

Apache Flink Reference

Stateful stream processing — event time, watermarks, windows, exactly-once state, and Flink SQL.

Last reviewed 29 Sep 2026 ·

Prerequisites: [Kafka](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) · [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/)

Related: [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) · [Data Ingestion & CDC](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/) · [Apache Iceberg](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) · [Beam and Dataflow](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/beam-dataflow/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/beam-dataflow/) · [Streaming SQL](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/streaming-sql/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/streaming-sql/) · [System Design](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Practice: [Lab 04 — Kafka Streaming](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/04-kafka-streaming) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/04-kafka-streaming)

Overview

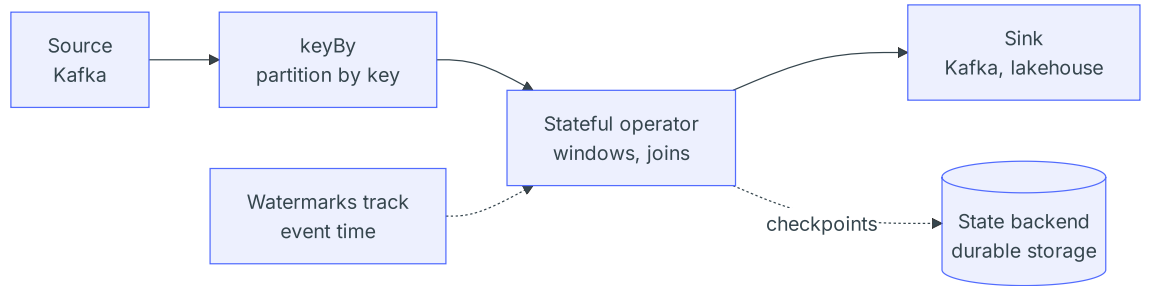
Challenge: Some decisions can't wait for a nightly batch — fraud checks, real-time inventory, alerting, and live metrics need results within seconds. Processing unbounded streams correctly is hard: events arrive late and out of order, aggregations need state that survives failures, and results must not be double-counted after a restart.

Solution: Apache Flink is a distributed engine built for stateful stream processing. It processes each event as it arrives (not in micro-batches), keeps large keyed state locally with periodic consistent **checkpoints** to durable storage, handles out-of-order data with **event time and watermarks**, and provides exactly-once state consistency. The same engine also runs batch jobs.



```
sources (Kafka, CDC, files) → Flink job  
└──────────────────────────→ sinks (Kafka,  
lakehouse tables,  
                               | parse → key by customer  
→ window/aggregate    databases, search indexes)  
                               | state per key (local,  
fault-tolerant)  
                               └ checkpoints → durable  
storage (S3/GCS/HDFS)
```

Relevance to data engineering: Flink powers low-latency pipelines — streaming ETL into lakehouse tables, real-time aggregations, CDC processing, and event-driven applications. Flink SQL makes most of it accessible without writing Java.



On this page

Basic - Core Concepts (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#core-concepts>) - **Architecture** (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#architecture>) - **Flink SQL Basics** (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#flink-sql-basics>)

Intermediate - [Time and Watermarks](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#time-and-watermarks) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#time-and-watermarks) - [Windows](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#windows) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#windows) - [Joins in Streams](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#joins-in-streams) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#joins-in-streams) - [PyFlink](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#pyflink) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#pyflink)

Advanced - State and Checkpoints (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#state-and-checkpoints>) - **Exactly-Once End to End** (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#exactly-once-end-to-end>) - **CDC and Lakehouse Pipelines** (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#cdc-and-lakehouse-pipelines>) - **Deployment and Operations** (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#deployment-and-operations>)

Reference - Common Pitfalls (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#common-pitfalls>) - **Cheat Sheet** (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#cheat-sheet>) - **Interview Questions** (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#interview-questions>) - **Further Reading** (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/#further-reading>)

Core Concepts

Concept	Meaning
Stream	Unbounded (continuous) or bounded (batch) sequence of records
Operator	A transformation (map, filter, aggregate, join) running as parallel subtasks
Keyed stream	Records partitioned by key, so all events for a key go to the same subtask and its state
State	Data an operator remembers between events (counts, windows, join buffers)
Event time	When the event happened, taken from the record
Watermark	A marker that says "no more events older than time T are expected" — lets windows close
Checkpoint	Periodic consistent snapshot of all state and source positions, for failure recovery
Savepoint	A manually triggered snapshot for upgrades, rescaling, and migrations
Parallelism	Number of parallel subtasks per operator

Architecture

Client —submit job→ JobManager (scheduling, checkpoint coordination, recovery)

```
graph TD; JM[JobManager] --- H[ ]; H --- TM1[TaskManager]; H --- TM2[TaskManager]; H --- TM3[TaskManager];
```

TaskManager TaskManager TaskManager
each with task slots running operator subtasks

```
(state backend)    (state backend)    (state backend)
and exchanging data over the network
                    |
                checkpoint storage (object storage / HDFS)
```

APIs: Flink SQL and the Table API (declarative, most common for data engineering) ·
DataStream API (Java/Python, full control over state and timers) · PyFlink (Python bindings for both)

Flink SQL Basics

```
-- Source: a Kafka topic as a dynamic table
CREATE TABLE orders (
  order_id      STRING,
  customer_id   STRING,
  status        STRING,
  amount        DECIMAL(12, 2),
  order_ts      TIMESTAMP(3),
  WATERMARK FOR order_ts AS order_ts - INTERVAL '10'
SECOND        -- tolerate 10 s of disorder
) WITH (
  'connector' = 'kafka',
  'topic' = 'order-events',
  'properties.bootstrap.servers' = 'kafka:9092',
  'properties.group.id' = 'flink-orders',
  'scan.startup.mode' = 'earliest-offset',
  'format' = 'json'
);

-- Sink: an upsert topic keyed by customer and window
CREATE TABLE revenue_5m (
  window_start  TIMESTAMP(3),
  customer_id   STRING,
  revenue        DECIMAL(18, 2),
  PRIMARY KEY (window_start, customer_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'revenue-5m',
  'properties.bootstrap.servers' = 'kafka:9092',
  'key.format' = 'json',
```

```

        'value.format' = 'json'
    );

-- Continuous query: runs until cancelled
INSERT INTO revenue_5m
SELECT window_start, customer_id, SUM(amount) AS revenue
FROM TABLE(
    TUMBLE(TABLE orders, DESCRIPTOR(order_ts), INTERVAL '5'
MINUTES)
)
WHERE status = 'placed'
GROUP BY window_start, window_end, customer_id;

```

Dynamic tables: in Flink SQL a stream is a table that changes over time; a continuous query produces another changing table — either append-only (new rows) or an updating/changelog stream (inserts, updates, deletes). Sinks must support the kind of stream the query produces.

Time and Watermarks

```

Event time:  10:00:01   10:00:03   10:00:02 (late, out of order)   10:00:07
Watermark = max event time seen - 10 s → a window ending
at 10:00:05 fires once the watermark passes 10:00:05

```

- **Event time** gives correct, reproducible results even when data is delayed or replayed; **processing time** is simpler but results depend on when events happen to arrive
- The **watermark delay** trades latency for completeness: a larger delay waits longer for late events
- **Idle sources/partitions** can stall watermarks; configure idleness
('scan.watermark.idle-timeout' / table.exec.source.idle-timeout) so one quiet partition doesn't block all windows
- Events arriving after the watermark are **late**; decide whether to drop them, allow lateness in the DataStream API, or correct results downstream (e.g. a batch reconciliation)

Windows

Window	Shape	Flink SQL
Tumbling	Fixed size, no overlap	<code>TUMBLE(TABLE t, DESCRIPTOR(ts), INTERVAL '5' MINUTES)</code>
Hopping (sliding)	Fixed size, overlapping	<code>HOP(TABLE t, DESCRIPTOR(ts), INTERVAL '1' MINUTE, INTERVAL '10' MINUTES)</code>
Cumulative	Growing within a period (e.g. running total per day, updated each minute)	<code>CUMULATE(TABLE t, DESCRIPTOR(ts), INTERVAL '1' MINUTE, INTERVAL '1' DAY)</code>
Session	Grouped by gaps of inactivity	<code>SESSION(TABLE t PARTITION BY user_id, DESCRIPTOR(ts), INTERVAL '30' MINUTES)</code>

```
-- Top 3 products per 10-minute window
SELECT *
FROM (
  SELECT *, ROW_NUMBER() OVER (PARTITION BY window_start,
window_end ORDER BY sales DESC) AS rn
  FROM (
    SELECT window_start, window_end, product_id,
COUNT(*) AS sales
    FROM TABLE(TUMBLE(TABLE order_items,
DESCRIPTOR(order_ts), INTERVAL '10' MINUTES))
    GROUP BY window_start, window_end, product_id
  )
)
WHERE rn <= 3;
```

Joins in Streams

Join	Semantics	State kept
Regular join	Every match across the full history of both sides	Grows without bound — set a state <u>TTL</u>

Join	Semantics	State kept
Interval join	Match events within a time range of each other (order ↔ payment within 1 hour)	Bounded by the interval
Window join	Match events in the same window	Bounded by the window
Temporal join (versioned table)	Join each event with the version of a table valid at the event's time (e.g. FX rate at order time)	Versions of the right-hand table
Lookup join	Enrich events by querying an external database at processing time	Optional cache

```
-- Interval join: payments within one hour of the order
SELECT o.order_id, o.amount, p.payment_id
FROM orders o
JOIN payments p
  ON o.order_id = p.order_id
 AND p.pay_ts BETWEEN o.order_ts AND o.order_ts + INTERVAL
'1' HOUR;

-- Temporal join: currency rate valid at order time (rates
is a versioned table with a primary key and watermark)
SELECT o.order_id, o.amount * r.rate AS amount_usd
FROM orders AS o
JOIN currency_rates FOR SYSTEM_TIME AS OF o.order_ts AS r
  ON o.currency = r.currency;
```

PyFlink

```
from pyflink.table import EnvironmentSettings,
TableEnvironment

t_env =
TableEnvironment.create(EnvironmentSettings.in_streaming_mode())
t_env.get_config().set("execution.checkpointing.interval",
"60 s")
```

```
t_env.execute_sql(open("sql/orders_source.sql").read())
# CREATE TABLE orders ...
t_env.execute_sql(open("sql/revenue_sink.sql").read())
# CREATE TABLE revenue_5m ...

# Submit the continuous INSERT; .wait() blocks (useful for
local runs)
t_env.execute_sql(open("sql/revenue_5m.sql").read()).wait()
```

Python UDFs are supported (`@udf`), but run in a separate Python process — prefer built-in SQL functions on hot paths. Connector JARs (Kafka, Iceberg, JDBC) must be on the classpath (`pipeline.jars` or the deployment image).

State and Checkpoints

State backend	Keeps state in	Use for
HashMap	JVM heap	Small state, lowest latency
RocksDB (embedded)	Local disk, with an in-memory cache	Large state (GBs–TBs per job); supports incremental checkpoints

```
SET 'execution.checkpointing.interval' = '60s';
SET 'execution.checkpointing.mode' = 'EXACTLY_ONCE';
SET 'state.backend.type' = 'rocksdb';
SET 'state.backend.incremental' = 'true';
SET 'state.checkpoints.dir' = 's3://my-
bucket/flink/checkpoints';
SET 'table.exec.state.ttl' = '1 d';           -- expire idle
state from regular joins and aggregations
```

How checkpoints work: the JobManager injects barriers into the sources; as barriers flow through the job, each operator snapshots its state asynchronously. When every operator has acknowledged, the checkpoint (state + source offsets) is complete. After a failure, Flink restores the latest checkpoint and replays from the recorded offsets.

Savepoints are user-triggered snapshots used to stop and resume a job across code changes, upgrades, and rescaling. Assign stable operator IDs (`uid`) in the DataStream API so state can be mapped after changes.

Exactly-Once End to End

Flink guarantees exactly-once *state*. End-to-end exactly-once also needs:

1. **Replayable sources** — e.g. Kafka offsets stored in checkpoints
2. **Transactional or idempotent sinks:**
3. Kafka sink with `exactly-once` delivery (transactions committed on checkpoint completion; consumers read with `isolation.level=read_committed`)
4. Lakehouse sinks (Iceberg, Delta, Paimon) that commit data files on checkpoints
5. Upserts keyed by a primary key into databases (idempotent)

Output only becomes visible when a checkpoint completes, so the checkpoint interval sets a lower bound on end-to-end latency for transactional sinks.

CDC and Lakehouse Pipelines

```
-- Flink CDC source (the Postgres CDC connector reads the
WAL directly)
CREATE TABLE customers_cdc (
  customer_id INT,
  email       STRING,
  country     STRING,
  PRIMARY KEY (customer_id) NOT ENFORCED
) WITH (
  'connector' = 'postgres-cdc',
  'hostname'  = 'app-db.internal',
  'port'      = '5432',
  'username'  = 'flink_cdc',
  'password'  = '<injected at deploy time>', -- e.g.
templated from a secrets manager; never commit
  'database-name' = 'app',
  'schema-name'   = 'public',
  'table-name'    = 'customers',
  'slot.name'     = 'flink_customers'
);

-- Continuously mirror into a lakehouse table (catalog
configured separately)
INSERT INTO lakehouse.silver.customers
SELECT customer_id, email, country FROM customers_cdc;
```

Common patterns: CDC → Flink → Iceberg/Paimon/Delta tables (streaming lakehouse) · Kafka → Flink → enriched Kafka topics for downstream services · Kafka → Flink → real-time aggregates in an OLAP store.

Deployment and Operations

Option	Notes
Kubernetes with the Flink Kubernetes Operator	Declarative <code>FlinkDeployment</code> resources; savepoint-based upgrades
Managed services	Amazon Managed Service for Apache Flink, Confluent Cloud for Apache Flink, Ververica, and others
YARN / standalone	Existing Hadoop or VM-based estates

What to monitor: checkpoint duration and failures · backpressure (operators that can't keep up) · consumer lag on sources · state size growth · watermark lag (event time vs wall-clock time) · restarts

Scaling: parallelism up to the number of source partitions for Kafka sources; rescale via savepoint (or reactive/adaptive scheduling); increase TaskManager memory for large RocksDB state.

Common Pitfalls

Pitfall	Symptom	Fix
Processing time used for business metrics	Results change when the job lags or replays	Event time with watermarks
Watermark delay too small	Many late events dropped; incomplete windows	Size the delay from measured lateness; handle late data explicitly
An idle partition	Windows never close; no output	Configure source idle timeout
Unbounded state in regular joins or <code>GROUP BY</code> without windows	State grows until the job fails	Windowed or interval joins; state <u>TTL</u>

Pitfall	Symptom	Fix
Checkpoints too frequent or too large	Backpressure, checkpoint timeouts	Incremental RocksDB checkpoints; tune interval and timeout; reduce state
Sink doesn't support updates	Errors or duplicates for changelog queries	Upsert-capable sinks with a primary key, or append-only query shapes
Changing a job without savepoints or stable UIDs	State lost on upgrade	Stop with savepoint; stable operator IDs
Python UDFs on hot paths	Low throughput	Built-in functions or Java UDFs
Forgetting the source's retention	Can't recover from an old checkpoint after long downtime	Kafka retention longer than the maximum recovery window

Cheat Sheet

Task	Flink SQL
Kafka source with watermark	<code>WATERMARK FOR ts AS ts - INTERVAL '10' SECOND + 'connector' = 'kafka'</code>
Tumbling window	<code>FROM TABLE(TUMBLE(TABLE t, DESCRIPTOR(ts), INTERVAL '5' MINUTES)) GROUP BY window_start, window_end, ...</code>
Session window	<code>SESSION(TABLE t PARTITION BY k, DESCRIPTOR(ts), INTERVAL '30' MINUTES)</code>
Upsert sink	<code>PRIMARY KEY (...) NOT ENFORCED + 'connector' = 'upsert-kafka'</code>
Point-in-time enrichment	<code>JOIN rates FOR SYSTEM_TIME AS OF o.ts AS r</code>
Bound state	<code>SET 'table.exec.state.ttl' = '1 d'</code>
Checkpointing	<code>SET 'execution.checkpointing.interval' = '60s'</code>
Deduplicate	<code>ROW_NUMBER() OVER (PARTITION BY id ORDER BY proc_time) = 1</code>

Flink vs Spark Structured Streaming vs Kafka Streams: Flink — true streaming, very large state, rich event-time semantics, SQL · Spark — micro-batch, one engine for batch and streaming, strong lakehouse integration · Kafka Streams — a library for Kafka-in/Kafka-out JVM applications, no separate cluster

Interview Questions

Q: How does Flink achieve fault tolerance with exactly-once state? A: Through asynchronous, consistent checkpoints based on barriers. The JobManager injects checkpoint barriers into the sources; each operator snapshots its state when barriers from all its inputs arrive, without stopping processing. A completed checkpoint contains all operator state plus the source positions. On failure, Flink restores the last checkpoint and replays sources from the saved offsets, so every event affects state exactly once.

Q: What are watermarks and why are they needed? A: A watermark is a timestamp flowing through the stream that declares that no events older than it are expected. Event-time windows can only produce results once the watermark passes their end — without watermarks, the system could never know whether more data for a window might still arrive. The watermark delay balances latency against completeness for out-of-order data.

Q: How do you keep state from growing without bound? A: Use time-bounded operations — windowed aggregations, interval joins, and window joins — whose state can be discarded once the watermark passes. For regular joins and non-windowed aggregations, set a state TTL so keys that haven't been updated expire. Monitor state size per operator, and choose RocksDB for large state.

Q: When would you choose Flink over Spark Structured Streaming? A: When you need very low latency (per-event processing rather than micro-batches), large keyed state, complex event-time logic (session windows, temporal joins, custom timers), or event-driven applications. Spark is often the better fit when the team and platform are already built on Spark, when second-level micro-batch latency is enough, and when one engine for batch and streaming on a lakehouse matters more.

Q: What does end-to-end exactly-once require besides Flink's checkpoints? A: A replayable source whose position is stored in the checkpoint, and a sink that either participates in the checkpoint via transactions (two-phase commit — e.g. the Kafka sink or lakehouse table sinks that commit on checkpoint completion) or writes idempotently (upserts by primary key). Downstream Kafka consumers must read committed data only.

Further Reading

- [Apache Flink documentation](https://nightlies.apache.org/flink/flink-docs-stable/) (https://nightlies.apache.org/flink/flink-docs-stable/)

- [Flink SQL: windowing table-valued functions](https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/table/sql/queries/window-tvf/) (https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/table/sql/queries/window-tvf/) (pinned to 1.19 — Flink's "stable" alias reorganized this path in a later release)
 - [Flink CDC](https://nightlies.apache.org/flink/flink-cdc-docs-stable/) (https://nightlies.apache.org/flink/flink-cdc-docs-stable/)
 - [Flink Kubernetes Operator](https://nightlies.apache.org/flink/flink-kubernetes-operator-docs-stable/) (https://nightlies.apache.org/flink/flink-kubernetes-operator-docs-stable/)
 - *Stream Processing with Apache Flink* — Fabian Hueske & Vasiliki Kalavri (O'Reilly)
 - *Streaming Systems* — Tyler Akidau, Slava Chernyak & Reuven Lax (O'Reilly)
-

Previous: [Kafka](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) • **Next:** [Beam and Dataflow](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/beam-dataflow/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/beam-dataflow/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026