Engineer

Change data capture

reading the changes made in a database and passing them on

What it is

Change data capture, usually shortened to CDC, means reading the changes made in a database and passing them on to other systems as they happen. Every insert, update and delete becomes an event that something else can act on: a reporting copy that stays current, a search index, another service that needs to know an order was placed. The database where the change happens stays the system of record, and the other systems follow it.

There are two broad ways to find the changes. The simpler one is to ask: run a query every few minutes for rows whose “last updated” timestamp is newer than the previous run. The other is to read the database’s own record of what it changed. PostgreSQL keeps that record in its write-ahead log (WAL), and its documentation states the rule behind it: changes to data files are written only after the records describing them have been flushed to permanent storage. Because the log already holds every change in the order it happened, reading it gives a complete history without querying the tables at all. This second approach, log-based CDC, is what the term usually means today.

In PostgreSQL the mechanism is called logical decoding, which the documentation defines as “extracting all persistent changes to a database’s tables” into a format that can be read without knowing the database’s internals. Three parts work together. The server has to write enough detail into the WAL, which the `wal_level` setting controls: its default, `replica`, is enough for copying data between servers, and `logical` adds what decoding needs. A replication slot represents one stream of changes from one database, delivered to its reader in the original sequence, and it remembers how far that client has read. An output plugin turns the WAL’s internal records into whatever format the client wants.

Most teams do not write the client themselves. Debezium supplies Apache Kafka Connect with connectors that obtain changes through each source database's CDC mechanism. Its PostgreSQL connector reads through `pgoutput`, which its documentation describes as PostgreSQL's built-in output plugin, or through `decoderbufs`, a plugin maintained by the Debezium community. Debezium can also run as a library inside a JVM application, or as Debezium Server, which sends change events to systems such as Amazon Kinesis, Google Cloud Pub/Sub and Apache Pulsar.

Use the concept in a decision

Put "What is change data capture?" to work in a real decision. We can help relate the concept to the system you are building or buying. Numbers arriving where they are needed, on a schedule, without anyone copying a spreadsheet at the end of the month.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

Polling looks cheaper, and for a small table that changes slowly it often is. Debezium's documentation lists what reading the log does that polling cannot. It captures every change, where a query sees only the state of a row at the moment it runs, so a row that changed three times between two runs shows up once. It captures deletes, where a deleted row is simply missing from the next result. It needs no extra column in the data model, such as a "Last Updated" field that every write path has to remember to set. It can carry the old state of a row and metadata such as the ID of the transaction that made the change. And it delivers changes with very little delay without the CPU cost of frequent polling: for PostgreSQL and MySQL, Debezium puts that delay in the millisecond range.

The second reason is agreement between a database and the messages other systems receive. Debezium names dual writes as the other approach it compares itself with: the application saves a change to its database and then sends a message about it. If the process stops between those two steps, the database and the message stream disagree, and nothing records which one is wrong. CDC removes the second write. The application commits once, and the message is derived from what was committed. The outbox pattern goes a step further: the application writes the message itself into an outbox table in the same

transaction as the change, and CDC publishes the rows of that table. Debezium's Outbox Event Router describes the result as avoiding inconsistencies between a service's own state and the events that other services consume.

The third reason is order. A replication slot preserves the source server's sequence when it delivers changes, grouped by the transaction that made them, as the example below shows. A consumer that applies them in that order rebuilds the same state, step by step. For an ETL load this makes incremental loading honest: a nightly job can take exactly the changes since its last run, deletes included, instead of reloading whole tables or trusting a timestamp column that one code path forgot to update.

What it looks like in practice

The smallest working example uses PostgreSQL alone. It needs `wal_level = logical` in the server configuration, which takes effect only after a restart, and a free replication slot; `max_replication_slots` defaults to 10. The session below runs as a superuser, as the documentation's own example does, and uses `test_decoding`, the output plugin from that example, which prints each change as a line of text.

```
-- postgresql.conf: wal_level = logical (takes effect after a restart)
SELECT * FROM pg_create_logical_replication_slot('shop_cdc', 'test_decoding');

BEGIN;
INSERT INTO orders (id, status) VALUES (42, 'pending');
INSERT INTO outbox (id, event) VALUES (7, 'order 42 placed');
COMMIT;

SELECT data FROM pg_logical_slot_peek_changes('shop_cdc', NULL, NULL);
-- BEGIN 10301
-- table public.orders: INSERT: id[integer]:42 status[text]:'pending'
-- table public.outbox: INSERT: id[integer]:7 event[text]:'order 42 placed'
-- COMMIT 10301

SELECT data FROM pg_logical_slot_get_changes('shop_cdc', NULL, NULL);
-- the same four rows; this call consumes them, so a second call returns none

SELECT pg_drop_replication_slot('shop_cdc');
```

The order and its outbox message come out together, inside the transaction that wrote them. `pg_logical_slot_peek_changes` reads without moving the slot, so it returns the same rows on the next call, while `pg_logical_slot_get_changes`

consumes them. A `CREATE TABLE` would produce only an empty `BEGIN` and `COMMIT`, because logical decoding does not carry DDL. The last line matters most: a slot left behind keeps holding on to server resources.

In production, Debezium plays the reader. On first start, its PostgreSQL connector takes a consistent snapshot of the tables it captures and then streams changes from the point where the snapshot was taken. The default `snapshot.mode`, `initial`, takes that snapshot only when no stored offsets exist, so a restart resumes from the last recorded position instead. Each event carries `before` and `after` images of the row, a `source` block, a timestamp and an `op` code: `c` for create, `u` for update, `d` for delete, `r` for a row read during a snapshot and `t` for truncate. After a delete, the connector also emits a tombstone, an event with the same key and a null value, so that Kafka's log compaction can remove the earlier messages for that key.

Both patterns from this article show up in the company's own pages. Between 2022 and 2024, the founder engineered 600 ETL/ELT pipelines in PL/pgSQL across several PostgreSQL development and production environments, including incremental loads using CDC and timestamp-based deltas. The note Selling the last ticket exactly once designs a flash sale around the outbox: every step records its message in the outbox within the same transaction as the change, change data capture publishes the outbox, and because a message can arrive more than once, every consumer is idempotent, keyed by the order.

Where it goes wrong

Nothing caps what a slot may keep. A slot holds on to every piece of WAL its consumer has not confirmed, and PostgreSQL's defaults set no ceiling on that. `max_slot_wal_keep_size` is `-1`, which lets slots retain an unlimited amount of WAL, and `idle_replication_slot_timeout` is `0`, which switches off the invalidation of slots that sit unused. A connector that stopped during a failed deploy can therefore fill the disk long after everyone has forgotten it. Watch `pg_replication_slots` for slots that are no longer active, and set `max_slot_wal_keep_size` to what the disk can afford, knowing that a consumer which falls further behind than that may no longer be able to continue. Recovering from that means a new slot, and a new slot starts from the present: Debezium warns that a dropped and recreated slot can silently skip the events in between, so take a fresh snapshot afterwards.

A quiet table holds the log back. Debezium's documentation describes a subtler version. A slot belongs to one database, and the connector advances its confirmed position only when it sees changes it captures. If the captured tables rarely change while other tables, or other databases on the same server, write heavily, the position barely moves and the WAL behind it keeps growing. The fix is a heartbeat: `heartbeat.interval.ms` makes the connector emit heartbeat events at a set interval, and `heartbeat.action.query` lets it write regularly to a small table in the captured database, which has to be part of the publication the connector reads, so the slot advances even when nothing else happens there.

Consumers assume each change arrives once. A logical slot emits each change once in normal operation, but its position is saved only at checkpoints, so after a crash it can return to an earlier point and send recent changes again. PostgreSQL's documentation leaves the consequences to the client. A consumer that adds each amount to a running total counts those changes twice. Make every consumer idempotent: apply each change as an upsert keyed by the row's primary key, or record the IDs of the events already processed, so that a second delivery changes nothing.

Updates and deletes arrive without the old values. What an update or delete event says about the previous row depends on the table's replica identity. With `REPLICA IDENTITY DEFAULT`, Debezium's events carry at most the previous primary-key values, and the connector produces neither update nor delete events when the table has no primary key. `REPLICA IDENTITY FULL` puts every previous column into `before`, at a price: PostgreSQL's documentation notes that it increases WAL volume, particularly when many `UPDATE` and `DELETE` statements run. Give every captured table a primary key, and switch to `FULL` only on the tables whose consumers need the old values.

Further reading

- Debezium Features — what log-based CDC captures that polling and dual writes miss, plus snapshots, filters and masking.
- Debezium connector for PostgreSQL — output plugins, snapshots, the change-event format, replica identity and WAL disk space.
- Logical Decoding Concepts — replication slots, output plugins, and why a slot nobody needs should be dropped.
- Logical Decoding Examples — creating a slot, peeking at and consuming changes, and dropping the slot again.
- Replication (server configuration) — `max_replication_slots`, `max_slot_wal_keep_size` and `idle_replication_slot_timeout`.

- Outbox Event Router — the outbox pattern and how Debezium routes outbox rows to topics.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is change data capture?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/change-data-capture/