CONDUKTOR

Understanding Change Data Capture

Contents

Introduction

Applications start their journey with a minimal data footprint. Initially, a monolithic application backed by an Online Transactional Processing (OLTP) database fulfills every data need of the application.

As applications evolve over time, they have to utilize data stores with different storage formats and access patterns to keep up with the overall user experience. That's not limited to a search index to perform full-text searches, a cache to speed up the reads, and a data warehouse for complex analytics on data. No one database can satisfy all those needs, causing applications to include different data stores in the architecture, storing data in a redundant and denormalized manner.

When applications store the same data in multiple places, it is critical to keep data consistent across the application. Source data systems and their derivations must be kept in sync to avoid potential inconsistencies in the application state. For example, transactions recorded in the operational databases must be reflected in the cache so that users can immediately search for their recent transactions.

Traditionally, batch Extract Transform Load (ETL) pipelines have been used to synchronize source databases with downstream data systems such as caches, search indexes, etc. Those pipelines were scheduled to run at weekly, daily, or hourly intervals, adding more latency to the synchronization.

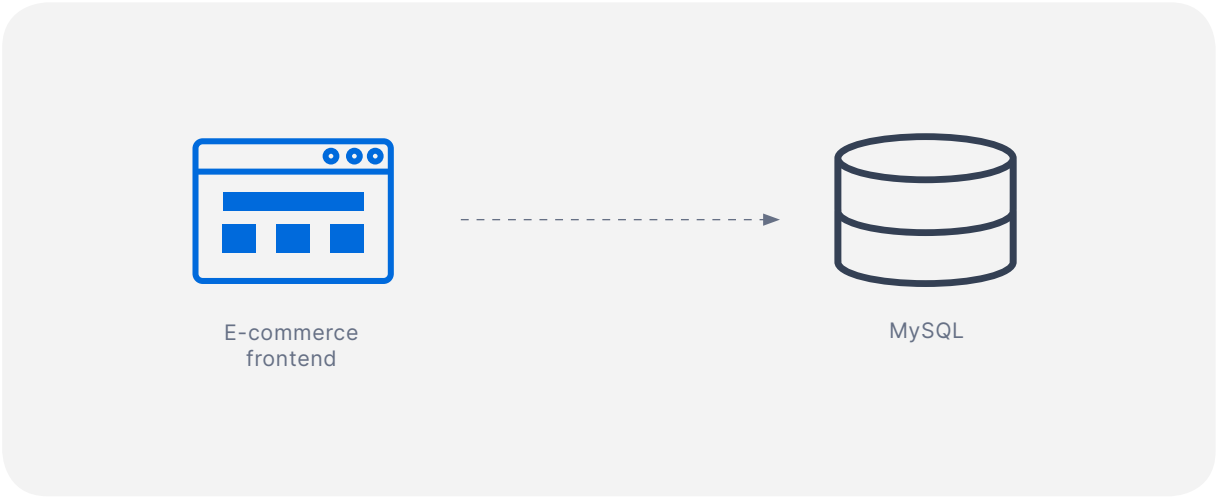
Change Data Capture (CDC) is an alternative approach for batch ETL, enabling real-time data replication across databases. CDC can detect, capture, and move data from a source database as data changes, allowing a real-time data synchronization across downstream systems.

The first few sections of this document walk you through a fictitious online store use case, explain the ETL shortcomings, and introduce you to CDC concepts. Then we discuss how the CDC works, its benefits, and how to implement a CDC system at scale. We conclude the document by discussing potential use cases for CDC, along with popular technology choices to build a CDC system in production environments.

ACME.com
A case in point

Consider ACME.com, a fictitious e-commerce store that started as a monolithic web application backed by a single MySQL database. Initially, the monolith housed the product catalog, search, order management, and payment processing functionalities.

FIGURE 1 • Initially, every site functionality used a single database.



As the business was booming, ACME.com had to support a larger user base, causing it to break apart the monolith and scale-out as a set of Microservices with different database technologies.

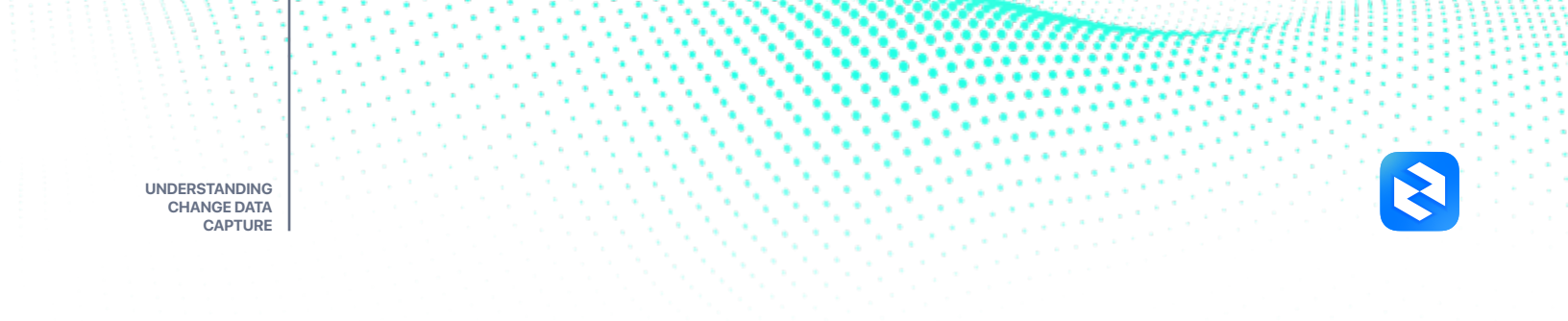
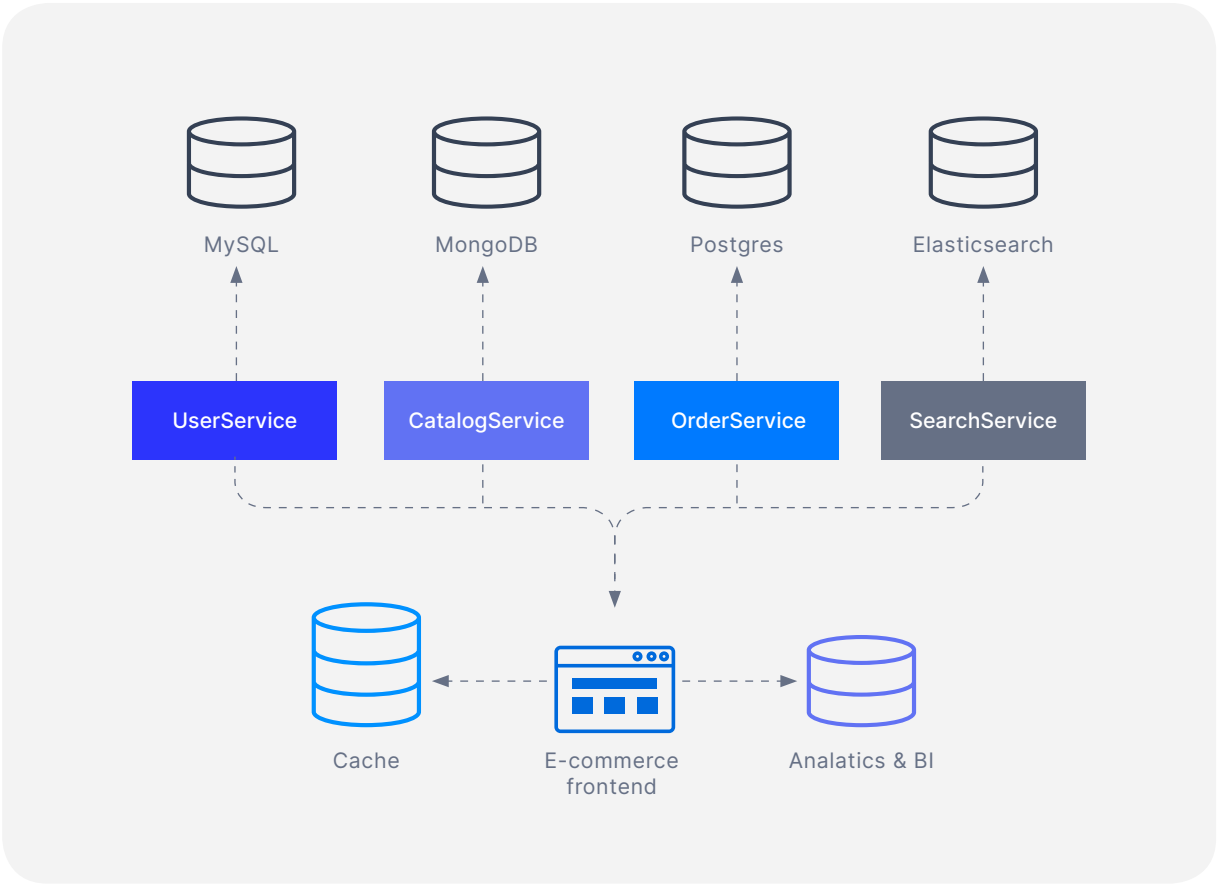


FIGURE 2 • Microservices architecture enforces polyglot persistence.



For example, the new architecture has:

- **UserService** backed by MySQL.
- **CatalogService** backed by MongoDB.
- **OrderService** backed by Postgres.
- **SearchService** backed by Elasticsearch.

Apart from that, a data warehouse and a cache have been added to the architecture.

Data must stay in sync.

Now that several databases exist in the application, the requirement is to find an efficient way to synchronize data among them. That ensures a consistent state across different application components.

For example,

- When the catalog is updated, the relevant Elasticsearch index must be updated.
- The data warehouse must reflect the recent orders received by OrderService.

Systems of records, source data, and derived data systems

When there are multiple versions of the same data set, you need to appoint one as the source of truth or the authoritative version. When there's a discrepancy across versions, the source of truth will always be accepted as the correct one. This version is often called Systems of Records data or source data.

The first time a user creates data, it is captured into Systems of Records. For example, when a customer creates an order, it is received by the OrderService and stored in Postgres.

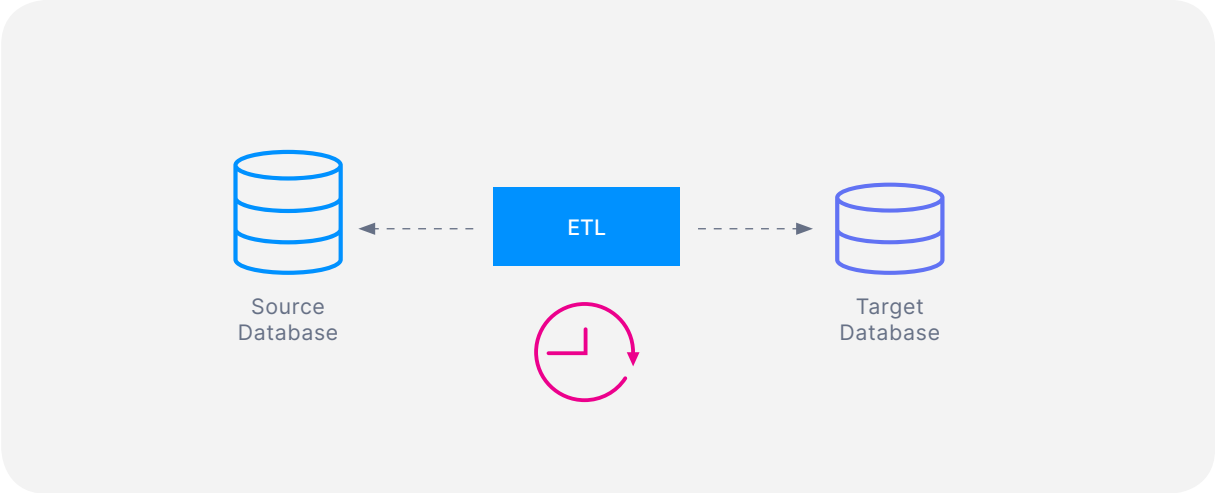
Other systems can take source data, apply transformations, and store their representations to serve different purposes. This data is called Derived Data — which is often redundant and denormalized. If you lose derived data, you can recreate it from the source. We will refer to derived data systems interchangeably as target databases or downstream systems.

For example, orders captured in Postgres are extracted, transformed, and moved into the data warehouse for analytics. The orders in the data warehouse have a different format and an access pattern compared to Postgres.

Batch ETL is slow

Traditionally, ETL jobs have been used to synchronize the source and derived data systems. These jobs run at regular intervals, like daily or hourly, extracting large batches of data from source databases, applying transformations, and loading them into destinations.

FIGURE 3 • ETL jobs run periodically to extract database content in large batches.



ETL jobs had one problem; latency, as it takes minutes to hours to days to move data among different systems. The revamped architecture of ACME.com expects real-time data synchronization across databases, which traditional ETL jobs can't deliver at scale.

A strawman solution to this problem would be to capture the changes made to source databases in real-time and replicate them to downstream derived data systems using a publish-subscribe mechanism.

That paves the path toward (CDC).

Change Data Capture (CDC)

CDC is the process of observing all data changes written to a source database and extracting them in a form in which they can be replicated to derived data systems.

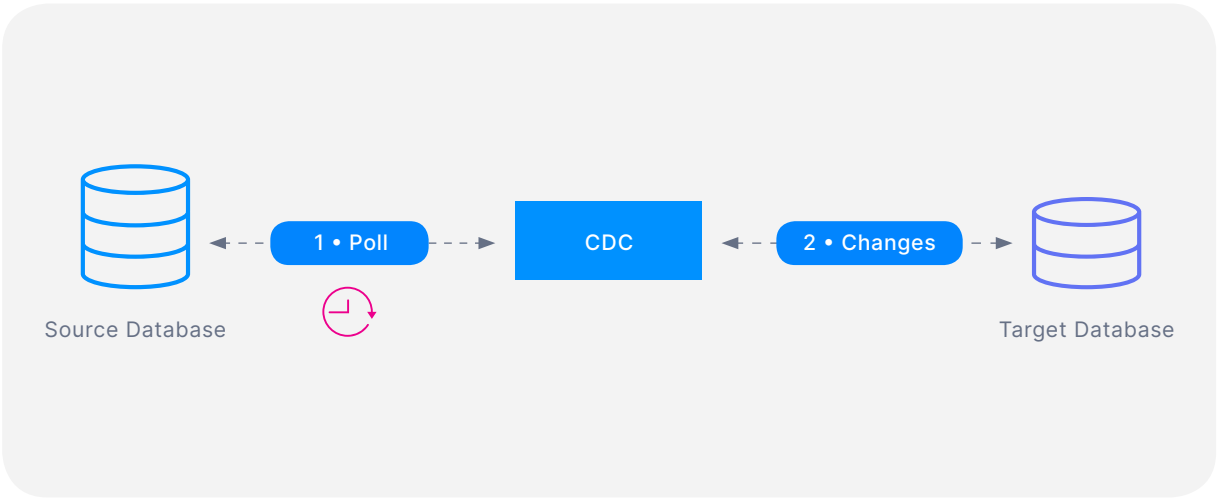
A CDC system is a software application capable of detecting changes from source databases and propagating them to derived data systems. We can categorize CDC systems based on how they detect changes from source databases.

- Pull-based (involves polling)
- Push-based (real-time)

Pull-based CDC systems

A pull-based CDC system works by periodically polling a source database for new changes. It requires the source database tables to add a special column to indicate a change has been made to a particular record. The CDC system then has to watch for the update on the column and fetch the changed record.

FIGURE 4 • Pull-based CDC polls the source database for new changes.



Two types of columns are often added to source tables to flag changes.

Row version

This requires the source database table to add a special column called ‘version.’ Every time a record is updated, the version number column of that record is incremented.

Consider the *users* table in the *UserService* database.

id	first_name	last_name	email	version
43212	John	Doe	jdoe@foo.com	0

A version column is incremented every time a user record is created or updated. The version column tells you how many times it’s changed.

id	first_name	last_name	email	version
43212	John	Doe	jdoe@bar.com	1

LAST_UPDATED timestamp

This method adds a special column LAST_UPATED (the name can be anything) to capture the timestamp that a particular row has been updated for the last time. Every time a record is updated, this column gets overwritten with the current timestamp.

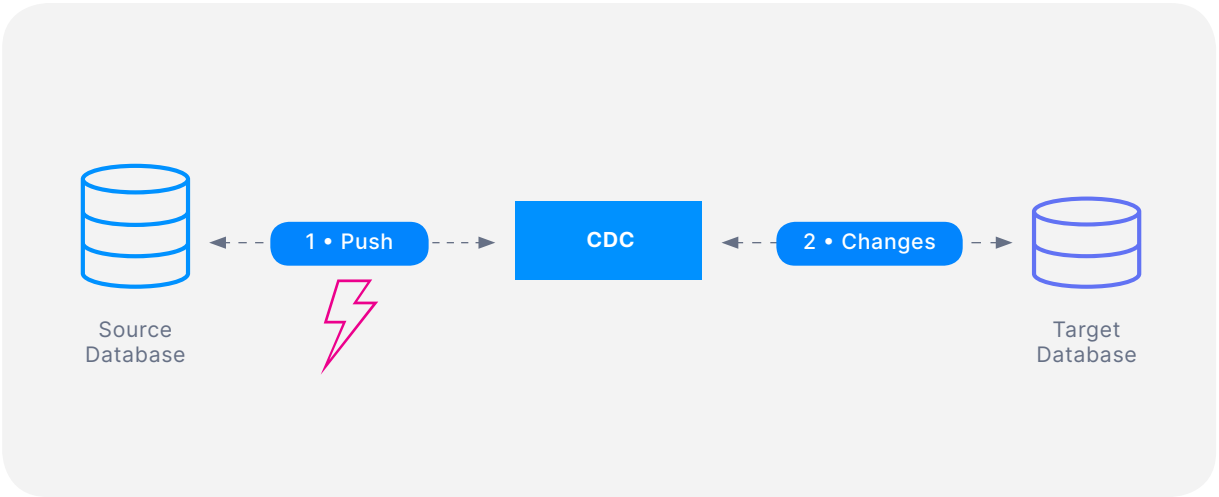
id	first_name	last_name	email	last_updated
43212	John	Doe	jdoe@foo.com	2022-05-25T04:51:07Z

Both of these methods are relatively straightforward to implement. However, they add additional overhead to source databases regarding performance and maintenance. Sometimes, adding change audit columns to all the tables in a database might feel like an invasive move. That'll make an impact on table schemas and long-term maintenance efforts. Moreover, frequent polling adds additional performance overhead to the source database.

Push-based CDC systems

A push-based CDC system detects a change in the source database as it happens and "pushes" it to the CDC system, propagating the change downstream. That doesn't require adding audit columns and polling for new changes.

FIGURE 5 • Change detection happens in real-time with push-based CDC.



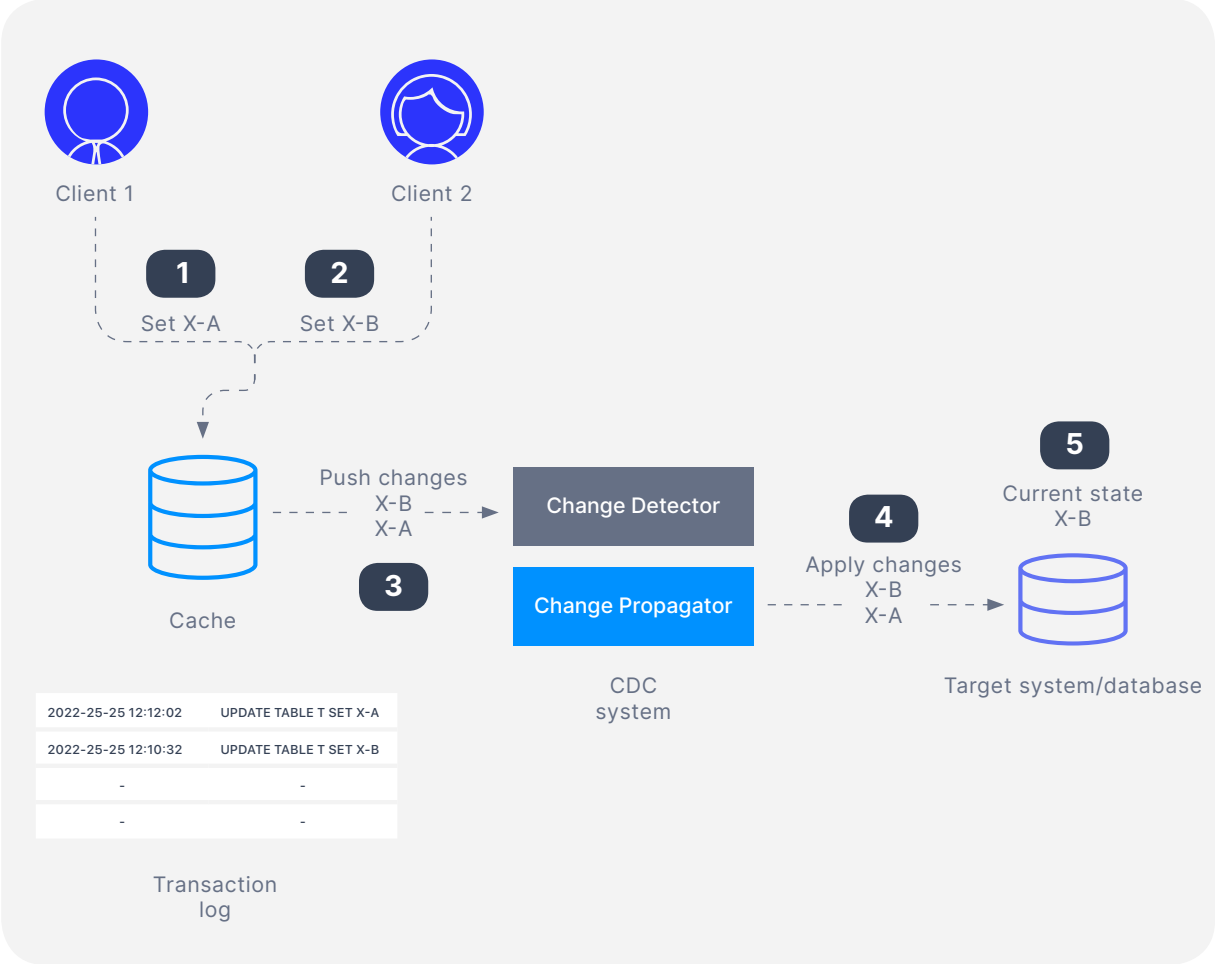
Watching the database transaction log is the primary mechanism for capturing changes applied to a source database table.

Transaction log tailing

When an insert, update or delete operation is applied to a table in the source database, they are recorded in the database's transaction log in the order of their occurrence. The CDC system "tails" this transaction log and propagates them into downstream systems while preserving the change order, allowing them to replay the changes to update their internal state.

Unlike the pull-based methods, there's no polling involved here as the CDC system can detect changes in the source database as they occur. That enables real-time change propagation across downstream data systems, allowing derived data systems to immediately observe changes in the source database.

FIGURE 6 • Architecture of a log-based CDC system.



In the above figure, Client 1 and Client 2 update the value of X in two transactions. The transaction log records the changes. Eventually, the CDC system picks up the changes and delivers them to the destination so that the changes can be replayed to target systems.

Most of the CDC systems we see today are based on a transaction log tailing strategy as it streams changes in real-time and doesn't add performance overhead to the source database.

Designing a production-grade real-time CDC system

CDC systems look elegant in theory. The real challenges come when you try to build them in production. There are a few critical capabilities they must deliver.

A production-grade CDC system should satisfy the following needs.

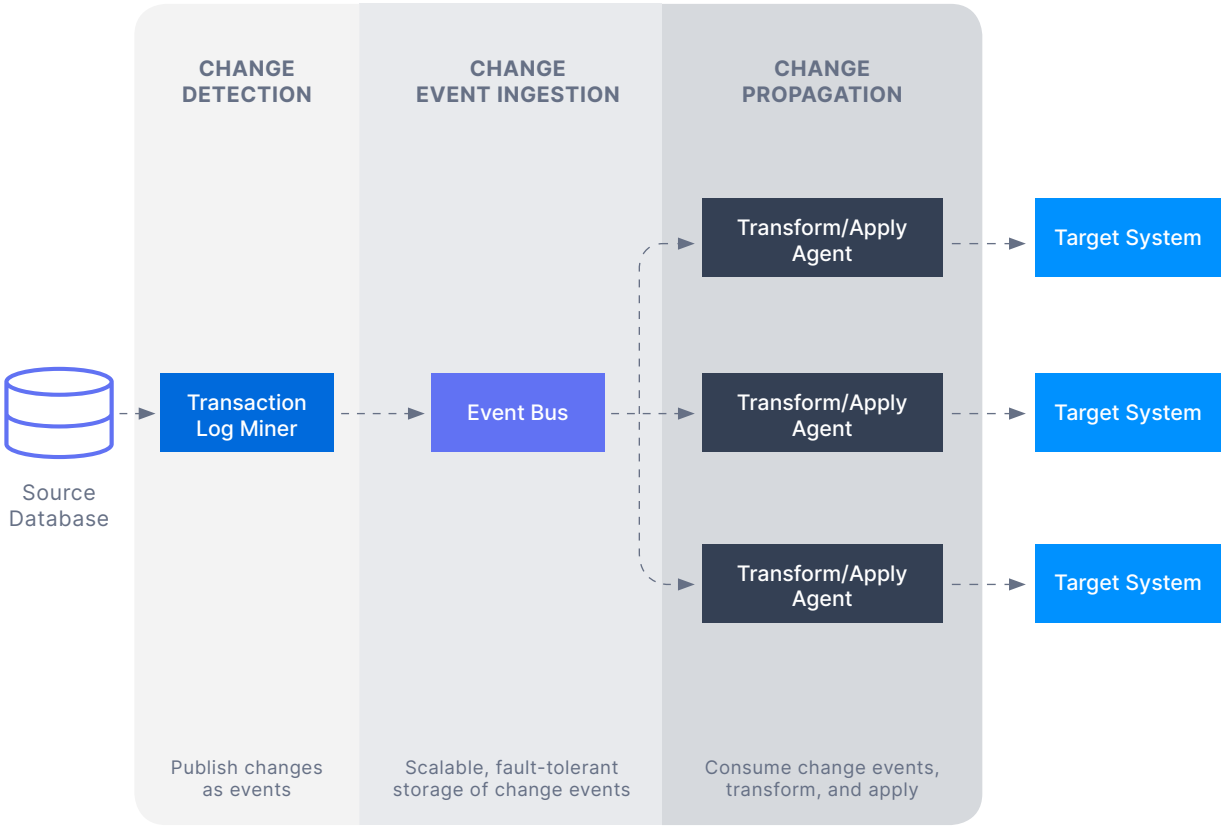
- **Message ordering guarantee** — The order of changes MUST BE preserved so that they are propagated to the target systems as is.
- **Pub/sub** — Should support asynchronous, pub/sub style change propagation to consumers.
- **Reliable and resilient delivery** — At-least-once delivery of changes. Cannot tolerate a message loss.
- **Message transformation support** — Should support light-weight message transformations as the event payload needs to match with the target system's input format.

We can design a CDC system based on the principles of Event-driven Architecture to exploit its asynchronous event-driven nature and loose coupling among application components.

Reference architecture of an event-driven CDC system

The following architecture represents the bare-minimum architecture of a real-time CDC system. Changes in the source databases are detected, captured, and propagated as asynchronous events.

FIGURE 7 • A real-time, event-driven CDC architecture.



Change event detection

The transaction log mining component captures the changes from the source database, converts them into events, and publishes them to the event bus. That happens in real-time while changes are made to the source database.

Each event is timestamped and contains only a single change. The format of a typical change event would look like the following:

```
{
  "schema": {...},
  "payload": {
    "before": {                    1
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": {                     2
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {                    3
      "name": "1.5.4.Final",
      "name": "dbserver1",
      "server_id": 223344,
      "ts_sec": 1486501486,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 364,
      "row": 0,
      "snapshot": null,
      "thread": 3,
      "db": "inventory",
      "table": "customers"
    },
    "op": "u",                     4
    "ts_ns": 1486501486308         5
  }
}
```

FIGURE 8 • Debezium's change event format.

Change event storage

Once change events are captured, we can directly deliver them to target systems. But that would make a tight coupling between the CDC system and the target system difficult for them to scale and evolve.

We can avoid that by introducing a middle man, an event bus between the CDC system and target systems. Change events are then written to the event bus, which provides highly scalable and reliable change event storage while preserving the order of received events.

An event bus could be a message broker or an event streaming platform. Message brokers like RabbitMQ or ActiveMQ provide transient event storage by default, but durable messaging comes at the cost of performance. Streaming platforms such as Kafka or Kinesis provide a durable event streaming capability by design. Choosing either of them should be use case driven. You can follow this blog for a detailed comparison across message brokers and event streaming platforms.

Change event propagation

Change events are written to a topic in the event bus, allowing publish-subscribe style event propagation. That allows multiple target systems to subscribe to that topic and receive change events as they are published. Also, the event bus can provide at-least-once or exactly-once delivery guarantee, depending on the implementation.

Usually, an intermediate component consumes the events, applies a lightweight transformation to the event payload, and publishes it to the target system. For example, a connector reads events from the topic, applies transformation, and updates a search index.

Why CDC is better than traditional batch-based ETL

Traditional batch ETL jobs extract databases in bulks, process them, and load them to target systems, significantly reducing data synchronization latency. Conversely, the event-driven CDC approach detects and propagates changes incrementally as they occur.

The following are a few benefits of CDC over traditional ETL and polling-based solutions.

- Changes are detected, captured, and propagated in real-time as they happen. That enables downstream consumers to act upon changes quickly. Compared to traditional batch-oriented systems, that is a considerable gain.
- The loosely coupled nature allows adding or removing components to the architecture with minimum impact. Source and target systems can be upgraded or replaced without affecting each other.
- The event bus in the middle provides reliable delivery of change events. Also, it can buffer incoming events if the rate of event production is higher than consumption. That will be beneficial for slow consumers.
- Unlike polling and trigger-based methods, CDC imposes no performance impact on the source system.

CDC use cases

CDC has been a popular choice for implementing a variety of use cases.

Real-time data integration

CDC can be used for data replication to multiple databases, data lakes, or data warehouses to ensure each resource has the latest version of the data. This way, CDC can provide multiple distributed (and even siloed) teams with access to the same up-to-date data.

Cache invalidation

CDC can be used for cache invalidation to ensure outdated entries in a cache are replaced or removed to display the latest versions.

Real-time analytics dashboards

CDC can synchronize database changes with Business Intelligence dashboards in real-time, for time-sensitive decision-making.

Full-text search

A CDC system can detect source database changes and replicate them with a full-text search engine like Elasticsearch.

Event-driven Microservices

CDC can be used to capture and publish the state changes in Microservices databases as events. That allows the collaborating Microservices to react in an event-driven manner.

CQRS model updates

CDC can capture the state changes in the write model and synchronize them with the read model in Command Query Responsibility Separation (CQRS) architecture.

Auditing and compliance

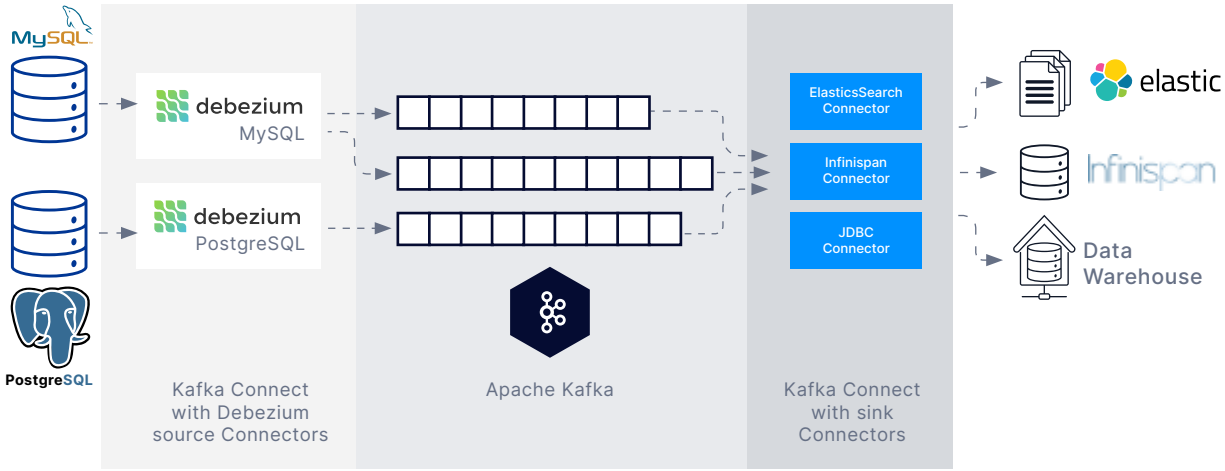
CDC can save the complete history of changes made to a database, allowing the saved to be used for auditing and archiving purposes.

CDC with Debezium and Kafka

Although there are many CDC implementations today, including commercial and open-source solutions, Debezium has been the de facto choice for many organizations.

Debezium is an open-source CDC platform built on top of Apache Kafka. Debezium has connectors to pull a change stream from databases like PostgreSQL, MySQL, MongoDB, and Cassandra and send that to Kafka. Kafka Connect is used as a connector for change detection and propagation.

FIGURE 9 • Debezium works closely with the Kafka ecosystem.



Its close association with the Apache Kafka ecosystem enables the real-time streaming of database change events into a Kafka topic. That allows developers to build event-driven applications to react to change events in real-time.

Conclusion

Synchronizing the changes applied to source databases with derived data systems such as caches, search indexes, data warehouses, etc., is a common problem in any organization. Traditionally, batch ETL pipelines were used, with an inevitable latency added to the process.

Transaction log-based Change Data Capture (CDC) systems enable real-time change capturing from source databases, enabling change events to be seen at derived systems almost immediately. When coupled with a scalable event-driven architecture, a CDC system can capture, store, and propagate change events across multiple derived systems with scalable publish-subscribe semantics.

A combination of Debezium and Kafka has been a popular choice among developers to build distributed, scalable, and reliable CDC systems in production. It is widely used today for use cases like streaming ETL, event-driven Microservices, and real-time data replication across heterogeneous environments.

conduktor.io

Company HQ

USA
Conduktor, Inc.
154 W 14th St
New York, NY 10011

European Offices

London
38 Chancery Lane
London WC2A 1EN
United Kingdom

Dublin
South Point, Herbert House
Harmony Row, Dublin 2
D02 H270, Ireland