

Data Ingestion & Change Data Capture

Moving data reliably from source systems into the platform — APIs, files, databases, and change streams.

Last reviewed 29 Sep 2026 · Lab-tested with Debezium 3.6.3.Final ·

Prerequisites: [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) · [Python for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) · [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/)

Related: [Lab 10](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/10-cdc-debezium) · [Kafka](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) · [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) · [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) · [Airflow](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) · [NoSQL and Operational Stores](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/nosql-operational-stores/) · [Streaming SQL](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/streaming-sql/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

Challenge: Source systems were built to run the business, not to feed analytics. Their data sits behind rate-limited APIs, in operational databases that must not be slowed down, and in files that arrive late, twice, or with a different schema than last week. Every downstream table is only as good as the ingestion layer that feeds it.

Solution: A deliberate ingestion design chooses, per source, *how* data is captured (full extract, incremental extract, change data capture, or event streaming), *where* it lands (an immutable raw layer), and *how* it is applied to downstream tables (append, overwrite, or merge) — with idempotency, schema handling, and monitoring built in.

Source	Capture method
Landing (raw / bronze)	Apply

```

SaaS / REST APIs      → incremental pull (cursor) →
files or tables, append-only → MERGE / overwrite
Files (SFTP, buckets) → event-driven pickup      →
partitioned by arrival date → append / overwrite
OLTP databases        → CDC from the transaction log →
change events (Kafka / files) → MERGE (upsert + delete)
Applications          → event streaming          →
topics → raw tables          → append

```

Relevance to data engineering: ingestion is where most production incidents start — missed records, duplicates, silent schema changes, and source outages. Getting it right makes every downstream layer simpler.



On this page

Basic - [Ingestion Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#ingestion-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#ingestion-patterns) - [Full vs Incremental Loads](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#full-vs-incremental-loads) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#full-vs-incremental-loads) - [The Raw Landing Layer](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#the-raw-landing-layer) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#the-raw-landing-layer)

Intermediate - [Ingesting from APIs](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#ingesting-from-apis) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#ingesting-from-apis) - [Ingesting Files](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#ingesting-files) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#ingesting-files) - [Extracting from Databases](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#extracting-from-databases) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#extracting-from-databases)

Advanced - [Change Data Capture \(CDC\)](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#change-data-capture-cdc) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#change-data-capture-cdc) - [CDC with Debezium](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#cdc-with-debezium) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#cdc-with-debezium) - [Applying Changes with MERGE](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#applying-changes-with-merge) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#applying-changes-with-merge) - [Schema Evolution](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#schema-evolution) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#schema-evolution) - [Build vs Buy](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#build-vs-buy) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#build-vs-buy)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/#further-reading)

Ingestion Patterns

Pattern	How it works	Latency	Captures deletes?	Load on source	Typical sources
Full extract	Copy the whole dataset every run	Hours	Yes (by comparison)	High	Small reference tables
Incremental extract	Copy rows changed since the last watermark	Minutes–hours	No	Medium	Tables with a reliable <code>updated_at</code>
Change data capture	Read inserts, updates, and deletes from the database log	Seconds	Yes	Low	OLTP databases
Event streaming	Applications publish events as they happen	Seconds	N/A (events)	None	Clickstream, IoT, microservices
File drop	Partners or systems deliver files to a location	Batch	Depends on content	None	Partner feeds, exports
API pull	Call a service's API on a schedule	Minutes–hours	Rarely	Rate-limited	SaaS tools

Choosing: use CDC for operational databases when you need deletes or low latency; incremental extracts when CDC isn't available and the source has a trustworthy change column; full extracts only for small tables; event streaming when you own the producing application.

Full vs Incremental Loads



```

from datetime import datetime, timezone

def extract_incremental(conn, table: str, watermark_store) ->
list[dict]:
    """Pull rows changed since the last successful run, with
    an overlap window."""
    last = watermark_store.get(table) or datetime(1970, 1, 1,
    tzinfo=timezone.utc)
    lookback = last.replace(microsecond=0) # optionally
    subtract minutes to catch late commits

    rows = conn.execute(
        f"SELECT * FROM {table} WHERE updated_at >= %s ORDER
    BY updated_at",
        (lookback,),
    ).fetchall()

    if rows:
        write_to_raw(table, rows)
# append-only landing
        watermark_store.set(table, max(r["updated_at"] for r
    in rows)) # only after a successful write
    return rows

```

Rules for incremental loads - Use `>=` with a small overlap (plus deduplication downstream) rather than `>` — transactions can commit out of timestamp order - Advance the watermark only after the data is durably written - Store watermarks outside the job (a table or key-value store), keyed by source and table - Schedule periodic full reconciliations: `updated_at` - based extraction never sees hard deletes

The Raw Landing Layer

Land data exactly as received, before any transformation:

```
raw/<source>/<entity>/ingest_date=2024-03-
15/batch_id=01HS9.../part-0000.json.gz
```



└─ one folder

per run: safe to rerun, easy to audit

Principle	Why
Immutable, append-only	You can always reprocess downstream from the original data
Partition by <i>arrival</i> time	Makes reruns and late data easy to reason about
Store the raw payload plus metadata (<code>_ingested_at</code> , <code>_source</code> , <code>_batch_id</code> , <code>_schema_version</code>)	Lineage, debugging, and deduplication
Keep the original format (JSON, CSV) or a lossless conversion	Nothing is lost before you understand it
Lifecycle rules to cheaper storage	Raw data is large and rarely read

Ingesting from APIs

```

import time
import requests
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

def api_session() -> requests.Session:
    s = requests.Session()
    retry = Retry(total=5, backoff_factor=1,
                  status_forcelist=[429, 500, 502, 503, 504],
                  respect_retry_after_header=True)
    s.mount("https://", HTTPAdapter(max_retries=retry))
    return s

def fetch_changed_records(base_url: str, token: str,
                          updated_since: str):
    """Cursor-paginated incremental extract. Yields pages so
    memory stays flat."""
    session = api_session()
    params = {"updated_since": updated_since, "limit": 500}
    while True:
        resp = session.get(f"{base_url}/v1/invoices",
                           params=params,
                           headers={"Authorization": f"Bearer {token}"},
                           timeout=30)

```

```

resp.raise_for_status()
body = resp.json()
yield body["data"]
cursor = body.get("next_cursor")
if not cursor:
    break
params["cursor"] = cursor
time.sleep(0.2) # stay well
inside rate limits

```

Concern	Practice
Pagination	Prefer cursor-based over offset-based (offsets skip or repeat rows when data changes mid-extract)
Rate limits	Honour <code>Retry-After</code> ; throttle proactively; spread large backfills over time
Incremental cursor	Use the API's <code>updated_since</code> / change-feed endpoint; persist the high-water mark
Authentication	Tokens from a secrets manager; handle OAuth refresh
Deletes	Look for a deleted/archived flag or events endpoint; otherwise reconcile periodically
Idempotency	Write each run to its own batch folder; deduplicate by record ID downstream

Ingesting Files

```

Partner drops file → object storage event → queue → loader
├─
validate: name, size, checksum, schema
├─ copy
to raw/<source>/ingest_date=.../
├─
record in a load manifest (file, checksum, rows, status)
└─ on
failure → quarantine/ + alert

```

- **Detect files with events, not listings:** storage notifications (to a queue) scale better than listing millions of keys

- **Keep a manifest** of processed files (name + checksum) so the same file is never loaded twice and a re-delivered file is detected
 - **Wait for completeness:** use `_SUCCESS` / manifest files or a size-stable check — partners often upload in pieces
 - **Validate early:** reject files with unexpected columns, encodings, or row counts into a quarantine area
-

Extracting from Databases

```
# Parallel JDBC read in Spark: split the table into ranges of
a numeric key
df = (spark.read.format("jdbc")
      .option("url", "jdbc:postgresql://replica-db:5432/app")
      # read from a replica, not the primary
      .option("dbtable", "(SELECT * FROM orders WHERE
updated_at >= '2024-03-15') AS src")
      .option("user", user).option("password", password)
      .option("partitionColumn", "order_id")
      .option("lowerBound", 1).option("upperBound",
50_000_000)
      .option("numPartitions", 16)
      # 16 concurrent queries – check the source can take it
      .option("fetchsize", 10_000)
      .load())
```

- Read from a **read replica** or snapshot, never the primary under load
 - Get a **consistent snapshot** for multi-table extracts (same transaction or snapshot timestamp)
 - Limit parallelism to what the source can handle; agree limits with the database owners
 - For large or frequently changing tables, move to **CDC** instead of repeated extracts
-

Change Data Capture (CDC)

CDC captures every insert, update, and delete from a source database as an ordered stream of change events.

Approach	How	Deletes	Source impact	Notes
Log-based	Read the transaction log (Postgres <code>WAL</code> , MySQL binlog, SQL Server CDC tables, Oracle redo)	Yes	Minimal	The standard approach (Debezium, cloud DMS, managed connectors)
Query-based	Poll with <code>WHERE updated_at > watermark</code>	No	Query load	Simple; misses deletes and intermediate states
Trigger-based	Triggers write changes to an audit table	Yes	Write overhead on every transaction	Legacy; avoid on busy databases

What log-based CDC gives you: every change in commit order, including deletes; before-and-after row images; the log position (`LSN/offset`) for exactly-once bookkeeping; and minimal load on the source.

CDC with Debezium

Debezium is an open-source CDC platform that runs as Kafka Connect source connectors (or embedded / as Debezium Server). Each captured table becomes a Kafka topic.

Postgres prerequisites: `wal_level = logical`, a user with replication privileges, and a publication for the captured tables.

```
{
  "name": "orders-postgres-cdc",
  "config": {
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "plugin.name": "pgoutput",
    "database.hostname": "app-db.internal",
    "database.port": "5432",
    "database.user": "debezium",
    "database.password":
"${file:/secrets/db.properties:password}",
    "database.dbname": "app",
    "topic.prefix": "app",
    "table.include.list": "public.orders,public.customers",
    "slot.name": "debezium_orders",
```

```

    "publication.autocreate.mode": "filtered",
    "snapshot.mode": "initial"
  }
}

```

Change event (simplified):

```

{
  "before": {"order_id": 42, "status": "placed", "amount":
99.50},
  "after": {"order_id": 42, "status": "shipped", "amount":
99.50},
  "source": {"table": "orders", "lsn": 24023128, "ts_ms":
1710496931000},
  "op": "u",
  "ts_ms": 1710496931512
}

```

op	Meaning	before	after
r	Snapshot read (initial load)	null	row
c	Insert	null	row
u	Update	old row (Postgres: null unless the table has <code>REPLICA IDENTITY FULL</code>)	new row
d	Delete	old row (Postgres by default: only the key, with placeholder values in the other columns)	null

Deletes and tombstones: each delete event is followed by a *tombstone*, a message with the same key and no value, so Kafka can compact the key away (`tombstones.on.delete=false` turns it off). A consumer must skip it. On Postgres, do not read non-key columns from a delete's `before` unless the table has `REPLICA IDENTITY FULL`.

Snapshot then stream: on first start the connector takes a consistent snapshot of existing rows (`op = r`), then continues from the exact log position where the snapshot ended — no gap and no overlap.

Operational essentials: monitor replication slot lag (an unconsumed slot makes the source database retain WAL and can fill its disk) · use Avro/Protobuf with a schema registry · route poison events to a dead-letter topic · plan for connector restarts (offsets are stored in Kafka)

Applying Changes with MERGE

Change events must be applied to a target table so it mirrors the source. The key steps: keep only the **latest change per key** within the batch (ordered by log position), then **merge** — inserting, updating, or deleting.

```
MERGE INTO silver.orders AS t
USING (
  SELECT *
  FROM (
    SELECT c.*,
           ROW_NUMBER() OVER (PARTITION BY order_id ORDER
BY source_lsn DESC) AS rn
    FROM   bronze.orders_changes c
    WHERE  batch_id = :batch_id
  ) latest
  WHERE rn = 1 -- last
change per key wins
) AS s
ON t.order_id = s.order_id
WHEN MATCHED AND s.op = 'd' THEN DELETE
WHEN MATCHED AND s.source_lsn > t.source_lsn THEN -- ignore
stale/replayed events
  UPDATE SET status = s.status, amount = s.amount,
updated_at = s.updated_at, source_lsn = s.source_lsn
WHEN NOT MATCHED AND s.op <> 'd' THEN
  INSERT (order_id, status, amount, updated_at, source_lsn)
  VALUES (s.order_id, s.status, s.amount, s.updated_at,
s.source_lsn);
```

Why each piece matters - *Deduplicate by log position*: one batch may contain several changes to the same row; only the last one counts - *Compare source_lsn*: replays and out-of-order batches can't overwrite newer data — this makes the apply idempotent - *Handle deletes explicitly*: or choose soft deletes (`is_deleted = true`) when downstream needs history. A hard delete forgets the log position, so a replayed older insert can bring the row back; a marked row keeps the position that protects it - *Keep the change log too*: an append-only history table of all events supports audits and SCD Type 2

Schema Evolution

Source change	Safe handling
New nullable column	Add it downstream automatically (table formats support schema evolution)
Column dropped	Keep the column downstream (NULL going forward); alert
Column renamed	Treat as drop + add; requires a mapping decision — alert and block
Type widened (int → bigint)	Usually safe to evolve
Type narrowed or changed	Block and alert; needs a migration plan

- Register schemas in a schema registry and set a **compatibility mode** for event topics
- Store the schema version with each raw record
- Agree on **data contracts** with source owners so breaking changes are announced, versioned, and tested before release

Build vs Buy

Option	Examples	Strengths	Trade-offs
Managed ELT connectors	Fivetran, Airbyte Cloud, Stitch, cloud-native connectors	Hundreds of sources, schema handling, no ops	Per-row/usage pricing; less control over edge cases
Open-source connectors, self-hosted	Airbyte OSS, Meltano, dlt, Debezium	Control, no per-row fees	You run and upgrade them
Cloud database migration/ <u>CDC</u> services	AWS DMS, GCP Datastream, Azure Data Factory <u>CDC</u>	Integrated with the cloud; managed <u>CDC</u>	Cloud-specific
Custom code	Python + orchestrator	Full control for unusual sources	You own reliability, retries, schema drift

Rule of thumb: buy (or use open-source connectors) for common SaaS sources; use managed or Debezium-based CDC for databases; write custom code only for unusual or high-value sources where the control is worth the maintenance.

Common Pitfalls

Pitfall	Symptom	Fix
Watermark on <code>updated_at</code> with <code>></code>	Rows committed out of order are skipped	<code>>=</code> plus a lookback window, and deduplicate downstream
Advancing the watermark before the write succeeds	Data permanently skipped after a failed run	Persist the watermark only after a durable write
Incremental extracts assumed to capture deletes	Deleted records live forever downstream	<u>CDC</u> , a soft-delete flag, or periodic full reconciliation
Offset-based API pagination during an extract	Duplicated or missing records	Cursor-based pagination or a stable sort key
Extracting from the primary database	Production slowdowns and angry application teams	Read replicas, snapshots, or log-based <u>CDC</u>
Abandoned <u>CDC</u> replication slot	Source database disk fills with retained <u>WAL</u>	Monitor slot lag; drop slots when retiring connectors
Applying <u>CDC</u> events without ordering	Older changes overwrite newer ones	Order and compare by log position (<u>LSN</u> /offset)
Loading the same file twice	Duplicates after partner re-deliveries	Manifest of processed files with checksums
Transforming during ingestion	Can't reprocess when logic was wrong	Land raw data unchanged; transform downstream
No freshness or volume monitoring per source	Missing data noticed days later	Alert on last successful load time and row counts per source

Cheat Sheet

Situation	Pattern
Small reference table	Full extract, overwrite
Large table with a reliable <code>updated_at</code> , deletes don't matter	Incremental extract + MERGE

Situation	Pattern
Operational database, deletes matter or low latency needed	Log-based <u>CDC</u> + MERGE
Own application producing events	Publish events to a stream, land append-only
Partner files	Event-driven pickup + manifest + quarantine
SaaS tool	Managed or open-source connector

Raw record metadata: `_source` · `_ingested_at` · `_batch_id` · `_source_file` or `_source_lsn` · `_schema_version`

Debezium op codes: `r` snapshot · `c` insert · `u` update · `d` delete

Idempotency toolkit: batch folders per run · deterministic record IDs · `MERGE` on keys · compare log positions · manifests for files

Interview Questions

Q: What is change data capture and why is log-based CDC preferred? A: CDC captures row-level inserts, updates, and deletes from a source database as a stream of change events. Log-based CDC reads the database's transaction log (the Postgres WAL, MySQL binlog) instead of querying tables, so it captures every change including deletes, preserves commit order, provides before-and-after images, and puts almost no load on the source. Query-based approaches miss deletes and intermediate states, and trigger-based approaches slow down every write.

Q: How do you design an incremental load that doesn't lose or duplicate data? A: Use a reliable change column or log position as a watermark, extract with an overlap (`>=` and a lookback window) to catch late commits, write each run to an immutable batch location, and advance the watermark only after the write succeeds. Downstream, deduplicate by primary key and apply with `MERGE` so replays are harmless. Add periodic reconciliation (row counts or checksums against the source) to catch anything the incremental logic misses, such as hard deletes.

Q: How would you replicate an operational Postgres database into a lakehouse in near real time? A: Enable logical replication, run a Debezium Postgres connector that snapshots existing rows and then streams changes to Kafka topics (one per table) with Avro schemas. A streaming job lands the change events append-only in a bronze table, then applies them to

silver tables with `MERGE` — the latest change per key by `LSN`, deletes applied explicitly, and `LSN` comparison to ignore stale events. Monitor replication slot lag, connector health, and end-to-end freshness, and keep the change history for audits.

Q: How do you handle schema changes from a source you don't control? A: Detect them at ingestion by comparing each batch's schema with the registered one. Let additive changes (new nullable columns) evolve automatically, but block and alert on breaking changes (renames, type narrowing, dropped required fields) rather than silently loading bad data. Store raw data unchanged so you can reprocess once the mapping is fixed, and establish data contracts with the source owners so changes are communicated in advance.

Q: When would you buy an ingestion tool instead of building one? A: For common SaaS sources and standard databases, managed or open-source connectors are usually cheaper than engineering time: they handle pagination, rate limits, schema changes, and API updates across hundreds of sources. Build custom ingestion when the source is unusual, when volume makes per-row pricing prohibitive, or when you need control over latency, security, or data handling that the tools can't provide.

Further Reading

- [Debezium documentation](https://debezium.io/documentation/) (<https://debezium.io/documentation/>) — connectors, event format, and operations
- [Kafka Connect documentation](https://kafka.apache.org/documentation/#connect) (<https://kafka.apache.org/documentation/#connect>)
- [dlt \(data load tool\)](https://dlthub.com/docs/intro) (<https://dlthub.com/docs/intro>) — open-source Python library for API and database ingestion
- [Airbyte documentation](https://docs.airbyte.com/) (<https://docs.airbyte.com/>)
- *Designing Data-Intensive Applications* — Martin Kleppmann (chapters on replication, change data capture, and stream processing)
- [Kafka](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/>) · [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) (<https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/>)

Previous: [Streaming SQL](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/streaming-sql/) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/streaming-sql/>) · **Next:** [Data Engineering System Design](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/) (<https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/>) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (<https://sarangambekar1997.github.io/data-engineering-handbook/>)

 September 29, 2026