

Apache Kafka

Why Apache Kafka?

Problem:

Lets Consider a typical retail system requirement. There are many POS systems in our store. If we have a multi store business, they are spread across the stores. If we are accepting online orders, our invoice events are also coming from our online web store.

Hence there are multiple sources for the same type of event. Similarly, there is no single destination for the invoice. The invoice is required by the finance, shipment, inventory planning, loyalty services, analytics, long-term storage and many more.

So how do we solve that many-to-many streaming requirement? Do we want to build all those streaming pipelines?



In the above diagram, each connection between a source and destination becomes a stream. It is practically impossible for the POS to send the invoice to twenty different services. Similarly, none of those target systems can handle the connections coming from hundreds of POS terminals.

Then how do we solve that kind of problem?

Anil
Ghuge
919850317294

Apache Kafka

Solution:

Databases as an Alternative

Let's try to evaluate the database as an alternative.



The above diagram, shows a possible solution for the given problem by using a database. We can place a database in between the producers and consumers and simplify the many-to-many relationship. Putting a database simplifies it to a many-to-one from the perspective of the event producers.

If we just look at a single producer, it is simply a one-to-one relation for each producer. Similarly, it

is one-to-many if we look at it from the event consumer side. However, when we look at an individual consumer, the read operation is a one-to-one relation.

The event producers such as POS terminals and our online web store, they all persist the JSON object into a database. The event consumers such as finance, shipment, inventory planning, loyalty services, analytics, etc. are sitting on the other side of the database.

They simply consume those events from the database. The producers and the consumers are decoupled, and we do not need to have a direct connection between producers and consumers. This solution appears to be a good alternative. **However, there are some challenges.**

Challenges with Database

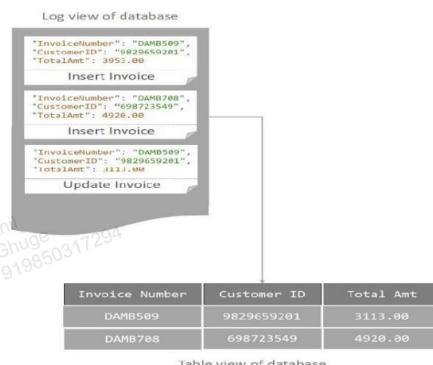
Using a database table to design a stream-of-events could be a tricky solution because a database table does not naturally represent a stream of events.

Let's try to understand what it means by saying "**Database table does not naturally represent a stream.**"

A typical database system records every insert and update transaction into a log file. These log entries are immutable records of all database changes that are recorded in sequential order.

Apache Kafka

This logging is necessary for the databases to handle failures. However, the tables are the aggregated and mutable materialized view of the log entries.



The above diagram shows a snapshot of a log file and an equivalent table. The log contains three events. The first and second events are new invoices, whereas, a third event is a change event. This change event represents an update operation in the first invoice.

Log file represents all three events. However, the table represents only two events. **The invoice change event is applied as an update operation on the earlier invoice.** From a database perspective, the table looks perfect. There are only two invoices. The third event updates the first invoice, and that's how a database table works.

However, if the event consumers are reading this table, they are likely to miss the change because they would be looking for new records, but there is no further record caused by the update event. Handling that problem is not a real big deal.

We may be able to handle that problem using some tricky design and application logic, but those things will make our solution complex and slow. That is what we mean when said, "**Database table does not naturally represent a stream.**"

There are many other reasons to reject a database for integrating real-time events among the applications. However, at this stage, instead of using a database table, we want to evaluate a solution that naturally represents an event stream.

Database Log Files

The log files are the most natural method to represent an event stream. The database already implements the notion of log files since decades. The streaming solution using a database offered a powerful capability.

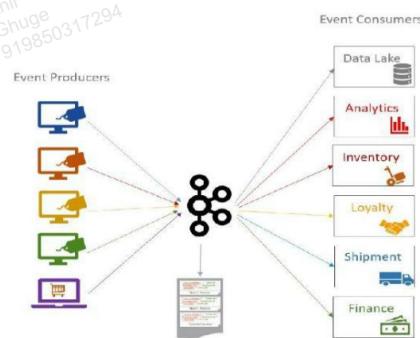
Apache Kafka

We were able to decouple event producers and event consumers. The producers and the consumers had no dependency on each other. The producer had to log the event into the database table, and that's it, they do not have to worry anything after that.

Similarly, the consumer reads from database tables and process those events. The solution appears to be an excellent option except for one fundamental problem. Instead of tables, we wanted logs to avoid missing track of updates. We wanted immutable log of events instead of mutable tables.

Enter the Kafka Apache Kafka

Apache Kafka takes that same idea of representing event streams using immutable log entries. So, if we take the earlier solution and replace the database system with Apache Kafka, we should get a new system represented



Kafka implements the log files and sits in the centre of the streaming infrastructure. The producers send the events to the Kafka, Kafka persists those events into the log files.

The event consumers are on the other side of Kafka. They read those events from the Kafka and process them. This solution delivers all the benefits that we discussed earlier such as decoupling of producers and consumers as well as simplifying the many-to-many relationship.

However, Apache Kafka needed a mechanism to write and read the log entries. The Kafka architects smartly discarded the notion of databases, and they did not implement Kafka as a shared database. That means, Kafka is not a database and it does not implement database access protocols such as JDBC/ODBC and native SQL. But still, they needed a read/write API for Apache Kafka. Apache Kafka decided to implement the Publish/Subscribe schematics of the messaging systems.

That is why a lot of people refer to Apache Kafka as a messaging system. However, Kafka was designed to be a powerful real-time streaming platform, and that's what it is.

Apache Kafka not only solves the core problem of transporting our events in real-time, but it does that in a **distributed, highly scalable and fault tolerant** manner to claim the leading position in real-time stream processing platforms.