Change Data Capture with Apache Pinot - How Does It Work?

By: Hubert Dulay

May 23rd, 2023 • 10 min read

Change Data Capture (CDC) is the process of capturing and communicating changes made to records in a data store, including INSERTs, UPDATEs, and DELETEs transactions to records.

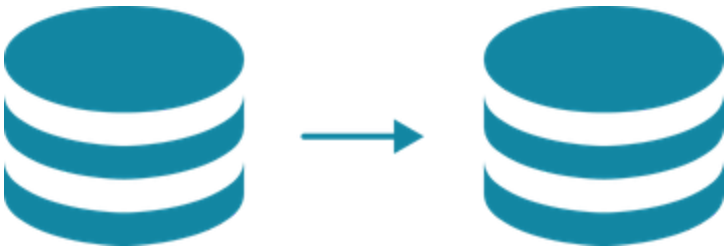
CDC implementations vary across different types of transactional databases, whether SQL or NoSQL. However, the means to ingest and analyze that data in Apache Pinot™ will generally remain the same.

As your applications interact with their data stores, they automatically log the transaction in a construct called a write-ahead log (WAL) in real time. In fact, each transaction reflects an event that has been recorded, naturally giving the WAL event streaming properties. This approach is typically used by relational OLTP databases like PostgreSQL.

NOTE: NoSQL databases also have the ability to perform CDC but may use other mechanisms than a WAL. CDC for NoSQL databases is outside the scope of this post.

The WAL is an append-only, immutable stream of events designed to replicate its data to another instance of the data store for high availability in disaster recovery scenarios (see diagram below). The transactions occurring on the left data store (primary) get replicated to the data store to the right (secondary). The applications connect to the primary data

store and replicate its data to the secondary data store. If the primary data store goes down, the application switches to the secondary data store.



The following diagram shows an example of a WAL in a data store. New transactions get appended to the end of the WAL. The old transactions are on the left, and the newer transactions are on the right.

Change data capture enables you to listen to this WAL by capturing these transactions and sending them downstream for processing. The data processing occurs in a different system where we can view the latest version of each record in other applications. Because of the real-time nature of the data, the subscribing applications to the stream of transactions receive real-time transaction events.

Pre-Image, Post-Image, or Diffs?

An important consideration for CDC is what specific elements of change it captures. Not all CDC implementations are the same. Some provide only the *post-image* — the complete state to which the record changes after an update. Some only provide the *diffs* (or *deltas*) — the specific changes made to the record at the time of the update, not the complete current state of the record. And others can provide the pre-image as well — what the state of the record was before the changes were applied.

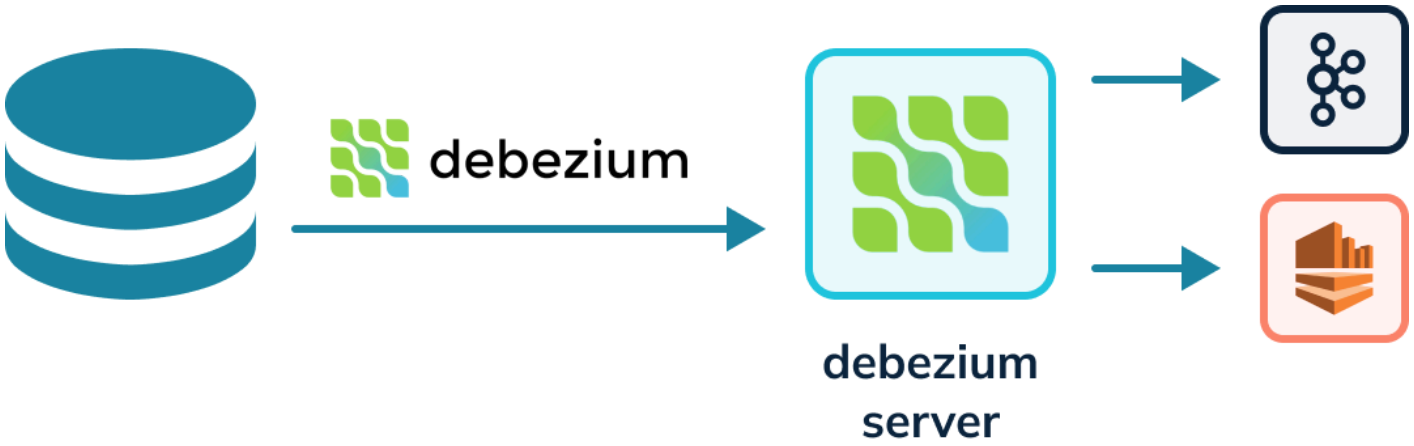
Different transactional databases may only provide one or two of these elements. Usually, it will provide the complete post-image or the diffs (or deltas) to the record. In

other cases, a CDC implementation might provide all three data elements — pre-, post-, *and* diffs. It is very important for you to understand what specific CDC data elements your transactional database provides because of how it limits the kind of analytics you can perform.

How to Capture Change Data with Debezium

Capturing change events requires specific knowledge of the database from which the changes are occurring; and there are many transactional databases. Debezium, an open source project, provides a set of connectors that can subscribe to WALs in many different data stores, such as PostgreSQL, SQL Server, and MongoDB. Their implementation involves the Kafka Connect framework, an open source framework that enables integrations to Apache Kafka®. Two types of connectors exist: source and sink. Debezium connectors are source-only connectors.

Kafka connectors must run in a Kafka Connect cluster, a highly available and distributed system for running connectors. Kafka connectors cannot run on their own and require a server. The Debezium project provides a Debezium server that can also run Debezium connectors capable of writing to other event streaming platforms besides Kafka, for instance, Amazon Kinesis. The diagram below shows a Debezium connector reading the WAL and writing to a Debezium server. The Debezium server can then write to either Kafka or Kinesis.



Debezium Data Format

For details on the Debezium format, check out the tutorial. Below, you'll find an example of a transaction event encoded in JSON coming from the Debezium connector.

```
{
  "schema": {...},
  "payload": {
    "before": {
      "user_id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": {
      "user_id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "name": "2.2.0.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",
    "ts_ms": 1486501486308
  }
}
```

A few elements to note:

- The schema element never changes and defines the schema of the payload
- The payload element holds three different elements:
  - before: shows the state of the record before it was changed; if this is null, then you can assume that the transaction is an INSERT
  - after: shows the state of the record after the record was changed; if this is null, then you can assume that the transaction is a DELETE
  - source: constitutes metadata that describes the source of the data
- The op element defines the actual transaction
  - Values:
    - c for CREATE (or INSERT)
    - r for READ (in the case of a snapshot)
    - u for UPDATE
    - d for DELETE
- The ts_ms element refers to the timestamp in milliseconds of when the transaction occurred

In the op element of the format, you may use a possible r value to determine if the record originated from a snapshot of the entire table in the data store. When the Debezium connector first starts, you could encounter existing records. You can configure the connector to first take a snapshot of the entire table to send as events downstream to its eventual destination. This will affect the treatment of records in the destination, in our case, Apache Pinot.

In Apache Pinot, we will have to create a schema that corresponds to the Debezium format. This could be defined a number of ways. I chose to bring the comments in the after field so users can access the latest values for any customer. I also kept the op at the top level. Since there are no metrics, that context in the schema is an empty array. I also preserved the after and before fields. Notice they are of type STRING. In Apache Pinot, you can assign a JSON index to any field containing multi-level JSON data. Apache Pinot will index all the values in the JSON payload so that any query referencing data in

those JSON fields would be fast. This will allow users to see previous values of the record in cases where the operation was a change. Lastly, I have a date time field to indicate when the last change was made.

```
{
    "schemaName": "customers",
    "dimensionFieldSpecs": [
        {
            "name": "user_id",
            "dataType": "STRING"
        },
        {
            "name": "first_name",
            "dataType": "STRING"
        },
        {
            "name": "last_name",
            "dataType": "STRING"
        },
        {
            "name": "email",
            "dataType": "STRING"
        },
        {
            "name": "op",
            "dataType": "STRING"
        },
        {
            "name": "before",
            "dataType": "STRING"
        },
        {
            "name": "after",
            "dataType": "STRING"
        },
        {
            "name": "source",
            "dataType": "STRING"
        }
```

```
    ],
    "metricFieldSpecs": [],
    "dateTimeFieldSpecs": [
        {
            "name": "ts_ms",
            "dataType": "LONG",
            "format": "1:MILLISECONDS:SIMPLE_DATE_FORMAT:yyyy-MM-dd'T'HH:mm:ss.SS
            "granularity": "1:MILLISECONDS"
        }
    ],
    "primaryKeyColumns": ["user_id"]
}
```

You may have an alternative schema depending on your use case. You don't need any of the fields I preserved. If at the end you only want the latest version, you can do that easily by only preserving the columns that matter to you.

Materialized Views

When looking up your record in Pinot, you only need to provide a WHERE clause with the primary key. Pinot will only return one record—the latest version of the record, not the history of the record—as a true materialized view should. Otherwise, you would have to provide more logic in the SQL statement that selects for the latest record. This adds latency to the query and may make downstream aggregations less accurate. Pinot provides a materialized view by implementing upsert for real-time tables with a primary key.

Upsert in Apache Pinot

Unlike any other real-time OLAP, Pinot offers native support for upsert for real-time ingestion. Upsert logic says, "If the record exists, update it or otherwise insert it."

You need upsert capabilities for dimensional data to simply SELECT for the record's primary key when retrieving it. Without upsert, you will need to find the latest version of

a record by comparing the latest timestamps, which leaves room for error.

This JSON document shows a schema snippet in Pinot that contains a primaryKeyColumns property. By applying this property, Pinot automatically enables the upsert feature. Upsert is completely transparent to the sender and therefore no specific programming is required.

```
{
    "primaryKeyColumns": ["user_id"]
}
```

You can further configure the behavior of the upsert to allow for different behaviors: FULL or PARTIAL.

A FULL upsert means that a new record will replace the older record completely if they share the same primary key.

PARTIAL only allows updates to specific columns and employs additional strategies.

Strategy	Description
OVERWRITE	Overwrite the column of the last record
INCREMENT	Add the new value to the existing values
APPEND	Add the new item to the Pinot unordered set
UNION	Add the new item to the Pinot unordered set if not exists
IGNORE	Ignore the new value, keep the existing value (v0.10.0+)
MAX	Keep the maximum value betwen the existing value and new value (v0.12.0+)
MIN	Keep the minimum value betwen the existing value and new value (v0.12.0+)

Source: Stream Ingestion with Upsert

Here is a sample snippet of a table configuration containing the property that configures the upsert strategy:

```
"upsertConfig": { "mode": "FULL" },
```

Upsert simplifies client queries in an extremely powerful way. More importantly, upsert assures the accuracy of any aggregations applied to updated columns, which proves especially important when the analytics lead to critical decisions.

Summary

Change data capture is the best way to capture changes in a database. Other options require comparing snapshots or applying complex modified timestamp logic. Other

solutions only emulate real-time, but change data capture embodies the only genuine real-time event streaming solution.

Debezium provides many other CDC connectors that you can find in their documentation. If you do not have a Kafka Connect cluster or do not use Kafka at all, you can use the Debezium server to run the CDC connectors and write to an alternative streaming system, such as Amazon Kinesis, Pub/Sub from Google Cloud, Apache® Pulsar™, Azure Event Hubs, and RabbitMQ.

Lastly, Apache Pinot enables upsert for any client sinking into it, which means the client does not need to implement upsert logic. Any client can generate a materialized view in Pinot. This makes the resulting table faster to query and provides more accurate analytics.

To try Pinot in the cloud, visit startree.ai for a free trial.

Foundation. This page has references to third party software - Presto, PrestoDB, ThirdEye, Trino, TrinoDB, that are not part of the Apache Software Foundation and are not covered under the Apache License.