DataCater

Technical guide
March 2023

Change Data Capture 101

Everything you need to know to get started with applying change data capture to database systems and APIs.

Contents

1 Motivation

Many data architectures employ different data stores for different use cases. They may use transactional database systems for organizing raw operational data, data warehouse systems for serving data to downstream analytics or ML applications, and search engines for providing powerful search interfaces. In addition to maintaining internal data systems, many enterprises need to integrate external data services into their architecture as well, which further increases the number of data stores in use.
Data stores are rarely operated in isolation but often need to exchange data with each other. Exemplary use cases for transferring data from a data source to a data sink are:

- Loading operational data from a database (data source) into a data warehouse system (data sink),
- indexing a database system (data source) with a search engine (data sink), or
- integrating external data services (data source) into internal database systems (data sink).

Instead of performing dual writes into multiple data stores at the same time, we recommend using one data store as the primary location of your data and transferring the data from that store to other locations. The common approach to transferring data between data stores, a problem often described as Extract Transform Load (ETL), is the recurrent execution of data pipelines that extract data from a data source, optionally apply transformations to the data, and eventually write the data to a data sink. While this approach is fairly easy to implement, it is very inefficient when applied in practice. For each run, data pipelines have to consider all data from the data source, even if there have not been any changes since the last run, effectively wasting computing resources and putting a significant load on all involved systems. To avoid impacting the performance of the consumed data source and interfering with other workloads, data pipelines are traditionally executed with a very low frequency, e.g., each night at 2 AM. As a consequence, data sinks are seldom in sync with data sources.

What if we could do better?

2 Introduction to Change Data Capture

Change data capture (CDC) is a technique used to detect and capture record-level change events that occurred in data stores, e.g., insertions of new records, updates of existing records, or deletions of records.

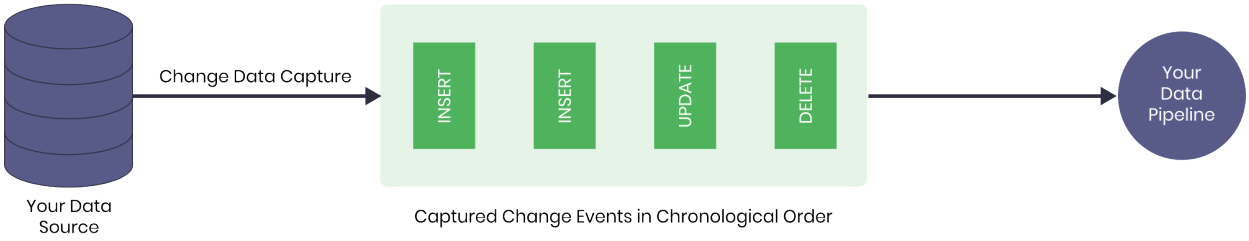

Figure 1: *Change data capture extracts change events, like INSERTs or UPDATEs, from data systems and makes them available for further processing.*

CDC offers two main advantages compared to the traditional full copy of data sets. First, data pipelines need to consider only the data that have changed since the last run, which requires much less computing resources. Second, given that information on change events become available immediately after their occurrence, data pipelines could turn into streaming applications, which process data change events in real time and always keep data sinks in sync with data sources.
The following sections discuss different approaches to implementing CDC.

2.1 Diff-based Change Data Capture

The naive approach to capturing data changes is comparing the current state of the data source to the state of the data source when the data pipeline has been last executed.
This diff-based CDC implementation has two main disadvantages. First, for computing the differences the implementation would still need to access all data regardless of whether they have changed. Second, space usage would strongly increase because the implementation would need to always keep a snapshot of the data from the time of the last run.
In practice, diff-based CDC implementations rarely offer advantages compared to full copies of the data unless the execution of data transformations or the loading of data into the data sink are the performance bottlenecks:

- ✓ Detection of all data change events (insertions, updates, and deletions)

- ✗ Uses the query layer for extracting data, which puts additional load on the data source
- ✗ Requires recurrent polling of the data source
- ✗ Determining the difference between two data sets is a compute-heavy operation that renders frequent executions impossible
- ✗ Additional space requirements for caching the state of the data set

2.2 Timestamp-based Change Data Capture

To ease the detection of changes one could introduce a column to the schema of the data source that holds the time when a record has been last modified. In some data sets, such a column may even already exist. Data pipelines could query the timestamp column to retrieve only those records that have been changed since the last run.
Timestamp-based CDC cannot capture deletions and is thus only suitable for applications that perform soft deletions or do not delete records at all.

- ✓ Can be applied to any kind of data store, e.g., APIs, database systems, object stores, etc.
- ✗ Uses the query layer for extracting data, which puts additional load on the data source
- ✗ Requires recurring polling of the data source
- ✗ Requires the presence of a column that tracks the time when the record has been last modified
- ✗ Cannot capture DELETEs unless the application uses soft deletions

2.3 Trigger-based Change Data Capture

Many database systems provide trigger functions as a means to performing user-defined actions once events, like insertions of data, occur (e.g., Triggers in PostgreSQL, Triggers in SQL Server, or Triggers in MySQL). Trigger functions could be used to capture all changes that occur in a particular table and track them, for instance, in a separate table used as an event queue. While trigger functions would allow data pipelines to only process the data that have changed since the last run, they still require recurring polling of the event table.

- ✓ Event-driven capturing of changes
- ✓ Detection of all data change events (INSERTs, UPDATEs, and DELETEs)

- ✓ Support for tracking custom information, such as the name of the database user performing the operation
- × Only useable with database systems
- × Change events trigger writes to the event table, often strongly increasing the execution time of the original operation (or transaction)
- × Changes to the schema of the data table must be manually propagated to the event table
- × Requires recurring polling of the event table
- × Vendor-specific code for implementing trigger functions, which impedes future migrations to other database systems

2.4 Log-based Change Data Capture

Most database systems maintain transaction logs (or operation logs) for replication or recovery reasons. These logs keep track of all state-changing operations applied to a database table. The log-based approach to implementing CDC leverages these logs for capturing and extracting change events.

Transaction logs are typically updated in real time and do not create any computational overhead, like trigger functions, which enables an implementation of CDC that does not only allow the event-driven processing of changes but also keeps the performance of the data source unaffected.

Although most of today's database systems provide transaction logs (see, for instance, MySQL's Binary Log, logical replication in PostgreSQL, or MongoDB's Oplog), log-based CDC cannot be used with any database system. Especially very old releases of database systems do not support such functionality.

Transaction logs do not keep the full history of operations performed on a database table, but only store the operations applied within a particular retention period (for instance, within the last seven days). At DataCater, we read a full snapshot of a database table before starting to consume the transaction log to be able to offer a full replication of a data source to a data sink.

In the context of CDC, replication logs offer the following advantages and disadvantages:

- ✓ Event-driven capturing of changes
- ✓ Detection of all data change events (INSERTs, UPDATEs, and DELETEs)

- ✓ No impact on the performance of the data source
- × The implementations of transaction logs are specific to the vendor, which impedes future migrations to other database systems
- × Transaction logs typically do not store the full history of a database table but only the operations performed within a particular retention period
- × Only useful with database systems

3 Implementing CDC with PostgreSQL

PostgreSQL is a famous open-source database management system, which is in production at a plethora of enterprises. In the typical setup, PostgreSQL manages the transactional data of applications, such as products in an e-commerce shop, and integrates third-party data systems for other purposes, e.g., a data warehouse for analytics, a BI tool for reporting, etc. The traditional approach to connecting PostgreSQL with other data stores is batch-based. From time to time, data pipelines extract all data from PostgreSQL and send them to downstream data stores, which is not only inefficient but also prone to errors.
CDC is a modern alternative that can extract record-level change events (INSERTs, UPDATEs, and DELETEs) from PostgreSQL in real-time. The main benefits of change data capture are:

- CDC captures change events in real-time, keeping downstream systems, such as data warehouses, always in sync with PostgreSQL and enabling fully event-driven data architectures.
- Using CDC reduces the load on PostgreSQL since only relevant information, i.e., changes, are processed.
- CDC enables the efficient implementation of use cases requiring access to change events of PostgreSQL, such as audit or changelogs, without modifying the application code.

In the following sections, we provide a comprehensive introduction to using change data capture with PostgreSQL. We cover three common approaches to implementing change data capture: triggers, queries, and logical replication. While each approach has its own advantages and disadvantages, our clear favorite is log-based CDC using PostgreSQL's logical replication. It is not only highly efficient, capturing all event types in real-time without harming the performance of the PostgreSQL database, but is also widely available, whether you're using a self-managed or a managed PostgreSQL installation, and applicable without introducing changes to the database schema.

3.1 Implementing CDC with Triggers in PostgreSQL

With trigger functions, we can listen for all INSERT, UPDATE, and DELETE events occurring in the table of interest and, for each event, insert one row into a second table, effectively building a changelog.

The PostgreSQL community provides a generic *TRIGGER* function, supported by PostgreSQL version 9.1 and newer, which stores all change events in the table *audit.logged_actions*. If we want to enable the trigger-based CDC approach for the table *public.users*, we can run the following SQL statement:

```
SELECT audit.audit_table('public.users');
```

Listing 1: *Enable Trigger-based CDC for the PostgreSQL table public.users.*

This approach to CDC stores captured events only inside PostgreSQL. If you want to sync change events to other data systems, such as a data warehouse, you would have to recurringly query the PostgreSQL table holding the change events (here named *audit.logged_actions*), which increases the complexity of the implementation.
Let's compare the pros and cons of implementing change data capture with triggers in PostgreSQL:

- ✓ Changes are captured instantly, enabling the real-time processing of change events
- ✓ Triggers can capture all event types: INSERTs, UPDATEs, and DELETEs
- ✓ By default, the PostgreSQL *TRIGGER* function used here adds helpful metadata to the events, e.g., the statement that caused the change, the transaction ID, or the session user name
- × Triggers increase the execution time of the original statement and thus hurt the performance of PostgreSQL
- × Triggers require changes to the PostgreSQL database
- × If change events shall be synced to a data store other than the same PostgreSQL database, we would need to set up a separate data pipeline, which polls the table filled by the trigger function (here *audit.logged_actions*)
- × Creating and managing triggers induces additional operational complexity

3.2 Implementing CDC with Queries in PostgreSQL

Another way to implement change data capture with PostgreSQL is the query-based approach.
If the schema of the monitored database table features a timestamp column indicating when a row has been changed the last time, we could recurringly query PostgreSQL using that

column and ask for all records, which have been modified since the last time we ran the same query. Assuming a table named *public.users* and a timestamp column named *updated_at*, such a query could be implemented as follows:

```
SELECT * FROM public.users WHERE updated_at > 'TIMESTAMP_LAST_QUERY';
```

Listing 2: *SQL query to extract all users, which have been modified since the last access time (please replace TIMESTAMP_LAST_QUERY).*

Note that the query-based CDC approach cannot capture DELETEs (unless using soft deletions). In the following, we list its advantages and disadvantages:

- ✓ Query-based CDC can be implemented without introducing any changes to PostgreSQL, if the schema holds a timestamp column indicating the modification time of rows
- × Query-based CDC implementations use the query layer for extracting data, which puts additional load on PostgreSQL
- × Query-based CDC requires the recurring polling of the monitored table (here *public.users*), which wastes resources if data rarely change
- × Query-based CDC requires the presence of a column (here *updated_at*) that tracks the time when the record has been last modified
- × Query-based CDC cannot capture DELETE events (unless the application uses soft deletions)

3.3 Implementing CDC with Logical Replication in PostgreSQL

Since version 9.4, PostgreSQL offers logical replication for efficiently and safely replicating data between different PostgreSQL instances on possibly different physical machines. Technically, it's a write-ahead log on disk, which holds all events that change the data of the PostgreSQL database, e.g., INSERTs, UPDATEs, and DELETEs.
PostgreSQL uses a subscription model with publishers and subscribers for the implementation of logical replication. For the purpose of implementing change data capture, we might employ the database of interest as a publisher and subscribe ourselves to its log.
While many database systems might already use logical replication, it is not enabled by default. You can enable logical replication by introducing the following change to the configuration file postgresql.conf:

```
wal_level = logical
```

Listing 3: *Enable logical replication in the config file postgresql.conf.*

In the next step, you need to grant your replication user (here *repuser*) network access in the configuration file pga_hba.conf (please refer to the PostgreSQL documentation for individual configuration):

```
host     all     repuser     0.0.0.0/0     md5
```

Listing 4: *Allow logical replication in the config file pg_hba.conf.*

Let's assume that you want to capture changes from the table *public.users*. You can enable CDC for this table by creating a new publication as follows:

```
CREATE PUBLICATION newpub
  FOR TABLE public.users;
```

Listing 5: *Create a publication for table public.users.*

In the next step, you can start subscribing to this publication. If you would like to consume the events from another PostgreSQL instance, you might create the subscription as follows:

```
CREATE SUBSCRIPTION newsub
  CONNECTION 'dbname=foo host=bar user=repuser'
  PUBLICATION newpub;
```

Listing 6: *Create a subscription for consuming changes.*

Technically, logical replication is implemented by a logical decoding plugin. If you are using a PostgreSQL version older than 10, you need to manually install a plugin in your PostgreSQL database, e.g., wal2json or decoderbufs. Since version 10, PostgreSQL ships the plugin pgoutput by default.
For the technical implementation of log-based change data capture, we highly recommend using one of the existing open-source projects, such as Debezium. DataCater's PostgreSQL source connector is based on Debezium.
Most managed PostgreSQL services, which we are aware of, offer support for logical replication, e.g., AWS RDS, Google Cloud SQL, or Azure Database.
The following list shows the advantages and disadvantages of using PostgreSQL's logical replication for implementing CDC:

- ✓ Log-based CDC enables the event-driven capturing of data changes in real-time
- ✓ Downstream applications have always access to the latest data from PostgreSQL

- ✓ Log-based CDC can detect all change event types in PostgreSQL: INSERTs, UPDATEs, and DELETES
- ✓ Consuming events via logical replication boils down to directly accessing the file system, which does not impact the performance of the PostgreSQL database
- × Logical replication is not supported by very old versions of PostgreSQL (older than 9.4)

4 Implementing CDC with MySQL

MySQL is one of the most popular database management systems in the world, successfully powering many applications since its first introduction in 1995. MySQL is typically used for managing the core (or transactional) data of applications, such as products or sales in an e-commerce shop, and is often complemented with other data systems, e.g., a data warehouse for analytics, a search engine for search, etc.
The traditional approach to syncing MySQL with complementary data stores is batch-based. From time to time, data pipelines extract all data from the MySQL database system and send it to downstream data stores. Change data capture (CDC) is a modern alternative to inefficient bulk imports. CDC extracts change events (INSERTs, UPDATEs, and DELETEs) from data stores, such as MySQL, and provides them to a data pipeline. The main advantages of CDC are:

- CDC typically captures changes in real-time, keeping downstream systems, such as data warehouses, always up-to-date and enabling event-driven data pipelines.
- Using CDC decreases the load on all involved systems since only relevant information, i.e., data change events, are processed.
- CDC enables straightforward implementations of use cases requiring access to data changes, such as an audit or changelog, without changing the code of applications.

In the next sections, you will get a full introduction to using change data capture with MySQL. We cover three common approaches to implementing change data capture: triggers, queries, and MySQL's Binlog. While each approach has its own advantages and disadvantages, at DataCater our clear favorite is log-based CDC with MySQL's Binlog. It is not only highly efficient, capturing all event types in real-time without harming the performance of the MySQL database, but is also widely available, whether you're using a self-managed or a managed MySQL installation, and applicable without introducing changes to the database schema.

4.1 Implementing CDC with Triggers in MySQL

Using MySQL's support for TRIGGER functions we can listen for all INSERT, UPDATE, and DELETE events occurring in the table of interest and, for each event, insert one row into another table, effectively building a changelog.
Let's assume a MySQL database features the table *users*:

```
CREATE TABLE users (
  user_id    INT,
  user_name  TEXT,
  user_email TEXT
);
```

Listing 7: *Schema of the MySQL table users.*

We could create a trigger, which captures all INSERT events from the table *users* and appends a new row for each INSERT event to the table *users_change_events*:

```
CREATE TABLE users_change_events (
  log_id          BIGINT AUTO_INCREMENT,
  event_type      TEXT,
  event_timestamp TIMESTAMP,
  user_id         INT,
  user_name       TEXT,
  user_email      TEXT
  PRIMARY KEY ('log_id')
);

CREATE TRIGGER user_insert_capture
 AFTER INSERT ON users
 FOR EACH ROW BEGIN
 INSERT INTO users_change_events (
   event_type,
   event_timestamp,
   user_id,
   user_name,
   user_email
 )
 VALUES (
   'INSERT',
   now(),
   NEW.user_id,
   NEW.user_name,
   NEW.user_email
```

```
27 );
```

Listing 8: *Schema of the MySQL table users_change_events and definition of the MySQL trigger user_insert_capture.*

UPDATE and DELETE events can be captured accordingly.
By default, this approach stores captured events only inside MySQL. If you want to sync change events to other data systems, you would have to recurringly query the MySQL table holding the events, which increases the complexity.
Let us have a look at the pros and cons of implementing change data capture with triggers in MySQL:

- ✓ Changes are captured instantly, enabling the real-time processing of change events
- ✓ Triggers can capture all event types: INSERTs, UPDATEs, and DELETEs
- ✓ Using triggers, it's easy to add custom metadata to the change event, such as the name of the database user performing the operation
- × Triggers increase the execution time of the original operation and thus hurt the performance of the database
- × Triggers require changes to the MySQL database
- × Changes to the schema of the monitored table (here *users*) must be manually propagated to the table filled by the trigger function (here *users_change_events*)
- × If change events shall be synced to a data store other than the same MySQL database, we would need to set up an external data pipeline, which polls the table filled by the trigger function (here *users_change_events*)
- × Creating and managing triggers induces additional operational complexity

4.2 Implementing CDC with Queries in MySQL

The second approach to implementing change data capture with MySQL is a query-based approach. If the schema of the monitored database table features a timestamp column indicating when a row has been changed the last time, we could recurringly query MySQL using that column and ask for all rows, which have been modified since the last time we queried MySQL. Assuming a table named users and a timestamp column named *updated_at*, such a query could be implemented as follows:

```
SELECT * FROM users WHERE updated_at > 'TIMESTAMP_LAST_QUERY';
```

Listing 9: *SQL query to extract all users, which have been modified since the last access time (please replace TIMESTAMP_LAST_QUERY).*

Note that query-based CDC cannot capture DELETEs - unless using soft deletions - but is limited to INSERT and UPDATE events.

- ✓ Query-based CDC can be implemented without introducing any changes to the database, if the schema holds a timestamp column indicating the modification time of rows
- × Query-based CDC implementations use the query layer for extracting data, which puts additional load on the MySQL database
- × Query-based CDC requires the recurring polling of the monitored table, which wastes resources if the state of MySQL does not change between two accesses
- × Query-based CDC requires the presence of a column that tracks the time when the record has been last modified
- × Query-based CDC cannot capture DELETE events unless the application uses soft deletions

4.3 Implementing CDC with Binlog in MySQL

MySQL offers the Binlog for efficiently and safely replicating data between different database instances on possibly different physical machines. Technically, the Binlog is a binary file on disk, which holds all events that change the state of the MySQL database, e.g., INSERTs, UPDATEs, DELETEs, schema changes, etc.
For the purpose of implementing change data capture, we might attach to the Binlog of the MySQL database and consume all change events occurring in the monitored table.
While many database systems already use the Binlog for replication purposes it is not enabled by default. If your MySQL instance does not yet use the Binlog, you can enable it by introducing the following changes to the MySQL configuration:

```
server-id          = 42
log_bin            = mysql-bin
binlog_format      = ROW
expire_logs_days   = 10
```

```
# define `binlog_row_image` for MySQL 5.6 or higher,
# leave it out for earlier releases
binlog_row_image  = FULL
```

Listing 10: *Enable the Binlog in the configuration of MySQL.*

Managed MySQL instances typically do not directly expose access to the configuration. However, most managed MySQL services, which we are aware of, offer support for the MySQL Binlog, e.g., AWS RDS, Google Cloud SQL, or Azure Database.
For the technical implementation of Binlog-based change data capture, we highly recommend using one of the existing open-source projects, such as Debezium or Maxwell's Daemon. DataCater's MySQL source connector is based on Debezium.
The following list shows the advantages and disadvantages of using MySQL's Binlog for implementing CDC:

- ✓ Binlog-based CDC enables the event-driven capturing of data changes in real-time
- ✓ Downstream applications have always access to the latest data from MySQL
- ✓ Binlog-based CDC can detect all change event types: INSERTs, UPDATEs, DELETES, and even schema changes
- ✓ Since reading events from the Binlog boils down to accessing the file system, this CDC approach does not impact the performance of the MySQL database
- × The Binlog is not available in very old versions of MySQL (at DataCater we support MySQL 5.5 or newer)
- × The Binlog does not store the entire history of change events performed on the database table but only the operations performed within a particular retention period (defined by the configuration option *expire_logs_days*), which is why we typically combine it with an initial full snapshot of the monitored table using a *SELECT * FROM table_name;* query

5 Implementing CDC with Web APIs

Today, CDC is most popular for being used as an efficient means to extracting data from database systems. The last two sections discussed how to use CDC with the database systems PostgreSQL and MySQL. Database systems typically provide a replication log, which can serve as the source for change events and is consumed by CDC connectors, like the excellent Debezium. However, CDC is not restricted to database systems but can also be used in other cases, where we don't have access to an event log, such as web APIs.
In this section, we show how to deploy change data capture to extract change events from JSON-based web APIs. As an example, we have a look at Shopify's Products endpoint, which returns a list of products:

```
{
  "products": [
    {
      "id":           632910392,
      "title":        "IPod Nano - 8GB",
      "vendor":       "Apple",
      "product_type": "Cult Products",
      "handle":       "ipod-nano",
      "created_at":   "2021-07-01T13:58:02-04:00",
      "updated_at":   "2021-07-01T13:58:02-04:00",
      [...]
    }
  ]
}
```

Listing 11: *JSON response from Shopify's Products endpoint.*

Our goal is to turn this static API response, which provides the state of the products at query time, into a real-time changelog that can be processed with a (streaming) data pipeline.

5.1 Status quo for (too) many use cases

The status quo for extracting data from a web API, such as Shopify's Products endpoint, is a recurring bulk load. For instance, each night at 2 am, one might access the API, extract all products, and pass them to a data pipeline for further processing - regardless of whether the products have been updated since the last access.

Depending on the amount of data, bulk loads cannot be performed too often because they might degrade the performance of downstream systems. While consuming a web API is usually not causing any performance issues, processing the consumed data and writing them into downstream data systems, such as other web APIs, might take (much) more time and become a bottleneck.

5.2 Keeping track of processed data

A first step to reduce the load on downstream systems is to enable the data extraction to detect which data have been changed since the last access. To this end, we might use timestamps provided by the API indicating the last time a record has been changed. In the case of Shopify's Products endpoint, this attribute is called *updated_at*. When consuming the data from the API, the connector would only consider those records, which have a value in the *updated_at* attribute greater than the last query time.
If the API does not provide such timestamps, we would need to use other, less preferable, means to detecting changes. For instance, we could maintain a set of the hashes of processed records and periodically compare the hashes of the records from the API with this internal set - if we discover an unknown hash we know that we have not yet processed this record.

5.3 Consuming only changes from a web API

At this point, we already made the data processing more intelligent. Although we can detect data changes, we still need to periodically extract all data from the API. What if we could improve on that?
Most APIs provide means to filter the data while querying them, typically by defining a parameter in the query string. For instance, Shopify allows us to define the query parameter *updated_at_min* to retrieve only products that have been updated after a given timestamp. Let's assume it's 2021-08-05 08:00 and we queried the Shopify API the last time one minute ago. We could access the following URI to retrieve all products, which have been changed since our last access:

```
/admin/api/2021-07/products.json?updated_at_min=2021-08-05T07:59:00-04:00
```

Using such filters allows us to consume only relevant data and query the API at a relatively high frequency.

5.4 But… Can't we just use webhooks?

In the context of web APIs, there is an alternative approach for exchanging change events: Webhooks. In this case, you - the subscriber - have to provide an API endpoint, which is called by the application whose change events you want to consume, whenever a change happens. While webhooks are widely supported in web applications, they have one main disadvantage compared to CDC: Webhooks don't support replaying events. If the subscribing API is offline when the event occurs (or the webhook is fired), the data get lost. In contrast, a CDC-based approach can recover from the downtime by continuing at the last read timestamp and will not lose any data.

5.5 How to put change data capture with APIs into practice

When it comes to selecting specific technologies for the implementation, we're of course a bit biased. At DataCater, we're big fans of Apache Kafka® for storing event data. The Kafka community provides an open-source connector for applying change data capture in the context of web APIs.
Of course, you may also implement your own connector or use something else than Apache Kafka.
If you want to save time and headaches, you might also consider using DataCater for implementing change data capture with web APIs. DataCater offers a plug-and-play CDC connector for web APIs, which takes only a few minutes to configure. In the case of the Shopify API, all you need to do is to fill out seven text fields in a web form.

6 Summary

In this guide, we provided an introduction to change data capture, discussed different techniques for capturing change events, and showed how to use CDC with the database systems PostgreSQL and MySQL as well as web APIs.
Change data capture is a very powerful tool for extracting data from source systems and enables you to transfer only relevant data. It is a foundational technology for implementing streaming data pipelines, which replicate data in real-time between different data systems.
Interested in trying out CDC in practice? Sign up for the free 14-day trial of DataCater Cloud and start capturing events from your data sources within a few minutes.

Published by DataCater GmbH in March 2023

DataCater is the real-time, cloud-native data pipeline platform based on Apache Kafka® and Kubernetes® that enables data and developer teams to unlock the full value of their data faster. DataCater is a simple yet powerful approach to building modern, real-time data pipelines. According to reports of our users, data and dev teams save 40% of the time spent on crafting data pipelines and go from zero to production in a matter of minutes. Users can choose from an extensive repository of filter functions, apply transformations, or code their own transforms in Python® to build their streaming data pipelines.

Website: https://datacater.io
Start a 14-day free cloud trial: https://cloud.datacater.io/sign_up

DataCater