blocks the fast lane at the gate and buffers its new records. In unaligned mode, the barrier moves on immediately and carries the buffered state with it.

The slow lane keeps draining records until its own barrier 17 arrives. Try the Lag control at High to see a bigger gap between the two barriers.

Once both barriers are accounted for, the operator snapshots a consistent state and forwards barrier 17 downstream. Both lanes resume normal flow.

The takeaway: alignment is the cost of consistency at multi input operators. Whether we block or snapshot buffers, the checkpoint still represents one exact position across all inputs. Next we will see what the operator does with that consistent snapshot.

STAGE 5 OF 8

Snapshot + Prepare

Barrier 17 has fully aligned at this task, so the task can now freeze its local view. This is the snapshot step we have been building toward since the barrier was first injected back in Stage 2.

The task copies keyed state and source offsets into a checkpoint snapshot. You can increase the State stepper to add more snapshot chunks and see how the volume grows.

Those snapshot chunks are written to checkpoint storage and turned into durable handles. Think of each handle like a receipt you can use to retrieve the data later.

At the same time, the sink seals the output it has written so far into a pending transaction for checkpoint 17. Try adjusting the Rows stepper to see more or fewer committable batches staged in the sink.

Only after both state handles and sink preparation are ready does the task send an acknowledgment for checkpoint 17 back to the coordinator.

The takeaway: we now have a recoverable state snapshot plus a prepared sink transaction, but neither is globally committed yet. Next we will see how the coordinator collects acknowledgments and triggers the final commit.

STAGE 6 OF 8

Checkpoint Commit

06 CHECKPOINT COMMIT

Every task has now snapshotted its state and prepared its sink transaction, just as we saw in the previous stage. The coordinator is waiting for acknowledgments from all participating writers. Try adjusting the Writers stepper to change how many tasks must report in.

Acknowledgments stream back to the coordinator one by one. Each ack confirms that the task has durable state handles and a sealed sink transaction for checkpoint 17.

Once the last acknowledgment arrives, the coordinator marks checkpoint 17 complete globally. This is the single moment that turns a collection of local snapshots into one atomic checkpoint.

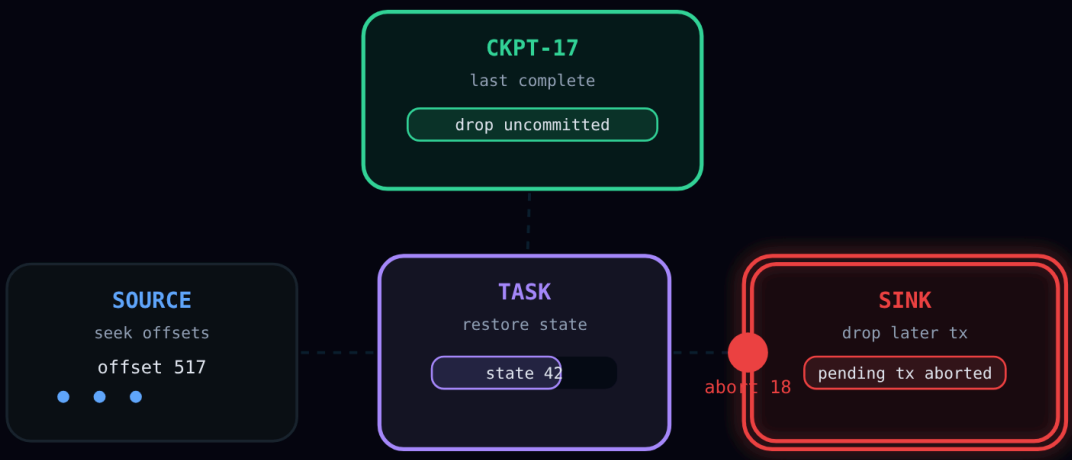
That global completion signal unlocks the sink committer for checkpoint 17 only. The Pending stepper shows how many prepared checkpoints the sink still tracks. Only the completed one can commit.

The committer publishes exactly one transaction for checkpoint 17 while leaving newer checkpoints pending. This is the first time any output from this checkpoint becomes visible to the outside world.

The takeaway: only checkpoints that completed everywhere become visible outside Flink. This delayed commit is what prevents duplicates. Next we will see what happens if the job crashes before or after this commit.

STAGE 7 OF 8

Crash Recovery



07 CRASH RECOVERY

We just saw checkpoint 17 commit successfully. But what if the job crashes before the next checkpoint completes? Newer records and a later sink transaction may still be in flight. You can use the Replay stepper to control how many records arrived after checkpoint 17.

A crash interrupts the job. All in flight records and uncommitted sink work are lost. Switch the Crash point control to see the difference between crashing before or after the latest sink commit.

On restart, Flink reloads task state and source offsets from the latest completed checkpoint, checkpoint 17. This is the durable snapshot we built in stages 3 and 4.

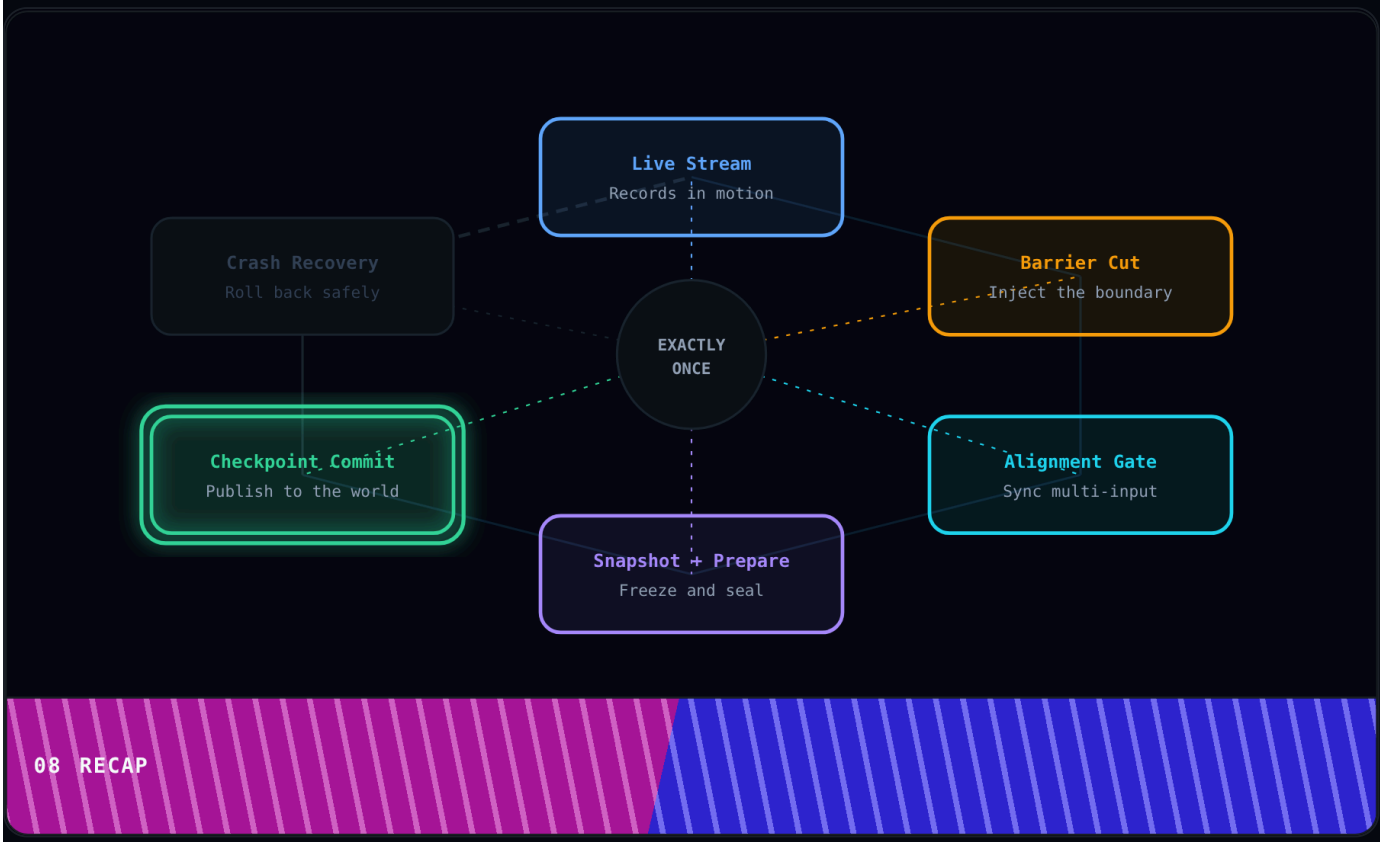
If the crash happened before commit, the newer sink transaction is aborted and disappears. If the sink had already committed, Flink keeps that output and resumes after it. Either way, no duplicates appear.

The source replays records after checkpoint 17 from the saved offsets. Because state was restored to the same point, these replayed records rebuild the same state transitions deterministically.

The takeaway: only completed checkpoints can publish external output, so replay never produces duplicate committed rows. This closes the exactly once loop. Let us now recap the full cycle from start to finish.

STAGE 8 OF 8

Recap



08 RECAP

It all starts with the live stream. Records flow in from Kafka partitions, operators update their state, and the sink stages writes privately. Remember from Stage 1: nothing is visible to the outside world yet.

Then the coordinator injects a barrier into every source channel. As we saw in Stage 2, this barrier draws a clean line between "before this checkpoint" and "after it" without stopping the data flow.

When an operator has multiple inputs, the barriers may arrive at different times. Stage 3 showed how Flink either blocks the early channel (aligned) or snapshots the in-flight buffer (unaligned) to keep the cut consistent.

Once all barriers have aligned, each task freezes its state and the sink seals a pending transaction. Stage 4 called this "snapshot plus prepare." The job now has a recoverable picture of that exact moment.

Only after every task acknowledges the checkpoint does the coordinator mark it complete. Stage 5 showed how the sink committer then publishes the prepared transaction, making output visible for the first time.

If a crash happens before the next checkpoint completes, Stage 6 showed that Flink restores from the last completed checkpoint, replays records from saved offsets, and aborts any uncommitted sink work. No duplicates, no lost data.

And that is the full cycle. Barriers cut the stream, alignment keeps multi-input operators consistent, snapshots freeze the moment, commits publish atomically, and recovery rolls back cleanly. This loop repeats with every checkpoint, giving Flink its exactly-once guarantee from source all the way to sink.