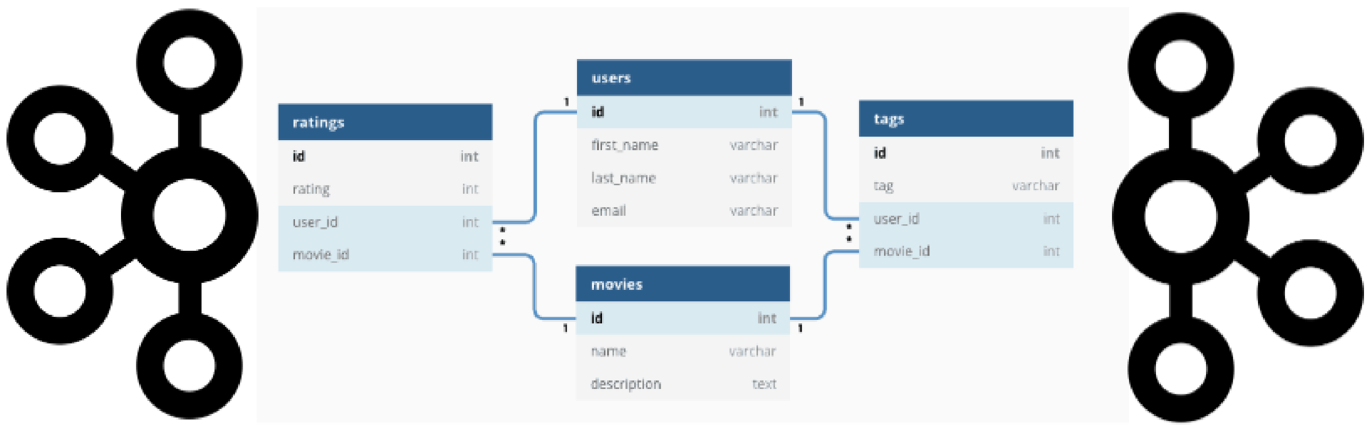
Transactional integration Kafka with database

Victor Alekseev Follow

Sep 21, 2020 · 15 min read



Introduction

More and more projects are now choosing Kafka as their messaging infrastructure. This technology's choice is not always driven by the real need to handle vast amounts of data with linear scalability. We can often hear about Kafka's application in the usual boring enterprise projects involving trivial document workflow management. The answering the question "why" in this case is not easy. Sometimes it can be caused by hype and impressive examples of Kafka's usage in notable projects. In other cases, it may be inspired by the opportunity to experiment at the expense of the customer's money, attracting him with advanced technologies.

If we use Kafka, for example, to analyze user activity on the high load web site, collecting all events regarding the client's behavior, we have an intensive data stream, but each event can be lost without problems for the business process as a whole. This approach does not apply to most enterprise applications, where we critically need a robust transaction integration between Kafka and the database.

Indeed, Kafka supports transactions, but it is only transactions between individual topics. We can atomically receive an incoming message, process it, send some outgoing messages, and reliably acknowledge at the same time the receipt of the original message. This approach works fine as long as we process only with Kafka data. Any attempt to involve any third-party data sources in the business process is stuck because the Kafka fundamentally does not support XA transactions.

In the most uncomplicated cases, we can store the instance state of any processing service locally and accompany each update by sending a message into a particular dedicated “archive” topic. This sending can be performed transactional with processing all other messages, so the data integrity does not suffer. After a crash restart, we can always restore this state by reading the whole message flow from this “archive” topic (“event sourcing” pattern, Kafka Streams uses a similar approach). Instance state can be implemented as in-memory structures, collection of files, and even local database (RockDB is often used for this).

However, this approach has a few significant drawbacks.

- First of all, in case of large data volume and complete data loss, the recovery process may be too long.
- Secondly, it is complicated to perform “joint” operations on data located in the different processing service instances. Any query operation has to be split into several ones with join received results after. It kills the scalability and may require an additional memory from the client for further data processing. The situation with update operations is even worth it because it will be quite complex to update several instances in a transactional manner.

In most cases, we want to continue to use the usual single database for all instances of the service and perform its update transactionally with sending/receiving Kafka messages. This article focuses on different approaches to implementing this requirement.

Sending messages

Let’s start by sending messages. We can do it non-transactionally and meet the following problems as a result:

- If we first send the database and then send messages, we risk that the recipient will never know about the update result.
- If we send a message first, then the recipient may request the updated data too early before the database transaction is committed. Also, the database update operation may end up with an error, and then the sender and the recipient of the message may be in an inconsistent state.

At first glance, it is possible to describe at least six fundamentally different approaches to reliable and consistent operational implementation.

Intermediate storage of messages in a database before sending

It is a standard and quite simple JEE patter:

- The database update operation creates messages and stores them in the database table — queue for dispatch
- A dedicated scheduler reads these messages from the queue, sends them out, and then deletes them.

In the case of the service instance's death, the maximum we risk is to send it and then receive some message twice. But in fact, the developers of any distributed systems should always be ready for such type of issues.

The application executes between one and two additional queries under the load for each sent message, depending on the dispatching strategy.

It is necessary to emphasize that, following this pattern, we don't use Kafka transactions at all. All messages are initially committed from Kafka's point of view.

Advantages:

- The most straight, transparent, and clear for all developers
- In the naive simplest case, it can be easily implemented ad-hoc based on "select for update skip locked" operation. This implementation will be about the same for all technologies and will not require complex infrastructure components.

- There are ready to use implementations, for example, Spring **JdbcMessageStore** or Yandex “DB-queue.”

Disadvantages:

- The considerable additional load on the database is permanent continuous polling even if the application does not perform any business activity, and messages are not sent at all.
- If messages are generated faster than sent out, the table accumulates more data and slows down the dispatching process even more. Thus, we have a system with negative feedback, which is not very good.
- From the sending process point of view, messages appear in the order of committing transactions, not in the order they are “sent” by the application by saving the record in the queue table. For example, lets one transaction sends messages at times T1, T3, and commits them at time T10. And another transaction does the same at times T2, T4, T5. In this case, messages will be really sent in the order of T2, T4… T1, T3.
- Keeping the order of the messages in the context of one partition requires having only one dedicated thread in the cluster at any given time, which sends out messages from this partition. Reliable implementation of such restriction is not simple too.
- Some correct, carefully tested, and ready to use implementation (such as **JdbcMessageStore**, for example) can be applied only together with specific infrastructure, which is heavyweight in itself.

As a result, this approach’s strictly correct implementation becomes quite complex and not scalable enough, especially under a high load. Those who wish can read the discussions on **JdbcMessageStore** performance problems on forums themselves. If we don’t especially care about the message order and high throughput isn’t critical, maybe Kafka isn’t needed too?

Kafka JDBC Connector

Kafka also suggests external for application implementation of the previous approach — JDBC Connector (Source and Sink). These modules allow transforming table records into messages and back. As already said, sending messages is implemented by polling the source database table with configurable interval.

The main advantage is that this function has already been implemented, and its integration with the already existing application requires minimum efforts from the developer.

An additional drawback to the above is the need to deploy other infrastructure within the cluster and administer it. Besides, this module's configuration options do not imply the possibility of saving the order of messages in the context of individual partitions, which makes the applicability of this approach somewhat limited.

In general, it seems that this approach is more a technology of integration of ready-made applications than a way to solve the problem.

"Change-Data-Capture" pattern on low level

It is also some modification of the first approach but without polling the database server by SQL queries. In this case, we install special software that can subscribe to system events, react to the new records in the corresponding table, and transform them into Kafka messages.

Now the next tools are available:

- **Databus** from Linkedin — only Oracle is supported
- **Debezium** — the most well-known solution, supports a lot of databases by plugins architecture
- **DBLog** from Netflix — it looks like not published as an open-source yet
- **Bottled Water** — it was popular, but not supported more

The list of advantages and disadvantages is about the same, but the main drawback of introducing the intermediate storage has been eliminated — the constant polling is absent.

Compositional transaction manager

This pattern is standard for the Spring ecosystem: **ChainedKafkaTransactionManager** based on **JpaTransactionManager** and **KafkaTransactionManager**, which in turn manages native Kafka transactions.

This approach can be considered as an implementation of the "1PC best effort pattern". This pattern's main idea is that changes of the most "stable" resource are committed with the last ones, significantly reducing the risk of both systems' inconsistent end state.

The main problem is in what order we unite transaction managers:

- Under load, the commit time of a database operation can be significant, and if we commit Kafka first, the customer receiving the message will not be able to request modified data
- If we commit Kafka after the database, there is a little chance that the service will die right in the middle of these two operations

Yes, the last probability is very low, and we usually neglect it. But if we multiply it by a large number of messages, we will get quite real losses, unacceptable, for example, during processing financial data. From the other point of view, if there are processing only a few messages, it might make sense to think again whether we need Kafka at all.

How can we eliminate the risk of loss? The trivial idea is to reliably keep the sending messages in a database and immediately delete them after the Kafka commit operation. The most straightforward implementation of this approach can be based on the next common components:

> Transactional scoped storage.

Initially, I tried to use the local thread variables, but after several unsuccessful attempts and a lot of quite complex code, I decided that Spring transactional context can be the best place for it.

Instead of variables, I added a third transaction manager into the **ChainedKafkaTransactionManager** configuration, which manages the **InfoHolder** object with the following information:

- **currentTx** — UUID of current Kafka transaction

- **commitError** — flag of Kafka commit error event
- **deep** — index of nested Kafka transaction
- **propagation** — propagation attribute of current Kafka transaction

> Special PseudoTransactionManager

ChainedKafkaTransactionManager calls it before and after all other transaction managers. Its only responsibility is providing access to the “last” **InfoHolder** instance with the maximum deep value — description of the last nested Kafka transaction. All instances of **InfoHolder** are stored in the **TransactionSynchronizationManager.resourceMap** collection and altered during transaction management related operations.

The other its logic is quite simple:

- on **doGetTransaction()** it creates the new **InfoHolder** object with the **deep = lastInfoHolder.deep + 1** (or 1, if it is the first instance). This code is executed before both other transaction managers.
- on **doBegin()** it receives the **InfoHolder** instance as a value of **transaction** parameter and register it in the **TransactionSynchronizationManager.resourceMap** collection by calling of **TransactionSynchronizationManager.bindResource()** function. Depending on the value of the **propagation** attribute of the **TransactionDefinition** parameter, it copies the value of **currentTx** property from the last **InfoHolder** instance to the current one or generates the new one (for example, if **propagation== REQUIRES_NEW**)
- on **doCleanupAfterCompletion()** it reads the last **InfoHolder** object and removes all messages from database with corresponding **currentTx** attribute, but only if **(deep == 0 or propagation == REQUIRES_NEW) and commitError == false** (the real expression is more complex, but for the illustration it is enough). Also, it performs **unbindResource()** operation and throws out the last **InfoHolder** object from the **TransactionSynchronizationManager** scope. This code is executed after both other transaction managers.

> Kafka ProducerInterceptor

- Its primary responsibility is saving any messages in the database with **currentTx** attribute, which can be read from the last **InfoHolder** object, provided by the **TransactionSynchronizationManager.resourceMap** collection.
- Sent messages are available as **ProducerRecord** instances and contain all necessary information for sending them again in case of any issues during commit.
- It is also necessary to take into account that Kafka ignores all exceptions from this code. So, if we need to use them, the only way is to save it in the last **InfoHolder** object.

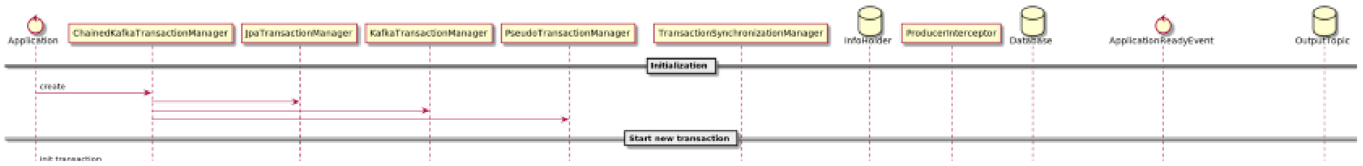
> The last component of this construction is slightly customized **KafkaTransactionManager.** On **doCommit()** method it:

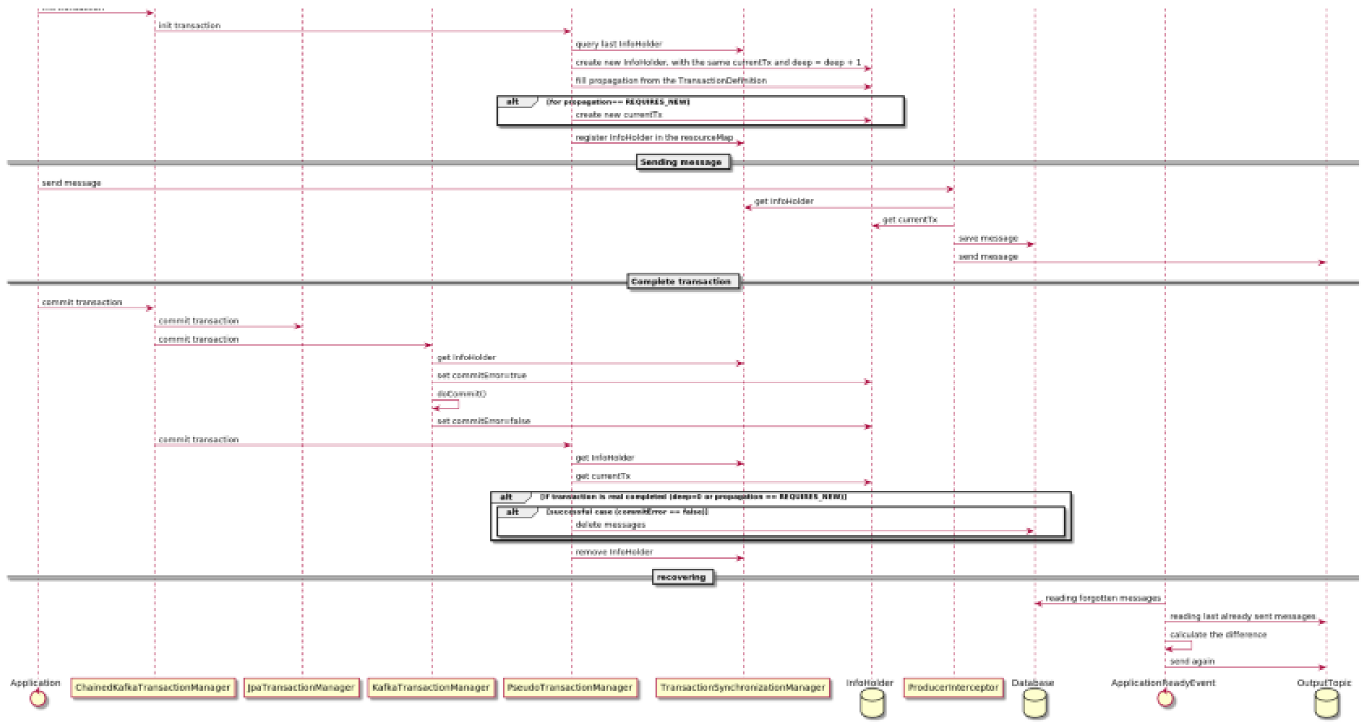
- gets last **InfoHolder** objects from **TransactionSynchronizationManager** and set its **commitError** attribute to true
- executes original **doCommit()** method
- set **commitError** attribute back to false

The last component is the listener of the **ApplicationReadyEvent**, which reads all "forgotten" messages from the table and sends them again while restarting the service after death. To minimize the count of the sent duplicates, it is possible to perform the next operations:

- Wait for some time until the transactions started while the service instance was unavailable are finished.
- Read the latest messages from the outgoing topic and delete the corresponding entries from the database table.

The following diagram is intended to illustrate the interaction described above:





Advantages:

- No constant database polling at all.
- The number of SQL requests can be minimized to 2 per transaction (by batching messages in the **InfoHolder** objects).
- The processes of sending messages and deleting successfully committed messages cannot “lag” behind each other.
- The table’s size for the temporary storage of messages “in the process of committing” is always small, and operations are performed very quickly.
- Implementation is compact and clear enough.

Disadvantages

- For each platform (Spring / JEE, for example), we need to implement it in a specific way.
- Due to the many different possible combinations of individual transactions, it is challenging to thoroughly test this code. After writing a few dozen tests, I realized that this process is truly endless.

- Any transactions related code is difficult to debug due to timeouts.
- Advanced frameworks like Camunda love their customizations at a given place, and some even try to transplant the transaction's context between threads (I have done so myself more than once). Nobody guarantees that your code keeps being compatible with such tricks.

Although I have managed to make this method work reliably, on the whole, it seems to be quite fragile and requires extensive and expensive testing. So, I can recommend it only for big long projects.

The usage of a destination topic as proof of delivery

This general idea of this approach is about the same:

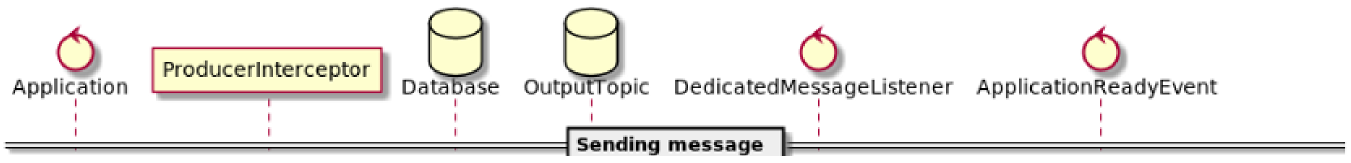
- Introduction **ChainedKafkaTransactionManager** as a combination of **JpaTransactionManager** and **KafkaTransactionManager.**
- Reliable saving Kafka messages during a database transaction.
- Removing that after the Kafka transaction's successful commit.

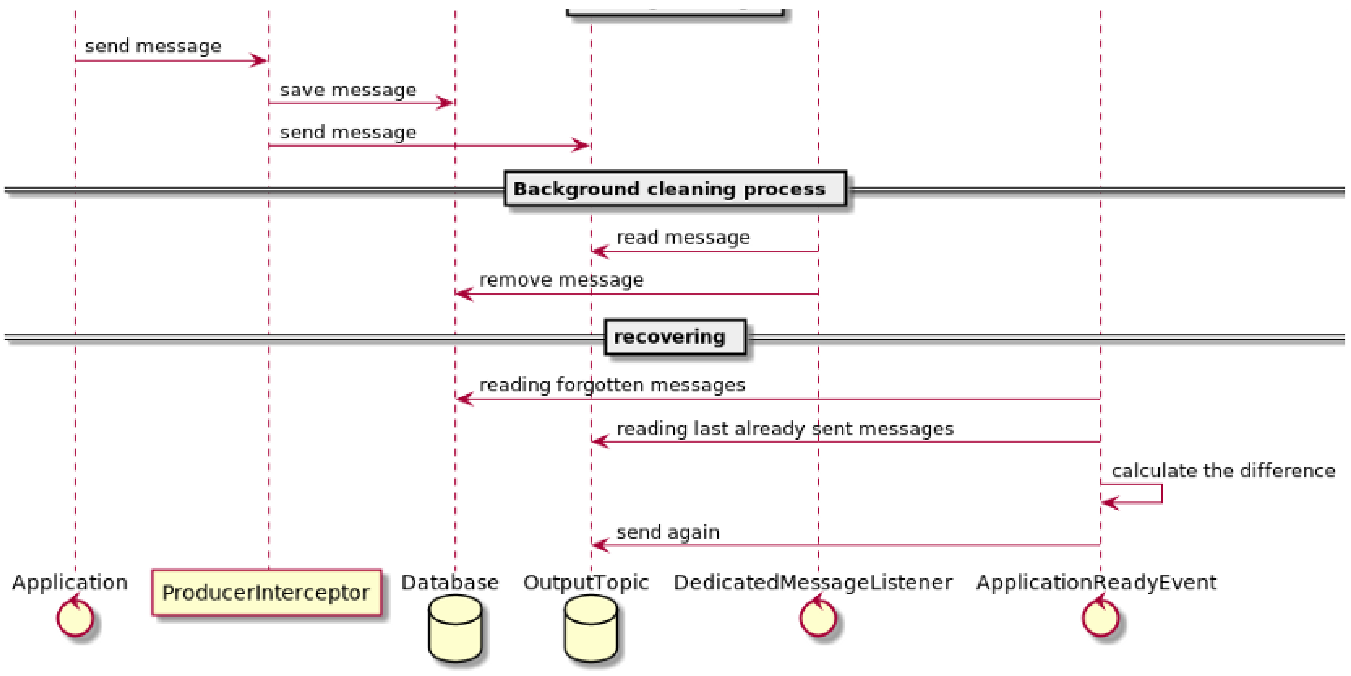
The only difference is the method of removing:

- Each message contains the primary key value of the corresponding database record.
- A dedicated system message listener receives all messages from the output topic and deletes related records from the database. It can read them from Kafka, acknowledge immediately, and remove entire data sets from the database for optimization purposes.

The recovery procedure is about the same, but due to a more significant lag between saving and removing database records, it is quite preferable to read the output topic before sending forgotten messages again.

The following diagram is intended to illustrate the interaction described above:





Advantages:

- Most logical, direct, and transparent in terms of distributed log technology
- Minimal invasion of system services and components
- Due to simplicity, it is exceptionally resilient to bugs and errors.
- No constant database polling at all, completely event-driven process
- In many cases can be easily implemented without any **ChainedKafkaTransactionManager** at all

Disadvantages

- Most expensive in terms of resource consumption (between one and two SQL queries per message, connections with brokers, back network traffic, listener threads, and so on)
- The significant lag between saving and removing database records, more data in the table

With some small corrections, this approach can be applied to entirely non-transactional services, such as mail servers, for example.

Combination Kafka with database TX transaction

There is such a transactional pattern as "Last Resource Gambit Optimisation (LRCO). " This approach means a non-XA capable resource may be (relatively) safely included in a two-phase commit transaction containing multiple resources.

Importantly: The transaction must be 2-phase (typically XA). All but one of the resources must be XA capable.

As far as I can see from the documentation, a lot of transaction managers supports this type of optimization, for example, JBoss provided and Atomikos

It looks like the most seamless solution, of course, if the database supports XA transactions. Unfortunately, Atomikos, for instance, out of the box, contains the implementation of this pattern only for JDBC data sources and not aware of Kafka. I could find any information about the successful implementation of this approach.

Business-driven approach

This approach is a combination of recommended by default **ChainedKafkaTransactionManager** with taking into account some business requirements and reasonable trade-offs.

First of all, I suggest measuring the frequency of lost messages under the reliable load and emulation of infrastructure problems. If you can miss a maximum of ten messages per year and the average sum of each transaction is about 100 euros, it means that investing a lot of time and efforts in the solution of the problem described above has no sense. Forget and continue to solve business tasks to increase the maximum possible damage in the future.

Secondly, if you do serious business, I suggest that you already have some SLA set, monitoring system, and instruments to fix problems with "stuck in the middle" business entities. The existence of such tools implies that individual business operations are already implemented in an impotent way. Therefore, the loss of the message will inevitably be detected by the monitoring, after which you will only have to request the last operation to be executed again. If you do not have such a monitoring system, you have a much more severe problem to solve than the one described in this article.

Third, the more complex the code is designed to minimize possible losses, the more errors it can contain inside. The more time it will take to develop, debug, and fix bugs. The more memory and processor time it will consume. Besides, we should not forget that incidents due to these bugs will also cost money.

In general, this reasoning should have been placed at the beginning of the article, but then no one would have read it further, and I would have no reason to display my in-depth knowledge.

Receiving messages

This situation is much easier: you can complete database transactions and acknowledge the Kafka message after even without the Kafka transactional mode.

If shit happens, the maximum problem is receiving the same message twice. Preventing it is not a significant problem (any case, in a distributed architecture, we must do so one way or another):

- Inserting the message's unique key (technical = topic name + partition index + offset or business, that may be better) into the dedicated table with a unique index. We can perform it immediately at the beginning of the process, and for duplicate database transaction will be immediately rollbacked.
- This table can be cleaned periodically not to lose performance as you hardly expect to receive duplicates at intervals of more than a week. However, everything certainly depends on the business process.
- Also, it is necessary to take into account that this table is high concurrent storage. So, we need to optimize it on the database level (for Oracle, for example, by INITRANS option), and by sharding (several separate tables, to each of them, are sent the keys with the corresponding remaining from the division of the hash key).

This schema is more or less optimal if the processing of each message requires the execution of the database-related operation. In this case, I/O throughput is limited by the database, and one additional query per message does not change the situation a lot. Also, most of the time, we need not only rejecting duplicates but also providing service's idempotency by returning the same value as before. This value has to be stored somewhere, and it is more reasonable to do it together with the unique message key. In the case of enterprise applications, this is roughly 99.99% of all use cases.

But what if you need to update the database only from time to time? For some optimization most straightforward approach, we can apply the Bloom filter — space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, to test whether an element is a set member.

From the implementation point of view, it is only a bit array. The only nuance is that it is falsely positive. By setting the probability of a false positive decision, which is acceptable from the business point of view, and the number of controlled elements, you can get the array's necessary size.

Thus an optimized implementation of to duplicate filtering may look like this:

- The state of the filter is stored in the database separately for each partition.
- The application uses **ConsumerRebalanceListener** for getting events about the rebalancing process.
- After assigning the partition, the application loads the state of the corresponding filter, after revoking — saves ones together with the result of data processing, including the last offset.
- The application applies the message's unique key to the filter and thus identifies the duplicate on receiving each message. Yes, our filter can return the false-positive results, and we can lose some message, but we can control the target frequency of this "shit-happens" and make the possible loss acceptable.
- Periodically the applications save the result of processing in the database, including the filter's state and the last offset.
- after a restart from the death, the application reads the previous state (filter + result of processing) from the database, applies offset, and continues the work

The Bloom filter can be applied in the previous case too to minimize the count of SQL queries. But it is reasonable only when we want to use quite a big time window (it results in a giant table) and don't care about idempotency. In this case, we firstly use the filter, and only in case of positive answer validate it with an SQL query.

Conclusion

Kafka is not the best solution for dealing with transactional use cases inside enterprise applications. If you are not Facebook and do not intend to develop a world scale payment system, you probably do not need it. XA transactions joined JMS and database, are able to satisfy all your requirements fully.

Nevertheless, it can imagine that Kafka can be useful in implementing CQRS / Event Sourcing patterns, also providing backup storage, and facilitating the introduction of new types of services.

When implementing a transactional interaction between Kafka and the database, it is recommended to follow the most straightforward approaches and not chase the absolute precision of implementation. Often, a compromise approach will minimize possible losses much better.

For sending messages, I can recommend an approach with standard **ChainedKafkaTransactionManager** without any additional customization. If you are paranoic, you can use a destination topic as proof of delivery. The best method for receiving messages is the dedicated table for storage received message's keys and answers.

And the final general advice: test, measure, and estimate the possible losses in terms of an amount of money. Do not try to minimize the cost of operating on a spherical horse in a vacuum.