TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Stanislav Matšel 202869IADB

Atomic Database Updates and Messages Publishing in Microservice Architecture

Bachelor's thesis

Supervisor: Aleksei Talisainen
MSc

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Stanislav Matšel 202869IADB

Jagamatu andmebaasi uuendused ja sõnumite avaldamine mikroteenuste arhitektuuris

Bakalaureusetöö

Juhendaja: Aleksei Talisainen
Magistrikraad

Tallinn 2023

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Stanislav Matšel

28.02.2023

Abstract

The aim of the thesis was to implement atomic database updates and event publishing in scope of existing application and provide description of work which was done to fulfil requirements.

The analysis evaluates technologies of existing application and theoretical approaches which can be used for problem-solving. Proof of concept was developed using information of analysis.

Solution described in proof of the concept was evaluated and delivered to production environment. During solution operation was found places for initial solution improvement, improvements was implemented and delivered to production environment as well.

The solution metrics was collected and analysed which gives an overview of solution performance.

The result of the thesis is a working solution using outbox pattern with transaction log tailing on production environment.

The thesis is written in English and is 46 pages long, including 7 chapters, 17 figures and 9 tables.

Annotatsioon

Jagamatu andmebaasi uuendused ja sõnumite avaldamine mikroteenuste arhitektuuris

Lõputöö eesmärk on rakendada jagamatu andmebaasi uuendused ja sõnumite avaldamine olemasoleva rakenduse raames ning kirjeldada, kuidas see oli tehtud et täita nõuded.
Töö käigus on tehtud olemasolevate rakenduste tehnoloogiate ning antud probleemi teoreetilisi lähenemisviiside analüüs. Analüüsi tulemused olid kasutatud, et välja töötada lahenduse kontseptsiooni ning tõendada antud lahendus võib kasutatud probleemi lahendamiseks.
Lahendus kirjeldatud kontseptsioonis oli testitud ning pärast testimist rakendatud tootmiskeskkonnas. Antud lahenduse kasutamise käigus oli kogutud info, mis võib algses lahenduses paremaks tegema. Lahenduses oli tehtud vajalikud täiustused, testitud ja rakendatud tootmiskeskkonnas.
Töö käigus oli kogutud ja analüüsitud lahenduse meetrikat, mis annavad ülevaade lahenduse jõudluse kohta.
Töö tulemus on töötav lahendus tootmiskeskkonnas, mis kasutab väljamineku mustri andmebaasi tehingulogi sabaga.
Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 46 leheküljel, 7 peatükki, 17 joonist, 9 tabelit.

List of abbreviations and terms

ActiveMQ	An open source, multi-protocol, Java-based message broker.
Apache Avro	A language independent, schema-based data serialization library
Apache Kafka	An event streaming platform
CDC	Change data capture
Cassandra	An open-source NoSQL distributed database
CockroachDB	A distributed database with standard SQL for cloud applications
Commit log	A log that records all transactions and the database modifications made by each transaction, same as transaction log
Database publication	A group of tables whose data changes are intended to be replicated through logical replication
Database transaction	A fundamental concept of all database systems that it bundles multiple steps into a single, all-or-nothing operation
Database trigger	A special stored procedure that is run when specific actions occur within a database
Db2	A family of database management system (DBMS) products from IBM that serve several different operating system (OS) platforms.
Debezium	An open-source distributed platform for change data capture
DynamoDB	A fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale developed by Amazon
JVM	Java virtual machine
Java	General-purpose, high-level, portable, object-oriented, interpreted language
Java virtual machine	A virtual machine that enables a computer load, verify, and execute Java bytecode.
Kafka Connect	A free, open-source component of Apache Kafka that works as a centralized data hub for simple data integration between databases, key-value stores, search indexes, and file systems
Kafka connect source connector	A ready-to-use component that can collect data from external systems into Kafka topics
Kafka event	A record of a state change in the system

Kafka event header	A part of Kafka event that contains metadata about it
Kafka event key	A part of Kafka even that represent key of event
Kafka event payload	A part of Kafka event that represent value of event
Kafka topic	A part of Apache Kafka which is used for organizing and storing events
LinkedIn Databus	A source-agnostic distributed change data capture system, which is an integral part of LinkedIn's data processing pipeline
Log mining	A process of reading the transaction log entry and publish change as a message-to-message broker
Message broker	A software that enables applications, systems, and services to communicate with each other and exchange information
Message relay	A process that sends the messages stored in the outbox to the message broker
Microservice	Software architectural style that structures an application as a collection of services which is responsible you own domain part
MongoDB	A document database
Oracle Database	A multi-model database management developed Oracle Corporation
POC	Proof of concept
Pooling publisher	A process of publishing events by polling the database outbox table
PostgresSQL	A powerful, open-source object-relational database system
RabbitMQ	Open-source message broker
Reactive Spring	A framework that enables to write asynchronous, nonblocking architecture
Replication slot	A stream of changes that can be replayed to a database client in the order they were made on the origin server
Retention policy	A policy that describes how long to keep a piece of information, where information stored and how to dispose it when its time
SMT	Single Message Transformation
SQL Server	Relational database management system developed by Microsoft
Spanner	Distributed SQL database management and storage service developed by Google
Spring Cloud	A framework that provides implementation of common patterns in distributed systems
Spring Framework	An application framework and inversion of control container
Transactional log	A log that records all transactions and the database modifications made by each transaction, same as commit log

Transactional log tailing	A software architectural pattern that describes idea of tailing the database transaction log and publishing each change as an event
Transactional outbox	A software architectural pattern that describes an approach to execute multiple tasks atomically
UML	Unified Modeling Language
Vitess	A database solution for deploying, scaling, and managing large clusters of open-source database instance
WAL	Write-Ahead Logging
Write Ahead Log	A standard method for ensuring data integrity in database systems

Table of contents

List of figures

List of tables

1 Introduction

In recent years microservice architecture has become a new trend in software development. It brings a lot of benefits to businesses, such as agility and cost reduction via granularity and reusability [1].

Microservice benefits are indisputable, but such architecture brings a lot of challenges for developers such as complexity of distributed system, difficulties to refactor across boundaries, testing service dependencies, data integrity, network congestion and latency, careful management of features that span multiple services and other [2], [3].

Solving problems of data integrity can be quite challenging for developers to keep data consistency. As we know, different microservices are responsible for different domain parts, but very frequently one microservice should know what happens in another too. There are different solutions for it - synchronous and asynchronous communication. Asynchronous communication happens via message brokers such as Kafka or RabbitMQ. It is usual to have some transaction in scope of one microservice, and when it is fulfilled, it should tell other microservices that it is done. But the transaction can be rolled back, then message should not be sent to message broker, it means sending messages to message broker should be part of the transaction. It is common to use asynchronous communication between microservices via message broker. Thus, the problem of atomic database updating and sending messages arises in microservice architecture. The problem needs to be solved to avoid data inconsistency and bugs.

Our development team faced the same problem of reliability database updating and messaging sending. The task of resolving the problem was set before me. It was quite challenging in scope of the given application and technology stack to implement proper atomic database updates and messages publishing to message broker that guarantee data consistency and synchronization between different microservices.

The thesis will give an overview of how theoretical approach can be implemented in scope of given application with given technology stack which can be useful for future cases.

2 Analysis of technologies used in application

A critical part of problem solving in scope of given application is analysis of existing technology stack. As we know technology stack can bring additional problems or limitation during problem solving. So, it is important to have analysis of existing technology stack for software engineer that provide better problem solution which is suitable for concrete application. This chapter brings overview of technologies which was used in application before proposal of the solution.

2.1 Java programming language

Java is general-purpose, high-level, portable, object-oriented, interpreted language. Java API was designed to be backward compatible that allow upgrade JVM version without updating language [4], [5].

There were economic and technical factors that led to use Java as the primary programming language for the application. Java platform provides APIs for general-purpose computing, which will be complemented with rich set of libraries for different purposes [6], [7].

2.2 Spring Framework

Spring Framework is an open-source application development framework designed for Java enterprise application. Spring handles infrastructure and developers can focus on application development. The framework has good support for building microservices in cloud. Spring Boot with Spring Cloud build good foundation for quickly launch application in cloud [8], [9], [10].

2.3 Reactive Spring

The casino platform should be designed to have low-latency and high-throughput loads. One of the main reasons why Spring reactive stack was chosen for the project. Spring reactive stack promises to bring performance and better resource utilization to the project

[11]. Reactive stack promises to bring a lot advantages, but it also creates additional complexity of code design. It changes the approach to how to write code – from object-oriented to functional programming, so the level of the development team should be more professional.

2.4 PostgresSQL database

PostgresSQL is a powerful object-relational database. PostgreSQL offers proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open-source community. It comes with a rich feature set and will be a good choice for an application which will use a relation database [12], [13].
PostgresSQL has some specific features such as Write-Ahead Log which should be considered during atomic database updates and messages publishing implementation.

2.5 Apache Kafka

Apache Kafka is an event streaming platform. Apache Kafka can be used for messaging, web site activity tracking, metrics, log aggregation, stream processing, event sourcing and log commit. In messaging domain Kafka is comparable to traditional messaging systems such as ActiveMQ or RabbitMQ, but has better throughput, built-in partitioning, replication, and fault-tolerance [14], [15].

2.6 Apache Avro

Apache Avro is a language independent data serialization system. Apache Avro uses JSON for specifying the data structure. Avro serialization and deserialization process allow use of two types of encodings: binary and JSON. Binary encoding is preferable because Avro does not include type and fields name during data serialization in this serialized data is small and can be delivered faster, but during deserialization process consumer needs deserialize data using Avro Schema [16], [17], [18].
In this project Avro Schema was associated with Kafka topics. Applying Avro Schema to Kafka topic gives several advantages. One of the biggest advantages is that an event can be validated before sending it to topic, so it can be guaranteed that only events with correct structure persist in topic. It also allows manage schema evolution via choosing

right compatibility type, so it protects downstream data consumers from malformed data [19], [20].

Avro Schema presence on topic can create additional complexity during atomic database updates and messages publishing implementation, because message should be serialized to given format before sending it to Kafka.

3 Analysis of transactional messaging patterns

This chapter brings overview of the problem and analysis of theoretical approaches which can be used for problem solving.

3.1 The problem

Microservices need to update the database and send messages/events to message queue. The problem of atomic updating the database and sending messages can take place and should be solved to avoid bugs and data inconsistency.

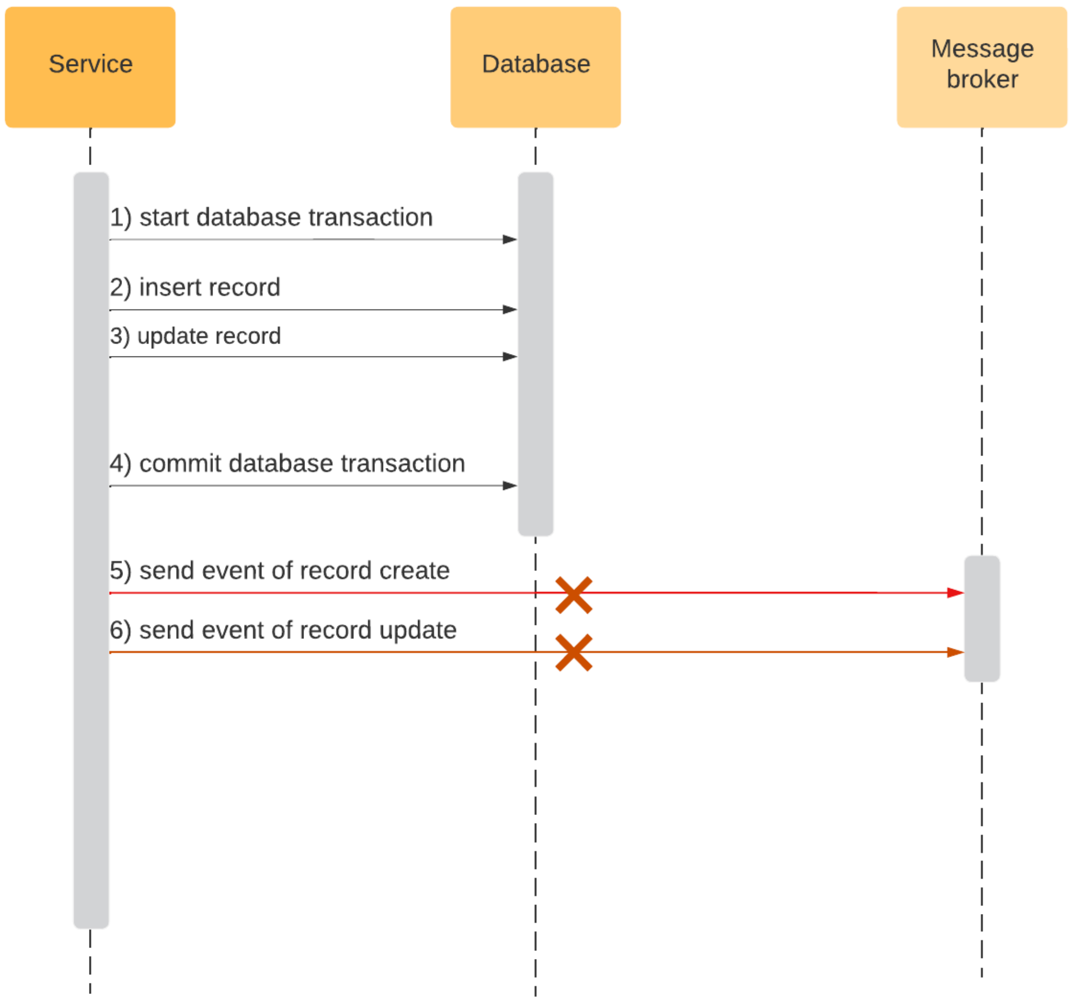
Service
Database
Message broker
1) start database transaction
2) insert record
3) update record
4) commit database transaction
5) send event of record create
6) send event of record update

Figure 1. Sequence diagram of nonatomic event publishing

As we can observe from sequence diagram (Figure 1) problem of sending event can occur in step 5 and 6 when service tries to send message to message broker. Root case can be broker failure or network issues.
There are different approaches to how this problem will be solved. This chapter brings overview of pattern are designed to solve the problem.

3.2 Publish events using database trigger

Atomic database updates and event/messages publishing can be achieved by using database triggers to insert record representing event into outbox table. A separate process will read records to send them to message broker [21]. This approach has one biggest disadvantage that logic of event publishing will be represented in database trigger, so this approach not considered as solution. Practical example of this approach can be found in [22].

3.3 Transactional outbox

Transactional outbox pattern uses the idea to insert records representing the event into additional outbox table inside same transaction with entity manipulation. In this way the service outboxes new events which should be sent to message broker. A separate process named message relay monitors changes in outbox table and publishes new events to message queue.
Transactional outbox pattern needs implementation of message relay. There are two common patterns how implement message relay:

- Polling publisher
- Transactional log tailing

Message relay with pooling publisher pattern periodically query outbox table to pull unpublished events, publish them to message broker and deletes published events from outbox table [2].
Pooling publisher can be implemented using the needed database driver and execute suitable query for fetching new messages for outbox table.

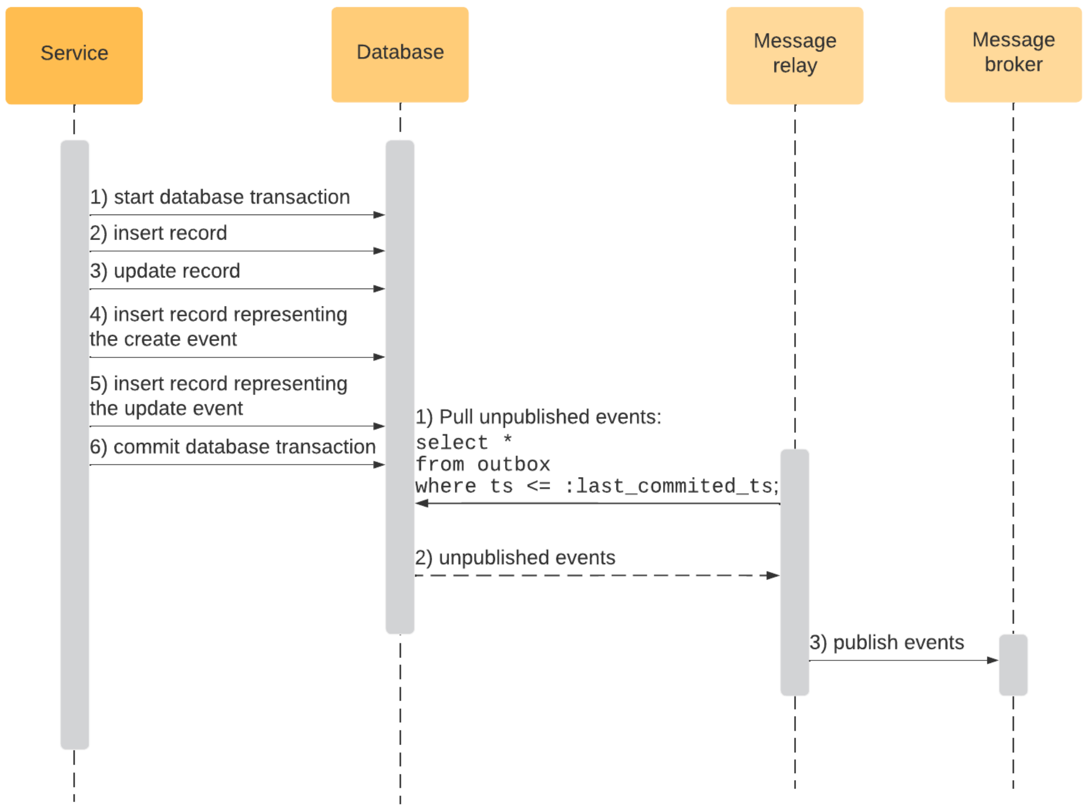
Service
Database
Message relay
Message broker
1) start database transaction
2) insert record
3) update record
4) insert record representing the create event
5) insert record representing the update event
6) commit database transaction
1) Pull unpublished events:
select *
from outbox
where ts <= :last_commited_ts;
2) unpublished events
3) publish events

Figure 2. Sequence diagram of transactional outbox with polling publisher message relay

Message relay with transactional log tailing pattern uses transaction log miner which is responsible retrieving changes in database using database transactional log or commit log.

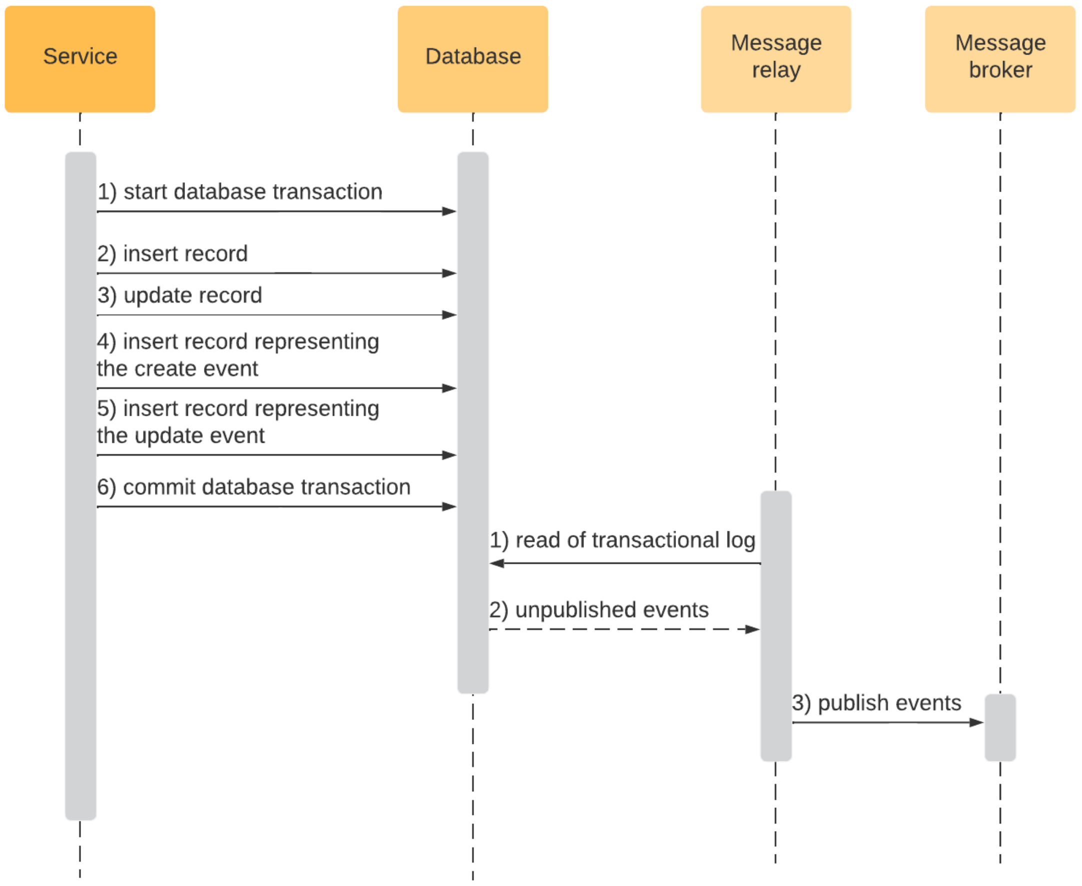
Service
Database
Message relay
Message broker
1) start database transaction
2) insert record
3) update record
4) insert record representing the create event
5) insert record representing the update event
6) commit database transaction
1) read of transactional log
2) unpublished events
3) publish events

Figure 3.Sequence diagram of transactional outbox with transactional log tailing message relay

It is simple to conclude using Table 1 that transactional log tailing message relay has more advantages and less disadvantages than pooling publisher message relay [2].

Table 1. Advantages and disadvantages of different message relay pattern

	Pooling publisher	Transactional log tailing
Pros	Simple approach, easy to implement Well on low scale	Well on high scale Implementation from scratch is complex Can be framework dependent No change required at the application level
Cons	Frequently pooling is an expensive operation Query pattern for NoSQL database is complex Bad on high scale	Hard to understand

Depending on database here is different technologies that can be used for transactional log mining.

Debezium is an open-source distributed platform for change data capture which can be used for MongoDB, MySQL, PostgresSQL, SQL Server, Oracle, Db2, Cassandra, Vitess, Spanner databases [23].

Some databases like CockroachDB, DynamoDB and SQL Server databases offer built-in CDC [24], [25], [26].

There are also database specific solutions like LinkedIn Databus for Oracle [27].

4 Proof of the concept

Previous chapters bring an overview of the problem, the application technology stack, the patterns of problem solving and technologies which can be used for the problem solving. This chapter will describe how problem can be solved using given information.

4.1 Choosing pattern and tools

Using information from chapter 3 about different approaches to implement atomic database updates and messages publishing it is decided to use transactional outbox pattern with transactional log tailing. As we can see from chapter 3 transactional outbox with transactional log tailing better on high scale and less expensive than polling publisher.
Information from chapter 2 should be used to choose the right tool. As we can see from the chapter, the existing application has PostgreSQL database which does not have built-in CDC functionality. PostgresSQL has one important feature Write-Ahead Logging [28]. It is possible to implement WAL miner using logical replication to capture changes from scratch, but it is complex and there are tools like Debezium which already solve this problem [29]. So, it was decided to use Debezium for these purposes.
Debezium can be embedded to any Java or Spring application, it gives more control, but adds one more component which should be developed [30]. Main task of this component is to read changes from database outbox table using Debezium and publish it to corresponding Kafka topic. Here is another tool which can be used to solve this problem – Kafka Connect. Kafka Connect is a tool for scalable and reliable streaming data between Apache Kafka and other data systems [31]. So, Kafka Connect connector can play role of message relay in transactional outbox pattern. Kafka Connect connector can be configured to use Debezium connector for PostgreSQL [32], [33].
As we can see from chapter 2 Avro schema with backward compatibility was applied to Kafka topic. It is possible to create a new topic and switch all consumers to a new topic, but it is quite expensive to switch all microservices to a new topic. So, it was decided to keep event structure as it is. The presence of a schema registry creates additional difficulties for outbox pattern implementation. The outbox table should contain a column

with Avro byte array serialized payload which should be published to Kafka topic. Also, some events in some topics had applied additional message headers which were not related to stored entities. All these characteristics of existing Kafka events should be considered during the implementation.

4.2 Architecture of the solution

4.2.1 Design of outbox table

Take into consideration all the requirements needed for Kafka event in topic final outbox table should contain information about event key, event payload and event headers. It is also important to have unique message id which can be included to headers for independent consumers by detecting duplicate events. The final design of outbox table shown in Table 2.

Table 2. Outbox table

Column	**Type**	**Modifiers**
id	uuid	not null primary key
topic	varchar (255)	not null
created_at	timestamp with time zone	not null default now()
event_key	varchar (255)	not null
event_payload	bytea	not null
event_payload_type	varchar (255)	not null
event_headers	varchar	

Table 3 gives an overview of the information that will be stored inside each column. The data from the table will be used by Kafka Connect source connector.

Table 3. Semantics of outbox table

Column	**Semantics**
id	unique id of each message
topic	name of topics in Kafka where the event will be routed
created_at	record creation timestamp
event_key	Kafka event key
event_payload	Kafka event payload represented by Avro byte array
event_payload_type	type of event payload represented by byte array format: body_type:event_type
event_headers	Kafka event headers format: key1:value1,key2:value2...keyN:valueN

4.2.2 Role of microservice

Microservice should be responsible open database transaction before manipulation with business entity then insert, update, or delete entity and insert records representing outboxed events to outbox table and finally close transaction. Important to note that microservice will be responsible for constructing correct outbox table record representing event. This action is repeatable therefore it was decided to delegate this responsibility to *DefaultOutboxWriter* class which will implement custom *OutboxWriter* interface. Figure 4 demonstrates interface and its implementation using UML. In this way we can reuse implementation in code and exchange implementation if there are changes in storing record representing event to database.

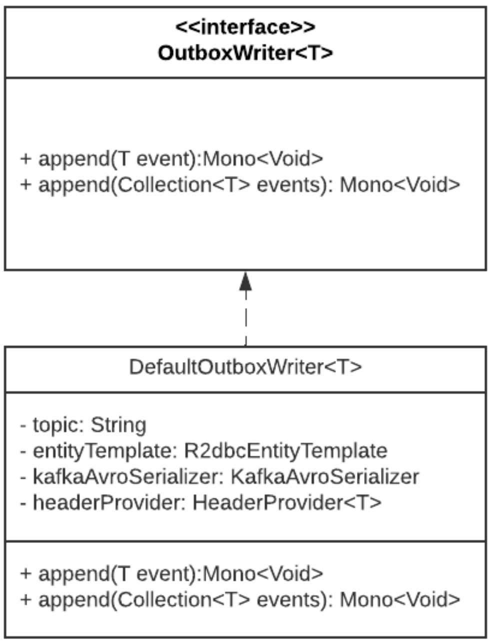
<<interface>>
OutboxWriter<T>
+ append(T event):Mono<Void>
+ append(Collection<T> events): Mono<Void>
DefaultOutboxWriter<T>
- topic: String
- entityTemplate: R2dbcEntityTemplate
- kafkaAvroSerializer: KafkaAvroSerializer
- headerProvider: HeaderProvider<T>
+ append(T event):Mono<Void>
+ append(Collection<T> events): Mono<Void>

Figure 4. OutboxWriter and DefaultOutboxWriter class diagram

In given application all business logic happens in service layer and communication with database using repository classes. Service layer class will be responsible to open database transaction before any changes in database then do changes and write event to outbox table using public methods of *OutboxWriter* interface.

4.2.3 Kafka Connect

Record representing event will be stored to outbox table and should be retrieved by Kafka Connect using source Debezium connector. Connector will be responsible for constructing Kafka event structure by using event key, payload, headers then event should be routed to corresponding topic. Kafka Connect has a set of built-in single message transformations which can be used for it [34]. The built-in transformations work quite well in simple scenarios, but it has not any transformation which can read additional event headers from column with given format, parse it and add them to Kafka event header. Kafka Connect provides the possibility to create custom transformation and include it to connector transformation pipeline [35]. The idea of such transformation is quite simple: connector generated data change event record, check that operation is create and read value of event_headers column from event, parse it using given format and construct new Kafka record with added header. The structure of connector generated initial data change event record can be found here [32].

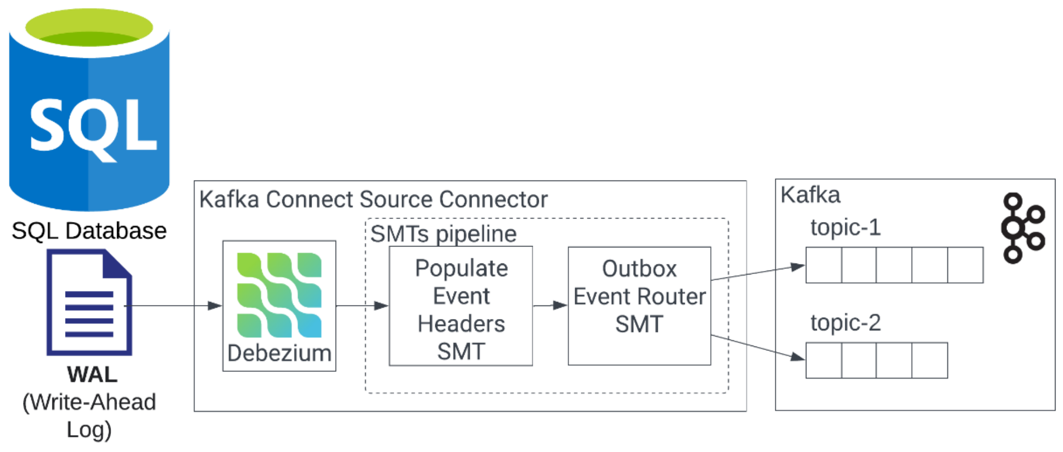

Figure 5. Visualisation of Kafka Connect source connector pipeline

When header is added then event can be routed to corresponding Kafka topic by using topic column from outbox table (Table 2). This task can be solved by using the available SMT Outbox Event Router [36]. The visualization of source connector SMTs pipeline can be found in Figure 5. It demonstrates that our pipeline has only two simple message transformations, but if needed it can have more.

It is important to note that Debezium streams change events for PostgreSQL source tables from publications that are created for the tables. Publications contain a filtered set of change events that are generated from one or more tables [32]. Here is only one database table (outbox) for which Kafka source connector should capture change events, so publication should be configured accordingly. It is important to note that the outbox table has only append mode, which means records representing event can be only inserted to the table, so only insert events should be captured by Kafka connect source connector. As a result, Postgres publication should be configured only for outbox table insert events. It can be done by executing following SQL command:

```
CREATE PUBLICATION dbz_publication FOR TABLE outbox
WITH (publish = 'insert');
```

Figure 6. Create a publication that only publishes INSERT operations in the outbox table

It is important to note that PostgreSQL WAL level should be set to logical to successfully read WAL by Debezium connector. It can be checked by executing following SQL command:

```
SHOW wal_level;
```

Figure 7. Check WAL level

If WAL set to some other value, it can be changed to logical by executing following command (Figure 8)

```
ALTER SYSTEM SET wal_level = logical;
```

Figure 8. Set logical level for WAL

After command execution configuration should be reloaded by database server restart or by execution configuration reload command [37].

Before connector setup it should be checked that the user has all the needed permissions: REPLICATION and LOGIN [38].

After that we can start creating Kafka connect source connector via Kafka Connect REST interface for outbox table (Figure 9).

```
POST /connectors/ HTTP/1.1
Host: localhost:8083
Content-Type: application/json
Content-Length: 2046

{
  "name": "outbox-debezium-source-connector",
  "config": {
    "name": "outbox-debezium-source-connector",
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "tasks.max": "1",
    "database.hostname": "${file:/var/secrets/kafka-connect/credentials.
properties:DATABASE_HOSTNAME}",
    "database.port": "${file:/var/secrets/kafka-connect/credentials.
properties:DATABASE_PORT}",
    "database.user": "${file:/var/secrets/kafka-connect/credentials.
properties:DATABASE_USER}",
    "database.password": "${file:/var/secrets/kafka-connect/credentials.
properties:DATABASE_PASSWORD}",
    "database.dbname": "${file:/var/secrets/kafka-connect/credentials.
properties:DATABASE_DBNAME}",
    "database.server.name": "${file:/var/secrets/kafka-connect/credentials.
properties:DATABASE_SERVER_NAME}",
    "schema.whitelist": "public",
    "table.whitelist": "public.outbox",
    "tombstones.on.delete": "false",
    "slot.name": "debezium_dev",
    "slot.drop.on.stop": "false",
    "publication.name": "dbz_publication",
    "plugin.name": "pgoutput",
    "transforms": "addHeaders,outbox",
    "transforms.addHeaders.type":
"com.b.k.e.c.t.DebeziumAddHeadersTransformation",
    "transforms.addHeaders.table.field.event.headers": "event_headers",
    "transforms.outbox.type": "io.debezium.transforms.outbox.EventRouter",
    "transforms.outbox.table.field.event.id": "id",
    "transforms.outbox.route.by.field": "topic",
    "transforms.outbox.table.field.event.key": "event_key",
    "transforms.outbox.table.field.event.payload": "event_payload",
    "transforms.outbox.route.topic.replacement": "${routedByValue}",
    "heartbeat.action.query": "update debezium_heartbeat set last_heartbeat_ts
= now();",
    "heartbeat.interval.ms": "60000",
    "value.converter": "io.debezium.converters.ByteBufferConverter"
    }
}
```

Figure 9. Connector creation command via Kafka Connect REST interface

The explanation of parameters is given in appendix 2 [39], [40], [41], [42], [43]. It is possible to follow how SMTs pipeline is configured. It is important to note that the order of transformations is significant, some transformations cannot be applied after others. In this case if change order of addHeaders and outbox, it will not work, because EventRouter class will encode event to Avro byte array, but DebeziumAddHeadersTransformation not able to encode from Avro byte array to retrieve needed values from change data capture event.

5 Solution operation

As result solution which was demonstrated in POC was evaluated and applied, but as we know no solution can be perfect on its first attempt. Some problems can occur during solution operation and the solution should be improved.
This chapter will describe problems which was faced during solution operation and how initial solution was improved to resolve this problem.

5.1 PostgreSQL inactive or lagging replication slots

It is widespread practice to use a single PostgreSQL database instance to reduce the cost of development and staging environment. WAL is shared between PostgreSQL server instance and there is no way to sprit it between database, schema, or table. There can be many updates in one database, but connector not captured changes for this database, as example here is connector for outbox table in development database and staging database, if here is a lot of inserts to development database then it can create replication slot lag. Also, there only one outbox table included to connector changes capturing, so if changes happen in other tables, then replication slot lag can occur. Inactive or lagging replication could cause problems in a database, like an ever-increasing disk usage not associated to any growth of the amount of data in the database [40]. Debezium connector have proposal for such situation. Here is way to include Debezium heartbeat [45].
For Debezium heartbeat implementation was designed additional database table *debezium_heartbeat* (Table 4).

Table 4. Debezium heartbeat table

Column	**Type**	**Modifiers**
last_heartbeat_ts	timestamp with time zone	default now() primary key

Usage of *debezium_heartbeat* table is simple every time when heartbeat happens insert new timestamp to *last_heartbeat_ts* column. In this way following query will be used as parameter for *heartbeat.action.query* in Kafka connector configuration:

```
UPDATE debezium_heartbeat
SET last_heartbeat_ts = NOW();
```

Figure 10. Debezium connector heartbeat query

Parameter *heartbeat.interval.ms* will be set to 60000 to execute this query every minute.

5.2 Huge amount of data in outbox table

Outbox table is representing append-only log in relational database. In this way every time when record representing event inserted to database stays here forever, which cases database size grow. In some use cases, it will be useful or necessary to store such records forever, but in scope of given application it just consumes disk space. One way to handle this situation is delete record representing event from database immediately after it has been processed by Kafka connect connector and sent it to corresponding Kafka topic. However, this approach may result in the loss of any useful information in case of a production incident. Another option is to establish a retention policy to determine the appropriate duration for retaining this data before it is deleted. Retention policy can be applied on database or application level. Retention policy on database level can be done by using *pg_cron* extension and specifying job using SQL command (Figure 11) for running at 00:00 on day-of-month 8 and on Monday and keep 1 week data [46].

```
SELECT cron.schedule('0 0 8 * MON', $$DELETE FROM outbox WHERE created_at
< now() - interval '1 week'$$);
```

Figure 11. Retention policy specification command in database table using pg_cron extension

Retention policy can be also implemented on application level by using Spring *@Scheduled* annotation using *0 0 8 ? * MON* cron expression and executing same SQL query by using *R2dbcEntityTemplate*.

Implementation retention policy in application level gives more flexibility to manipulate cron job period via environment variable and change query if needed without administrator just change code and re-deploy latest version. The reason for given solution was used application level cron job for outbox table clean up.

6 Results

This chapter will give overview about solution operation results.

6.1 Kafka source connector performance

Every new solution needs to be evaluated before going to production environment. Load tests were performed on the testing environment for the reliability of the solution verification.

Kafka connector metrics (Table 5) was collected for 15 minutes during load test from testing environment. During this period, approximately 4640 records were written to the outbox table, processed by the connector, and written to Kafka topics. This can be calculated by using *source-record-write-total* maximum value subtract minimum value of period.

Table 5. Kafka source connector metrics from testing environment (Apr 13, 1:20 pm – Apr 13, 1:35 pm)

Metrics	Avg.	Min.	Max.
Average time in milliseconds taken by the task to commit offsets [43] (offset-commit-avg-time-ms)	11.08	0	49
Average size of the batches processed by the connector [43] (batch-size-avg)	2.59	9e-3	3.6
Maximum time in milliseconds taken by the task to commit offsets [43] (offset-commit-max-time-ms)	11.08	0	49
Maximum size of the batches processed by the connector [43] (batch-size-max)	9.02	1	14
Average percentage of the task's offset commit attempts that succeeded [43] (offset-commit-success-percentage)	1	1	1
Average percentage of the task's offset commit attempts that failed or had an error [43] (offset-commit-failure-percentage)	0	0	0

Most recent number of records polled by the task but not yet completely written to Kafka [43] (source-record-active-count)	0.033	0	2
Maximum number of records polled by the task but not yet completely written to Kafka [43] (source-record-active-count-max)	9.02	1.00	14
Average number of records polled by the task but not yet completely written to Kafka [43] (source-record-active-count-avg)	1.40	9e-3	2
Number of records output from the transformations and written to Kafka for the task belonging to the named source connector in the worker [43] (since the task was last restarted) (source-record-write-total)	13.88k	11.54k	16.18k
After transformations are applied, this is the average per-second number of records output from the transformations and written to Kafka for the task belonging to the named source connector in the worker (excludes any records filtered out by the transformations) [43] (source-record-write-rate)	5.12	0.021	7.19
Before transformations are applied, this is the number of records produced or polled by the task belonging to the named source connector in the worker (since the task was last restarted) [43] (source-record-poll-total)	13.88k	11.54k	16.18k
Before transformations are applied, this is the average per-second number of records produced or polled by the task belonging to the named source connector in the worker [43] (source-record-poll-rate)	5.19	0.017	7.22

To check the count of records written to outbox table during this period here was executed query (Figure 12) and received result of 4640 records. Number of records written to outbox table is the same as number of records written to Kafka during this period collected by metrics. 100% of records representing events were successfully read by connector and published to corresponding Kafka topic which also confirmed by parameter from Kafka Connect metrics.

```
SELECT COUNT(*)
FROM outbox
WHERE created_at >= '2023-04-13 13:20'
  AND created_at <= '2023-04-13 13:35';
```

Figure 12. SQL query to check count of record written to outbox table during

To collect statistics on the write rate per minute and per second of the outbox table, the queries (Figure 13, Figure 14) were executed. There are results of the queries (Table 6)**Error! Reference source not found.**.

```
SELECT AVG(req_per_ts), MIN(req_per_ts), MAX(req_per_ts)
FROM (SELECT TRUNC(EXTRACT(EPOCH FROM created_at) / EXTRACT(EPOCH FROM
INTERVAL '1 min’)) AS ts, COUNT(*) AS req_per_ts
      FROM outbox
      WHERE created_at >= '2023-04-13 13:20'
        AND created_at <= '2023-04-13 13:35'
      GROUP BY ts) AS ts_and_req_per_ts
```

Figure 13. SQL query to collect statistic of outbox table write rate per minute for given period

```
SELECT AVG(req_per_ts), MIN(req_per_ts), MAX(req_per_ts)
FROM (SELECT TRUNC(EXTRACT(EPOCH FROM created_at) / EXTRACT(EPOCH FROM
INTERVAL '1 s')) AS ts, COUNT(*) AS req_per_ts
      FROM outbox
      WHERE created_at >= '2023-04-13 13:20'
        AND created_at <= '2023-04-13 13:35'
      GROUP BY ts) AS ts_and_req_per_ts
```

Figure 14. SQL query to collect statistic of outbox table write rate per second for given period

Table 6. Record the writes rate of testing environment for period (Apr 13, 1:20 pm – Apr 13, 1:35 pm)

	Avg.	Min	Max
Records/per minute	330	27	421
Records/per second	6	1	19

The results (Table 5, Table 6) passed acceptance criteria and solution was published to production.

Real metrics can only be collected and analyzed after solution operation in the production environment. Kafka connector metrics (Table 7) was collected for period 9 days from production environment. During this period, approximately 12.08 million records were written to the outbox table, processed by the connector, and written to Kafka topics. This can be calculated by using *source-record-write-total* maximum value subtract minimum value of period.

Table 7. Kafka source connector metrics from prod (Apr 4, 12:00 am – Apr 13, 4:29 pm)

Metrics	Avg.	Min.	Max.
Average time in milliseconds taken by the task to commit offsets [43] (offset-commit-avg-time-ms)	6.02	4.05	23.9
Average size of the batches processed by the connector [43] (batch-size-avg)	7.24	1.31	22.7
Maximum time in milliseconds taken by the task to commit offsets [43] (offset-commit-max-time-ms)	6.02	4.05	23.9
Maximum size of the batches processed by the connector [43] (batch-size-max)	19.21	6.27	61.4
Average percentage of the task’s offset commit attempts that succeeded [43] (offset-commit-success-percentage)	1.00	0.99	1
Average percentage of the task’s offset commit attempts that failed or had an error [43] (offset-commit-failure-percentage)	0	0	0
Most recent number of records polled by the task but not yet completely written to Kafka [43] (source-record-active-count)	0.19	0	1.35
Maximum number of records polled by the task but not yet completely written to Kafka [43] (source-record-active-count-max)	19.21	6.27	61.42

Average number of records polled by the task but not yet completely written to Kafka [43] (source-record-active-count-avg)	3.71	0.85	11.52
Number of records output from the transformations and written to Kafka for the task belonging to the named source connector in the worker (since the task was last restarted) [43] (source-record-write-total)	24.99M	18.57M	30.65M
After transformations are applied, this is the average per-second number of records output from the transformations and written to Kafka for the task belonging to the named source connector in the worker (excludes any records filtered out by the transformations) [43] (source-record-write-rate)	14.52	2.64	45.46
Before transformations are applied, this is the number of records produced or polled by the task belonging to the named source connector in the worker (since the task was last restarted) [43] (source-record-poll-total)	24.99M	18.57M	30.65M
Before transformations are applied, this is the average per-second number of records produced or polled by the task belonging to the named source connector in the worker [43] (source-record-poll-rate)	14.51	2.63	45.55

To check the count of records written to outbox table during this period here was executed query represented in **Error! Reference source not found.** and the result of the query is 12088969 records. Number of records written to outbox table is the same as number of records written to Kafka during this period collected by metrics. 100% of records representing events were successfully read by connector and published to corresponding Kafka topic which also confirmed by parameter from Kafka Connect metrics.

```
SELECT COUNT(*)
FROM outbox
WHERE created_at >= '2023-04-04 00:00'
  AND created_at <= '2023-04-13 16:29';
```

Figure 15. SQL query to check count of record written to outbox table during given period

To collect statistics on the write rate per minute and per second of the outbox table, the queries (Figure 16, Figure 17) were executed. There are results of the queries (Table 8).

```
SELECT AVG(req_per_ts), MIN(req_per_ts), MAX(req_per_ts)
FROM (SELECT TRUNC(EXTRACT(EPOCH FROM created_at) / EXTRACT(EPOCH from
INTERVAL '1 min')) AS ts, COUNT(*) AS req_per_ts
      FROM outbox
      WHERE created_at >= '2023-04-04 00:00'
        AND created_at <= '2023-04-13 16:29'
      GROUP BY ts) AS ts_and_req_per_ts;
```

Figure 16. SQL query to collect statistic of outbox table write rate per minute for given period

```
SELECT AVG(req_per_ts), min(req_per_ts), max(req_per_ts)
FROM (SELECT TRUNC(EXTRACT(EPOCH FROM created_at) / EXTRACT(EPOCH FROM
INTERVAL '1 s')) AS ts, COUNT(*) AS req_per_ts
      FROM outbox
      WHERE created_at >= '2023-04-04 00:00'
        AND created_at <= '2023-04-13 16:29'
      GROUP BY ts) AS ts_and_req_per_ts;
```

Figure 17. SQL query to collect statistic of outbox table write rate per second for given period

Table 8. Record the write rate of production environment for period (Apr 4, 12:00 am – Apr 13, 4:29 pm)

	Avg.	Min	Max
Records/per minute	867	1	4347
Records/per second	15	1	2579

Using metrics (Table 7) it can be concluded that connector is able capture and process quite a high number of records. The period when 2579 records were inserted to database was additionally analyzed and it took only one minute to process all records by connector.

7 Summary

The aim of this thesis was to implement atomic database updates and event publishing in scope of existing application and provide description of work which was done to fulfil requirements.

This thesis demonstrates how atomic database updates and event publishing can be implemented in scope of given technologies.

During the thesis was done analysis of technology stack of given application, which allowed to consider the peculiarities of application and to choose more suitable tools for problem solving. The thesis collects information about different patterns for problem solution, analysis benefits and drawbacks of each of pattern. Proof of the concept was developed by author taken into consideration collected information about application and possible theoretical implementations. Proof of the concept was applied to staging environments first, then evaluated and applied to production environment. After that author collected feedback about solution and adjusted it to fulfil new requirements.

The thesis collects metrics results during load testing in testing environment and real solution operation in production environment, which made it possible to verify solution performance. The metrics show that solution is stable under high load and performs very well in the production environment.

The approach evaluated and developed in scope of the thesis can be adjusted and used in other applications with same technology stack. As given application uses SQL database and the solution provided in the thesis based on SQL database features, so future development of the topic can be development solution for NoSQL databases.

References

[1] *Microservices and APIs: Why Should Businesses Care?*, Akana, Mar. 5, 2023. Accessed: Mar. 5, 2023. [Online]. Available: https://www.akana.com/resources/microservices-why-should-businesses-care.

[2] C. Richardson, Microservices Patterns: With Examples in Java, New York: Manning, 2018.

[3] *Microservice architecture style*, Microsoft, 2023. Accessed: Mar. 3, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/ microservices.

[4] Java: The Complete Reference, Eleventh Edition 11th Edition, New York: McGraw-Hill Education, 2019.

[5] *Upgrading major Java versions*, Oracle, Oct. 6, 2014. Accessed: Mar 7, 2023. [Online]. Available: https://blogs.oracle.com/java/post/upgrading-major-java-versions.

[6] *Java Top 10 Libraries*, JavaTpoint, 2021. Accessed: Mar. 7, 2023. [Online]. Available: https://www.javatpoint.com/java-top-10-libraries.

[7] *Java Platform, Standard Edition & Java Development Kit*, Oracle, 2023. Accessed: Mar. 7, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/.

[8] *Overview of Spring Framework*, VMware, 2023. Accessed: Mar. 7, 2023. [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/ html/overview.html.

[9] *Spring Framework Overview*, VMware, 2023. Accessed: Mar. 7, 2023. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html.

[10] *Spring Cloud: Overview*, VMware, 2023. Accessed: Mar. 7, 2023. [Online]. Available: https://spring.io/projects/spring-cloud.

[11] *Reactive*, VMware, 2023. Accessed: Mar. 5, 2023. [Online]. Available: https://spring.io/reactive.

[12] *PostgreSQL: About*, The PostgreSQL Global Development Group, 2023. Accessed: Mar. 5, 2023. [Online]. Available: https://www.postgresql.org/about/.

[13] *PostgreSql: Feature Matrix*, The PostgreSQL Global Development Group, 2023. Accessed: Mar. 5, 2023. [Online]. Available: https://www.postgresql.org/about/featurematrix/.

[14] *Apache Kafka: Introduction*, Apache Software Foundation, 2023. Accessed: Mar. 9, 2023. [Online]. Available: https://kafka.apache.org/intro.

[15] *Apache Kafka: Use Cases*, Apache Software Foundation, 2023. Accessed: Mar. 9, 2023 [Online]. Available: https://kafka.apache.org/uses..

[16] *Apache Avro 1.11.1 Documentation*, The Apache Software Foundation, 2022. Accessed: Mar. 9, 2023 [Online]. Available: https://avro.apache.org/docs/1.11.1/..

[17] *Apache Avro: Data Serialization and Deserialization*, Apache Software Foundation, 2022. Accessed: Mar. 9, 2023. [Online]. Available: https://avro.apache.org/docs/ 1.11.1/specification/#data-serialization-and-deserialization.

[18] *Guide to Apache Avro*, Baeldung, Feb. 19, 2023. Accessed: Mar. 9, 2023. [Online]. Available: https://www.baeldung.com/java-apache-avro.

[19] *Schema Evolution and Compatibility*, Confluent, 2023. Accessed: Mar. 9, 2023. [Online]. Available: https://docs.confluent.io/platform/current/schema-registry/avro.html.

[20] *Why Avro for Kafka Data?*, Confluent, Feb. 25, 2015. Accessed: Mar. 9, 2023. [Online]. Available: https://www.confluent.io/blog/avro-kafka-data/.

[21] *Publish events using database triggers*, microservices.io, 2023. Accessed: Mar. 11 Mar, 2023. [Online]. Available: https://microservices.io/patterns/data/database-triggers.html.

[22] T. Eric, *How to Build a Reliable Live Data Replication Architecture (Part 1): Track Events With SQL Triggers*, Jul. 13, 2021. Accessed: Mar. 11, 2023. [Online]. Available: https://medium.com/bina-nusantara-it-division/how-to-build-a-reliable-live-data-replication-architecture-part-1-track-events-with-sql-triggers-4d6cb3c12eac.

[23] *Debezium: Connectors*, Debezium Community, 2023. Accessed: Mar. 18, 2023. [Online]. Available: https://debezium.io/documentation/reference/stable/ connectors/index.html.

[24] *Cockroach: Change Data Capture Overview*, Cockroach Labs, 2023. Accessed: Mar. 18, 2023. [Online]. Available: https://www.cockroachlabs.com/docs/stable/ change-data-capture-overview.html#what-is-change-data-capture.

[25] *Change data capture for DynamoDB Streams*, Amazon, 2023. Accessed: Mar. 18, 2023. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/

developerguide/Streams.html.

[26] *Enable and disable change data capture*, Microsoft, 2023. Accessed: Mar. 18, 2023. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/track-changes/enable-and-disable-change-data-capture-sql-server?view=sql-server-ver15.

[27] *LinkedIn Databus*, LinkedIn, 2023. Accessed: Mar. 18, 2023. [Online]. Available: https://github.com/linkedin/databus.

[28] *PostgreSQL: Write-Ahead Logging*, The PostgreSQL Global Development Group, 2023. Accessed: Mar. 19, 2023. [Online]. Available: https://www.postgresql.org/docs/current/wal-intro.html.

[29] *PostgreSQL Extensions to the JDBC API*, The PostgreSQL Global Development Group, 2023. Accessed: Mar. 19, 2023. [Online]. Available: https://jdbc.postgresql.org/documentation/server-prepare/.

[30] *Introduction to Debezium*, Baeldung, May 6, 2023. Accessed: Mar. 19, 2023. [Online]. Available: https://www.baeldung.com/debezium-intro.

[31] *Kafka Connect*, Confluent, 2023. Accessed: Mar. 19, 2023. [Online]. Available: https://docs.confluent.io/platform/current/connect/index.html#kafka-connect.

[32] *Debezium connector for PostgreSQL*, Debezium Community, 2023. Accessed: Mar. 19, 2023. [Online]. Available: https://debezium.io/documentation/reference/stable/connectors/postgresql.html.

[33] G. Morling, *Reliable Microservices Data Exchange With the Outbox Pattern*, Feb. 19, 2019. Accessed: Mar. 19, 2023. [Online]. Available: https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/.

[34] *Single Message Transforms for Confluent Platform*, Confluent, 2023. Accessed: Mar. 20, 2023. [Online]. Available: https://docs.confluent.io/platform/current/connect/transforms/overview.html#single-message-transforms-for-cp.

[35] *Custom transformations*, Confluent, 2023. Accessed: Apr. 4, 2023. [Online]. Available: https://docs.confluent.io/platform/current/connect/transforms/custom.html#custom-transformations.

[36] *Outbox Event Router*, Debezium Community, 2023. Accessed: Apr. 4, 2023. [Online]. Available: https://debezium.io/documentation/reference/stable/transformations/outbox-event-router.html.

[37] *PostgreSQL Documentation: ALTER SYSTEM SQL Commands*, The PostgreSQL Global Development Group, 2023. Accessed: Apr. 8, 2023. [Online]. Available: https://www.postgresql.org/docs/current/sql-altersystem.html.

[38] *Debezium connector for PostgreSQL: Setting up permissions*, Debezium Community, 2023. Accessed: Apr. 8, 2023. [Online]. Available: https://debezium.io/documentation/reference/stable/connectors/postgresql.html#postgresql-permissions.

[39] *JSONPath Syntax*, SmartBear Software, 2023. Accessed: May 7, 2023. [Online]. Available: https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html.

[40] *Debezium connector for PostgreSQL: Connector configuration example*, Debezium Community, 2023. Accessed: May 7, 2023. [Online]. Available: https://debezium.io/documentation/reference/stable/connectors/postgresql.html#postgresql-example-configuration.

[41] *How to Use Kafka Connect - Get Started*, Confluent, 2023. Accessed: May 7, 2023. [Online]. Available: https://docs.confluent.io/platform/current/connect/userguide.html.

[42] *Outbox Event Router*, Debezium Community, 2023. Accessed: May 7, 2023. [Online]. Available: https://debezium.io/documentation/reference/stable/transformations/outbox-event-router.html.

[43] *Kafka Connect Configurations for Confluent Platform*, Confluent, 2023. Accessed: May 7, 2023. [Online]. Available: https://docs.confluent.io/platform/current/installation/configuration/connect/index.html.

[44] *Set up logical replication to Aiven for PostgreSQL: Manage inactive or lagging replication slots*, Aiven Team, 2023. Accessed: Apr. 9, 2023. [Online]. Available: https://docs.aiven.io/docs/products/postgresql/howto/setup-logical-replication#manage-inactive-or-lagging-replication-slots.

[45] *Debezium connector for PostgreSQL: WAL disk space consumption*, Debezium Community, 2023. Accessed: Apr. 9, 2023. [Online]. Available: https://debezium.io/documentation/reference/stable/connectors/postgresql.html#postgresql-wal-disk-space.

[46] *PostgresSQL extention pg_cron official github page*, Citus Data, 2023. Accessed: Apr. 10, 2023. [Online]. Available: https://github.com/citusdata/pg_cron.

[47] *Monitor Kafka Connect and Connectors: Use JMX to Monitor Connect*, Confluent, 2023. [Online]. Available: https://docs.confluent.io/platform/current/connect/monitoring.html#use-jmx-to-monitor-kconnect. Accessed: May 6, 2023.

[48] *Kafka Connect Security Basics*, Confluent, 2023. [Online]. Available: https://docs.confluent.io/platform/current/connect/security.html#fileconfigprovider. Accessed: Apr. 8, 2023.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Explanation of connector configuration parameters

JSONPath of payload	**Explanation**
$['name']	The name of the connector
$['config’]	The connector’s configuration
$['config’]['name']	The name of the connector
$['config']['connector.class']	The name of this PostgreSQL connector class
$['config']['tasks.max']	The maximum number of tasks that should be created for this connector
$['config']['database.hostname']	The address of the PostgreSQL server. FileConfigProvider is used [39] for given configuration.
$['config']['database.port']	The port number of the PostgreSQL server. FileConfigProvider is used [39] for given configuration.
$['config']['database.user']	The name of the PostgreSQL user that has the required privileges. FileConfigProvider is used [39] for given configuration.
$['config']['database.password']	The password for the PostgreSQL user that has the required privileges. FileConfigProvider is used [39] for given configuration.
$['config']['database.dbname']	The name of the PostgreSQL database to connect to. FileConfigProvider is used [39] for given configuration.

$['config']['database.server.name']	The database server logical name. FileConfigProvider is used [39] for given configuration.
$['config']['schema.whitelist']	List of schemas to include in copying
$['config']['table.whitelist']	List of tables to include in copying
$['config']['tombstones.on.delete']	Controls whether a delete event is followed by a tombstone event
$['config']['slot.drop.on.stop']	Whether or not to delete the logical replication slot when the connector stops in a graceful, expected way
$['config']['publication.name']	The name of the PostgreSQL publication created for streaming changes when using pgoutput
$['config']['plugin.name']	The name of the PostgreSQL logical decoding plug-in installed on the PostgreSQL server
$['config']['transforms']	Aliases for the transformations to be applied to records
$['config']['transforms.addHeaders.type']	The addHeaders transformation class
$['config']['transforms.addHeaders.table.field.event.headers']	The addHeaders transformation configuration of column containing headers in key1:value1,key2:value2...keyN: valueN format
$['config']['transforms.outbox.type']	The outbox transformation class
$['config']['transforms.outbox.table.field.event.id']	Specifies the outbox table column that contains the unique event ID. This ID will be stored in the

	emitted event’s headers under the id key
$['config']['transforms.outbox.route.by.field']	Contains a value that the SMT appends to the name of the topic to which the connector emits an outbox message
$['config']['transforms.outbox.table.field.event.key']	Specifies the outbox table column that contains the event key.
$['config']['transforms.outbox.table.field.event.payload']	Specifies the outbox table column that contains the event payload
$['config']['transforms.outbox.route.topic.replacement']	Specifies the name of the topic to which the connector emits outbox messages
$['config']['heartbeat.action.query']	Specifies a query that the connector executes on the source database when the connector sends a heartbeat message
$['config']['heartbeat.interval.ms']	Controls how frequently the connector sends heartbeat messages to a Kafka topic.
$['config']['value.converter']	Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka