We want to populate data continuously to the Kafka topic mentioned in the previous section. How can we achieve this? You can use the application implemented previously or make suggestions using alternative approaches.

We can use the “last_updated_at” column to perform incremental loads. On each run, we update the last processed timestamp and fetch only the records that changed after that point. However, this approach only captures inserts and updates. If a hard delete happens, the row is physically removed from the table, so we won’t detect that delete event at all.

A trigger-based approach is another option. We can create a trigger on the Package table to capture INSERT, UPDATE, and DELETE operations. This trigger writes both the operation type and the affected record into a separate “change_log” table. The application periodically reads this table, processes the new records. Processed records can be flagged using fields like is_transferred or last_updated_at, and periodically deleted to manage table growth. This approach ensures that no changes are missed, but the additional overhead of triggers may lead to performance issues under high traffic.

All INSERT, UPDATE, and DELETE operations in the database are written to database-specific transaction logs, such as the WAL (Write-Ahead Log) in PostgreSQL. CDC tools that can read these logs (for example, Debezium or Oracle GoldenGate) can capture changes in real time without adding extra load to the application or querying the table directly.

After the CDC tool sends the database changes to Kafka, stream processing engines like Apache Spark Streaming or Apache Flink can be used to perform real-time processing and analytics on this continuous data flow.

In this work, instead of using traditional polling or trigger-based methods to continuously send data to Kafka, I used a more modern and scalable approach. All INSERT/UPDATE/DELETE operations occurring in PostgreSQL are captured by Debezium through the transaction log and written in real time to the **kafka_case.public.package** topic. Debezium’s ability to perform an initial load, track only selected columns, and apply filters makes the architecture very flexible and efficient. After that, my Java application consumes this CDC topic, transforms the record, and sends the output in MappedPackage format to the **mapped_package_cdc** topic. This way, database changes are immediately streamed to Kafka, processed inside the application and continuously delivered to a second topic in their transformed form.

Architecture of the my solution