DATABASE REPLICATION WITH CHANGE DATA CAPTURE OVER KAFKA

BY DOMINIQUE CHANET
LEAD ARCHITECT AT KLARRIO

Klarrio
STREAMING AHEAD

INTRODUCTION

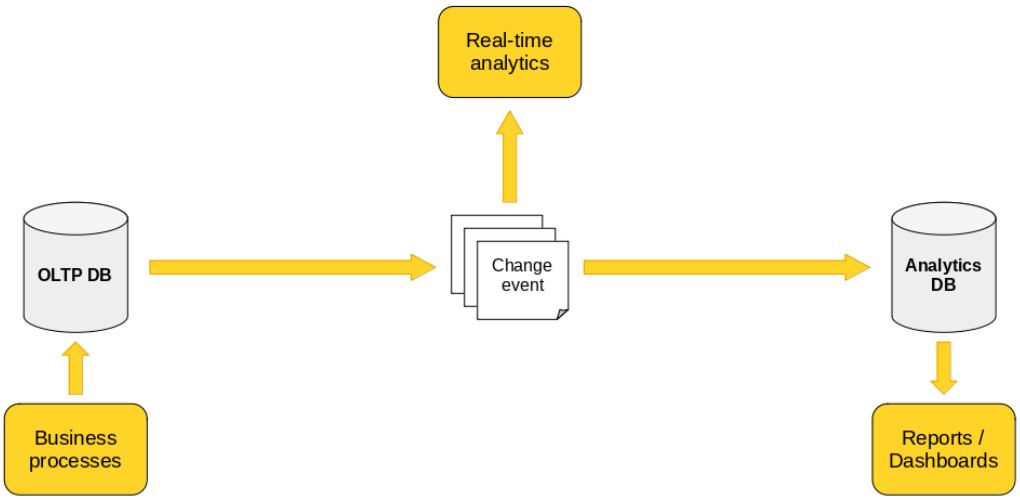
Suppose you have built your operations around a solid (and probably costly) relational database like Oracle or MS SQL Server. Your business is growing, and the demands on the database are growing exponentially faster: not only is there a linear growth in the number of business transactions, but your Business Intelligence team is constantly coming up with new, exciting analyses and dashboards that, unfortunately, require you to run heavy queries against your database.

The database is creaking at the seams, and you can't have that, because your business relies on it. Looks like the only option is to open your wallet, and pay Oracle or Microsoft another cool million for additional capacity, right?

Well, maybe not. What if you could, relatively cheaply, create a read-only copy of the data in an open-source database server, like PostgreSQL or MySQL? While your critical business processes keep humming along in the tried-and-trusted commercial database, your BI team can go wild on a read-only copy of the data that is stored in a cheaper, non-performance-sensitive database.

And maybe your BI team has the ambition to provide you with real-time insights, for which you need real-time streaming data processing. In other words: you'd like to capture the changes that happen in the production database and act on them in real-time.

We're aiming for the double whammy here: extract all changes from the database in real-time, push them onto Kafka topics, and store them in a secondary database for offline analysis. Those same Kafka topics can then be used for real-time analysis with Flink or ksqlDB. Conceptually, the solution looks like this:



This is a question we get asked at Klarrio on a regular basis, so we dove in and figured out what is possible with the open-source tools available today.

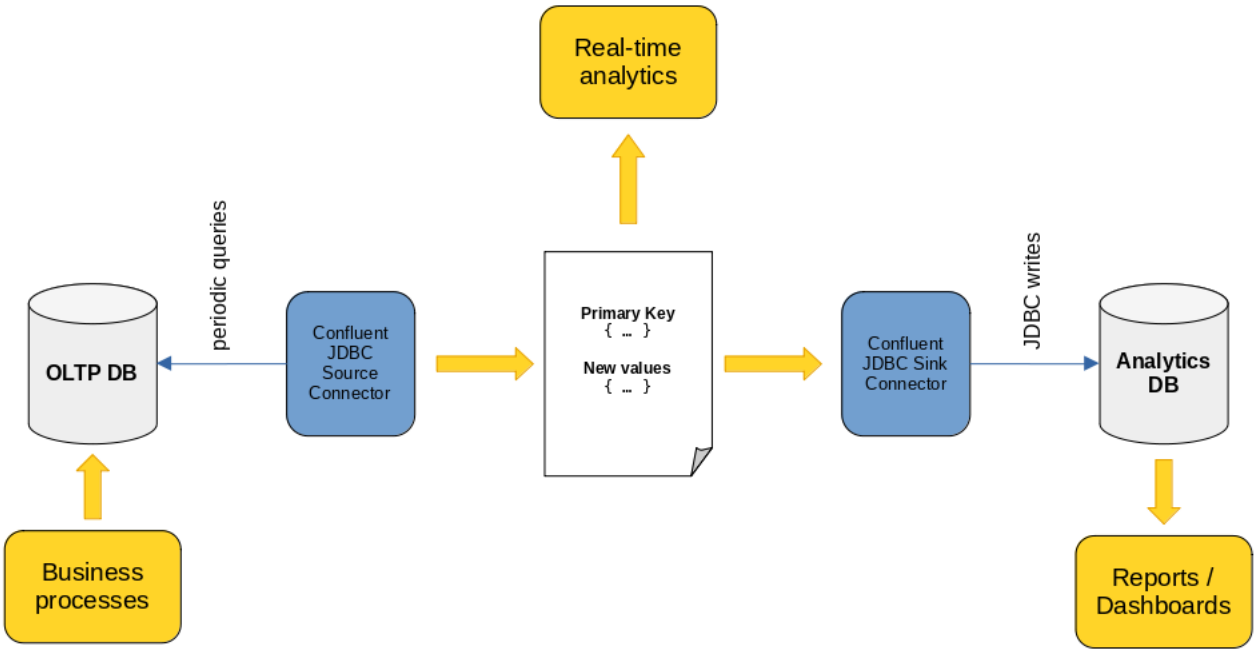
CONFLUENT'S KAFKA CONNECT JDBC CONNECTOR

As luck would have it, it looks like Confluent (the corporate backers of Apache Kafka) have exactly what we need: the Kafka Connect JDBC Connector.

On the face of it, it has all the necessary building blocks:

- a source connector that can connect to a bunch of different databases, and extract table changes,
- a sink connector that can consume the change records emitted by the source connector, and write them to various target databases.

We can expand our conceptual diagram from above into something like this:



So it should be as simple as spinning up a Kafka Connect cluster, configuring the JDBC Source and Sink connectors, and we're done, right? Unfortunately, it isn't. Let's review some of the difficulties we encountered trying to do just that.

CHALLENGES

LICENSE

Depending on your use case, this may be a biggie or a total non-issue, but it's worth mentioning nonetheless. The connectors are licensed under the Confluent Community License (CCL). This is not an Open Source Initiative approved open source license and is likely, not suitable for use in many corporate contexts. The CCL FAQ will likely answer any questions you have regarding its suitability.

HOW CHANGES ARE CAPTURED

The JDBC Source connector can use different strategies to capture changes in the source tables. Unfortunately, none of them are really generally suitable.

- bulk mode will periodically dump the entire contents of the table. That is pretty wasteful for large tables, and is not exactly suitable for real-time change data capture: if you do this at a high frequency, you'll kill the performance of the source database. And even then, you'd always have to compare the two latest snapshots to distill what actually changed in the meantime.

- incrementing mode requires all source tables to have a column that is strictly incremented whenever new rows are added. That column must have the same name in all tables you want to capture. If you comply with these demands (which potentially means that you'll have to change the source database schema), the connector still only captures new rows that are added to the table. UPDATEs and DELETEs are not captured at all.

- timestamp mode requires all source tables to have a column that holds the last-update timestamp. You guessed it: that column must have the same name in all tables. If you adapt all table schemas to comply with this condition, the connector will capture INSERT and UPDATE actions for you, but it still won't capture DELETEs.

HOW CHANGES ARE REPRESENTED

The JDBC Connectors use a very basic data format to represent change records on Kafka:

- the message key is a struct with the values for the table's primary key fields. If a table does not have a primary key, the key is empty.

- the message value is a struct with the new values for the affected table row.

This message format is sufficient to represent INSERTs in a table, and UPDATEs (and potentially DELETEs) in a table with a well-defined primary key.

However, in a table without a primary key, the available information isn't sufficient to distinguish UPDATE from INSERT events. Furthermore, even if the source connector could capture DELETE events, for a table without a primary key they would be emitted as messages with an empty key and an empty value, so there is no way to figure out exactly which table row was deleted.

It's not exactly best practice to define database tables without a primary key, but they do exist in the field, so it's worth noting that this representation can't really deal with them.

HOW DATABASE VALUES ARE REPRESENTED

Kafka Connect has its own type system that abstracts away the differences between the type systems of the various data sources and sinks it can connect to. In theory, this is a great concept, as it makes data integration between various sources incredibly smooth and hassle-free. The downside, however, is that details can get lost in translation.

SQL databases have a very rich type system that allows the representation of values with arbitrary precision, like precise decimal numbers and timestamps down to the microsecond level. Out of the box, the Kafka Connect type system cannot represent all of these rich types in their full resolution. While the type system is extensible, the JDBC connectors don't take advantage of this, and simply cast all captured values to one of the standard Connect types.

CONCLUSION

So where does that leave us? If your aim is to build a generally applicable CDC/replication system that faithfully replicates all source data, the Confluent JDBC Connectors are clearly insufficient.

They do, however, provide a hassle-free, out-of-the-box solution if the following conditions are met:

- the Confluent Community License should be acceptable to your organization,
- your database tables are append-only, or append-and-update-only (no DELETEs),
- in case you need support for UPDATE events, you only need it on tables with a primary key,
- your table schemas fit the JDBC Source Connector requirements, or you're prepared to adapt them to fit,
- you can live with the loss of precision incurred by the translation to standard Kafka Connect data types.

DEBEZIUM AS THE SOURCE CONNECTOR

It turns out that the Confluent JDBC Source connector is not exactly the gold standard in open source CDC solutions. That gold standard does exist, though, in the form of Debezium.

Debezium ticks a lot of boxes for us:

- It's licensed under the Apache 2.0 license, which is both Open Source and corporate-friendly.
- It's an actively maintained project with a vibrant open source community of contributors.
- It supports a lot of databases: Oracle, MS SQL Server, PostgreSQL, MySQL, ...
- It hooks into the database's replication mechanisms to provide real-time streaming updates, capturing INSERT, UPDATE, DELETE and TRUNCATE actions alike without difficulty, and without requiring any updates to the source table schemas.
- It provides rich change events that include old and new values for each column and even provides transaction IDs to correlate changes across different tables.
- It extends the basic Connect type system with a rich set of additional types that can capture the full precision of the source database type system.

We have experimented extensively with Debezium as a CDC tool, and we're pleased to report that it works really well. Be aware that there are some rough edges still, especially concerning the handling of very large change records and large object (BLOB, CLOB) support. For the 99% use case, though, it works like a charm.

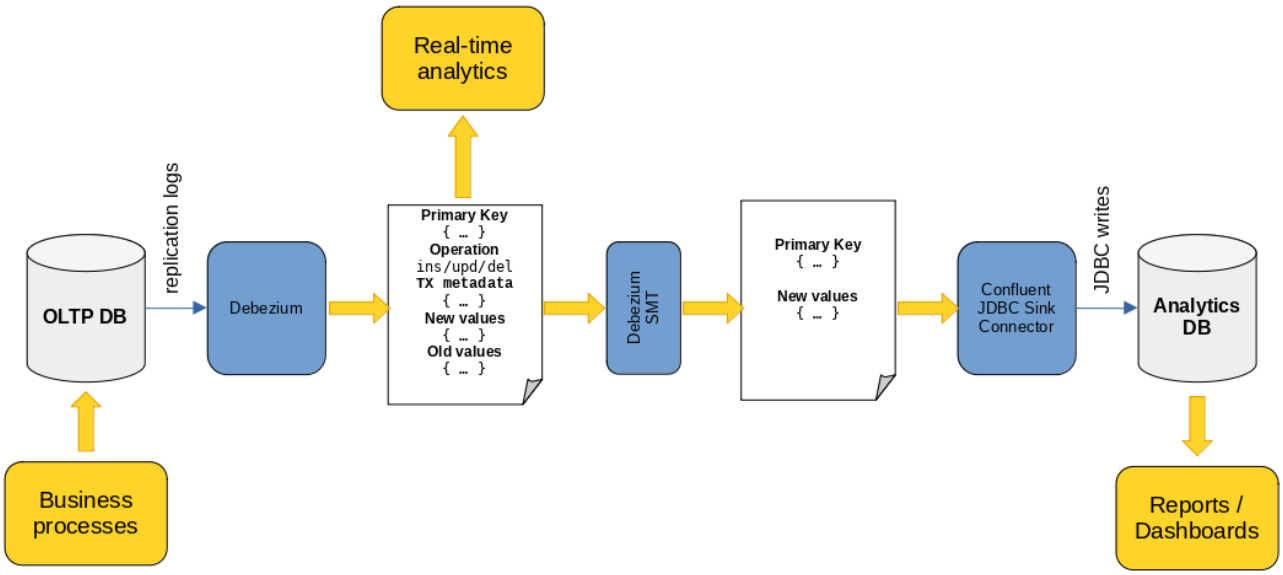
Unfortunately, there is no anti-Debezium to use at the sink side of the equation, so while we now have rich and detailed change events for real-time streaming processing, we still need to find a solution to write the change events to our target database.

COMBINING DEBEZIUM WITH THE CONFLUENT JDBC SINK CONNECTOR

The first thing we can try is to combine Debezium with the Confluent JDBC Sink Connector. The Debezium change event format is different from the one the JDBC Sink Connector expects, so we need a way to rewrite the events for consumption by the sink connector.

Debezium offers the necessary functionality out of the box, in the form of a [Kafka Connect Single Message Transform (SMT)](#).

Expanding on the conceptual diagram again, we end up with something like this:



This combination offers some improvements over the pure-Confluent solution:

- Change data capture is more efficient (hooking into the source database's replication system is wildly more efficient than periodically querying all tables).
- Real-time streaming processing of the change events has access to the richer Debezium event format.
- This solution supports DELETEs in the source database, provided the DELETE events happen in a table with a well-defined primary key. While the Confluent JDBC Source Connector can't detect or produce DELETE events, the Sink Connector knows how to deal with them if they appear on the change event topic.

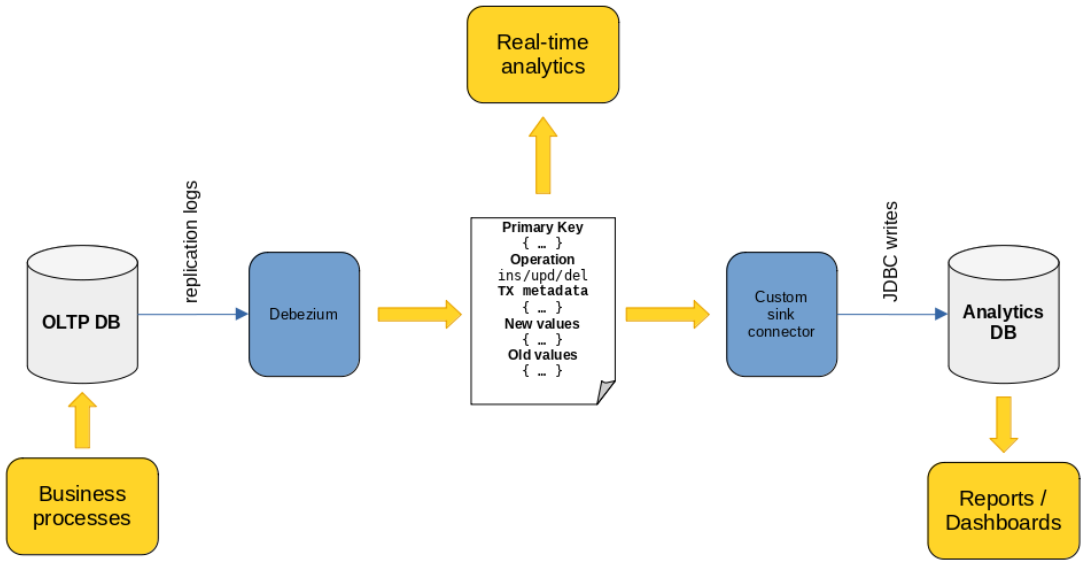
Still, we've not arrived at a fully generic solution. The following limitations still apply:

- There is no support for TRUNCATE events.
- For tables without a primary key, DELETE events are still unsupported.
- To cater for the limited type support in the JDBC Sink Connector, we need to configure Debezium to use only the standard Connect data types, so we still lose some precision in the captured data.
- Half of our solution still relies on the Confluent JDBC Connector, so the same licensing caveats as above apply.

COMBINING DEBEZIUM WITH A CUSTOM SINK CONNECTOR

If we want to build a fully generic solution that can handle all use cases, and is unencumbered by non-Open Source licenses, we'll have to complement Debezium with a custom-built sink connector. This will finally give us the holy grail of CDC/replication solutions: a pixel-perfect, efficient, fully open-source change capture and replication system over Kafka.

Let's expand our conceptual diagram one last time:



Luckily, we don't have to start from scratch to build the custom sink connector. The JDBC Sink connector provides us with a usable skeleton to build upon. We have a problem though: the aforementioned CCL license may not be suitable in all contexts where we want to use our solution.

The lovely folks at Aiven have us covered, though: they maintain a fork of the Confluent JDBC Connector, branched off from the last Apache 2.0 release done by Confluent. You can find it here.

Now that we have a base to build upon, let's review the changes we need to make.

SUPPORT FOR THE RICH DEBEZIUM EVENT FORMAT

The Debezium change record format is much richer than the JDBC Connector format. In particular, the presence of both pre- and post-change field values in the change event will help us deal with the disambiguation of UPDATE and DELETE events for tables without a primary key.

That means we can't use the Debezium SMT that drops the additional information anymore, and we'll have to rework the sink connector code to interpret the full Debezium event format instead.

SUPPORT DELETE AND TRUNCATE OPERATIONS

The Aiven JDBC Sink Connector was forked from the Confluent source before the latter implemented support for DELETE events. Hence, we have to re-implement DELETE support from scratch. Given that the Confluent DELETE support was limited to tables with a primary key, this from-scratch reimplementation gives us the opportunity to architect DELETE support for all tables, with or without a primary key.

TRUNCATE events weren't supported in either version of the JDBC Connector, so we have to add support for these as well.

SUPPORT UPDATE EVENTS FOR TABLES WITHOUT PRIMARY KEY

The Debezium change event format has sufficient information (pre- and post-event field values for the change row) to implement proper UPDATE support for tables without a primary key. We should re-architect the sink connector to make good use of this information to distinguish between INSERT and UPDATE events in primary-key-less tables.

SUPPORT FOR THE EXTENDED DEBEZIUM TYPE SYSTEM

We need to support the extended Debezium types that capture the full resolution of the database types.

ACCOUNT FOR THE DIFFERENCES IN TYPE SYSTEM BETWEEN SOURCE AND TARGET DATABASE

Not all SQL databases are created equal: there is a large common ground, but some types are handled differently. For example, the Oracle and PostgreSQL DATE types have a different resolution, and Oracle treats NULL string fields as equivalent to empty strings, whereas PostgreSQL doesn't. All databases also have their own extensions to the ANSI SQL standard.

Now that Debezium has given us an accurate representation of the data in the source tables, we'll have to deal with these incompatibilities more explicitly and write the appropriate conversion routines.

SUPPORT DIFFERENT NAMING CONVENTIONS

This is a rather trivial oddity, but one that caught us by surprise during our experimentation.

Table and field names in an SQL database are typically case-insensitive, so you can write FIELDNAME or fieldname in queries and refer to the same field. Internally, each database

translates the table and field names to a canonical form. For Oracle, that canonical form is ALL_UPPERCASE, whereas for PostgreSQL that canonical form is all_lowercase.

At the same time, both databases allow for a form of case sensitivity in table and field names, as long as those names are properly quoted. Take, for example, the following SQL statements:

```
CREATE TABLE foo (id INT PRIMARY KEY, data INT, “DaTa” INT);
INSERT INTO foo VALUES (1, 10, 100);
```

In both PostgreSQL and Oracle, the following queries yield identical results:

Query	Result
`SELECT data FROM foo WHERE id = 1;`	10
`SELECT DATA FROM foo WHERE id = 1;`	10
`SELECT DaTa FROM foo WHERE id = 1;`	10
`SELECT "DaTa" FROM foo WHERE id = 1;`	100

This mostly-case-insensitive nature of databases is a great convenience to humans, as it absolves us from thinking about proper capitalization most of the time. For an automated replication system, however, it makes life hard.

If the replication system discovers a source table field called DATA in an Oracle database, what should it expect the corresponding target table field in PostgreSQL to be called? Probably data, but maybe it’s “DATA”? Likewise, a source table field named data in PostgreSQL may correspond to DATA or “data” in an Oracle target table.

We can take a stab at solving this heuristically, assuming that whatever field names correspond with the canonical naming scheme for the source database must be translated to the canonical naming scheme for the target database, and everything else must be quoted. So an Oracle source field called DATA would be translated to a PostgreSQL target field named data, but an Oracle source field named DaTa would be translated to a PostgreSQL target field named “DaTa”. This is likely what you want in 99.9% of all cases. Until, of course, you bump into the one case where that is not what you wanted. Such is the curse of trying to build a generically applicable solution.

The most generic solution is to introduce an additional configuration option in the Sink connector that allows for explicit table and field name mappings. The default behavior should be the heuristic described above, and the new configuration option allows the user to tweak the behavior whenever needed.

CONCLUSION

It's definitely possible to build a combined Change Data Capture and replication system based on Kafka and Kafka Connect.

Depending on your use case, and the limitations you're willing to accept, you may even be able to build it from off-the-shelf components, with Confluent's JDBC Connector, potentially combined with Debezium for better performance and richer events.

To build a fully generic solution, though, that can handle all use cases (UPDATE and DELETE events for tables without a primary key, TRUNCATE events, precise and non-standard data types, ...), off-the-shelf projects won't cut it. Debezium has you covered on the source side of things, but on the sink side, we'll need to build our own solution. Luckily, there's a skeleton to build upon, in the form of the Aiven JDBC Sink Connector.

Do you have a need for such a system? Klarrio can help! Contact us at info@klarrio.com, or www.klarrio.com.

ABOUT THE AUTHOR

Dominique Chanet is Lead Architect at Klarrio.
Prior to that, he was a Software Architect in the CTO Office of Technicolor Connected Home, and member of the Technical Steering Committee of the AllSeen Alliance, which stewarded the open source AllJoyn IoT project. Dominique holds a PhD in Computer Science from Ghent University, Belgium.

Method of contact: info@klarrio.com

CONTACT US

BELGIUM
Tel: +32 (0)3 296 87 06
Email: info@klarrio.com

NETHERLANDS
Tel: +31 (0)10 313 25 24
Email: info.nl@klarrio.com

GERMANY
Tel: +49 2407 50 23 180
Email: info.de@klarrio.com

UNITED STATES
Tel: +1 919 649 2997
Email: info.usa@klarrio.com

AUSTRALIA/PACIFIC RIM
Tel: +61 402 850 059
Email: info.aus@klarrio.com

WWW.KLARRIO.COM

Klarrio is not only a recognized expert in real-time streaming data analysis and processing, we're also a team of cloud-native engineers and data scientists with years of experience leading the transformation of some of the most innovative organizations in the world.

We're known for our ability to build open source-based cloud-agnostic solutions that seamlessly process huge volumes of data. More importantly, we're also known for our ability to deliver valuable results and our willingness to tell you the truth from the very beginning.

Our goal is to help you take full advantage of the countless benefits streaming data can offer over traditional IT architectures. Start streaming ahead today.

GENERAL DISCLAIMER
The information in this document may contain predictive statement, including but not limited to, statements regarding data security, future financial results, operating results, and new technologies. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purposes only, and constitutes neither an offer nor a commitment. Klarrio may change the information at any time without notice, and is not responsible for any liabilities arising from your use of any of the information provided herein.

@klarr_io
@Klarrio
linkedin.com/company/klarrio

Klarrio
STREAMING AHEAD