UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in Computer Science

Final Dissertation

A Change Data Capture System for SpazioDati

Design and Implementation

Supervisor	Student
Prof. Gabriel Mark Kuper	Daniele Parmeggiani

Academic year 2020/2021

Dedication

Abstract

This Theses presents work carried out during my curricular internship, which pertains the design and implementation of a Change Data Capture system. An interest of the marketing team gave rise to the demand for the presented system, as explained in Chapter 1, which in essence solves the problem of capturing data change events from a data source, sometimes aggregating such data, and store them in a separate place for the marketing team to make further use thereof.

An introduction to the scope and rationale for the system is presented in the first chapter. The several technologies that constitute the subsystems presented are explained in the second chapter, along with their structural composition into the overall system. Next, several properties of the data at hand are introduced in Chapter 3 and one of them is explored in greater detail in Chapter 4. Additionally, some paths for possible future enhancements are outlined in Chapter 5. In Appendix A the listings of relevant source code are provided.

As the contents of this document pertain the work I've carried out within SpazioDati, all information relevant to their intellectual properties has been anonymized so as to not disclose it. The methods for such anonymization have been portrayed in §1.1. Nowhere in this document their customers' data is shown or cited, partly, or in whole.

Contents

Chapter 1

Introduction

The work presented in this Theses pertains the content of the curricular internship carried out from November 2020 through December of the same year.

The curricular internship, part of the required credits for the Bachelor's Degree in Computer Science at the University of Trento, was performed remotely[1] at SpazioDati, a technology company based in Trento, for their product Atoka.

> SpazioDati works on applying Semantic Text Analysis and Machine Learning models on massive amounts of corporate data to provide services – both B2B and B2C – of Sales Intelligence, Lead Generation, Data Cleansing, and more. Our main product, Atoka, is used by thousands of small and big companies in Italy and abroad. The underlying graph also powers Dandelion API, a state-of-the-art text analytics semantic engine.[2]
>
> Atoka is the latest SpazioDati product which, thanks to Big Data and semantics, collects detailed information on 6 million companies in Italy and 4 million companies in the United Kingdom. Atoka allows you to create and extract extremely precise and targeted "customer lists", minimizing the information asymmetry that marketing managers and sales managers often face when undertaking a business development campaign. Atoka provides companies with all the information on potential targets, as well as for monitoring the evolution of the market and competitors. In other words, it facilitates B2B lead generation and allows access to a daily updated database of 6 million Italian companies and 4 million British companies, thanks to the partnership with Cerved Group. No other provider in Italy is able to support such numbers [...]: with 500 million web pages and 70 thousand news items analyzed, Atoka allows Sales and Marketing Intelligence actions never seen before in the Italian market.[3]

[1]Due to regulations related to the Coronavirus epidemic, on-premises internships have been discouraged, and should have only been performed in case of necessity. Such was not the case for the work presented here. See [10] for further details.

[2]Description via `https://datarade.ai/data-providers/spaziodati/profile`, last visited January 29, 2021.

[3]Ibid.

1.1 Non Disclosure

In this document several information internal to SpazioDati is referenced. In order to avoid disclosing information relevant to their intellectual property, the relational tables later referenced in this Theses have been anonymized.

In particular the names of the eleven tables referenced have been replaced with the Greek letters α, β, γ, δ, ϵ, ζ, η, θ, ι, κ, and λ. Additionally, referenced column names do not represent the real names used in SpazioDati; nevertheless they provide a sensible metaphor to their actual counterpart.

TODO The contents of this Theses have been reviewed by SpazioDati and they have acknowledged this document does not contain information sensitive to either their customers or their intellectual properties.

1.2 Problem Statement

Before stating the matter at hand, let us first consider the prior rationale for this endeavor.

Since the Atoka service has a subscription-based business model,[4] one important problem considered by the marketing team is the necessity to keep already-subscribing users engaged with the product, so as to induce a continuation of the subscription.

The marketing team has considered that a user, presented with insightful data about his usage of the product, would observe his possible under-usage, or learn techniques to a more efficient usage of Atoka. This would be achieved by investigating the aforementioned usage of the product, employing activity data recorded on their internal relational database services.

Since the queries intended for this application are computationally expensive, instead of running those queries directly on the main database, it was decided the database would emit data change events[5] that would be sent to, and recorded in, a separate database, used specifically for this application.

By design, such a mechanism effectively produces data duplication, but impacting the performance of Atoka's database service (thus degrading the overall user experience) was valued a problem much greater than the expense of data replication.

The queries carried out by the marketing team are, though, of little concern for this Theses. Our interest lies in making the data involved available to the team, which will later employ it as they deem fit.

1.3 Change Data Capture Systems

In the previous Section, the necessity to record data as it was modified emerged. Such need may be fulfilled by a Change Data Capture (CDC) system;[6] in this Section we will go over what it amounts to.

[4]I.e. users pay a recurring fee for access to the service and may decide to stop paying (discontinuing the subscription), according to the terms of service.

[5]Here, *data change events* refers to a series of events that can be used to reproduce the changes that have occurred in the database. The actual meaning of this expression will become clearer in §1.3.

[6]For further reference see [6].

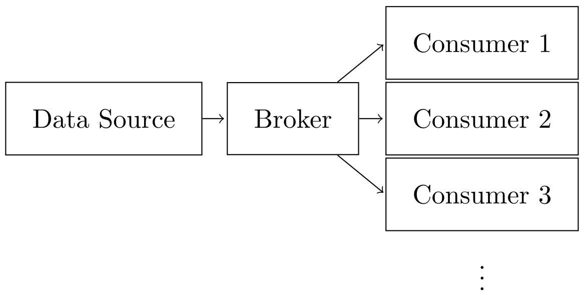

Figure 1.1: A Generic CDC System.

With *change data* we refer to data regarding a modification of some data source's contents; thus, logically, CDC systems are composed of at least a data source.[7] Along with it, a CDC system requires a mechanism to deliver this data to some sub-systems that consume it. As such, we can outline at least three different components of a CDC system: a data source, a change data broker, and one or several consumer sub-systems.

A broker is intended as a delivery mechanism for messages, by which we mean content, interchanged between systems. For a CDC system, message content delivered by a broker is change data events, which are sent from the data source to the downstream consumers. What we expect from this sub-system is that messages are delivered to consumers in the order they were received, are consumed exactly once, and have negligible delay.

In Figure 1.1 a generic outline of a CDC system is presented.

Furthermore, in the case at hand, a PostgreSQL server is the data source, Apache Kafka (along with Kafka Connect, and a Debezium Connector) is the broker, and the several consumers are Kafka Connect Sinks, additionally enriched with Kafka Streams.[8] In Figure 1.2 the overall system, i.e. the content of this Theses, is presented; relations to parts of a generic CDC system are outlined with dashes.

As previously mentioned, the presented CDC system's output is stored in a PostgreSQL database, which has been specifically created for the purpose discussed in §1.2. It must not be confused with the data source.

1.4 On the Duality between Tables and Streams

We have since said that there would be a relational database as data source, another one as the output of the CDC system, and in between we have talked about some *events*. Clearly, as these events are materialized as messages, they do not pertain to a relational data model; instead they refer to a streaming data model, in which several messages are interchanged. Spontaneously, a question arises: how do these two models coexist and sustain a back and forth translation?

The answer is to be found in the two fundamental concepts of the models.

Let us consider a table (or relation) T. If this table is updated (i.e. a row is inserted, deleted or changed) we say there is a change data event δ. We

[7]Which may also be a component of other systems.
[8]Cf. Chapter 2.

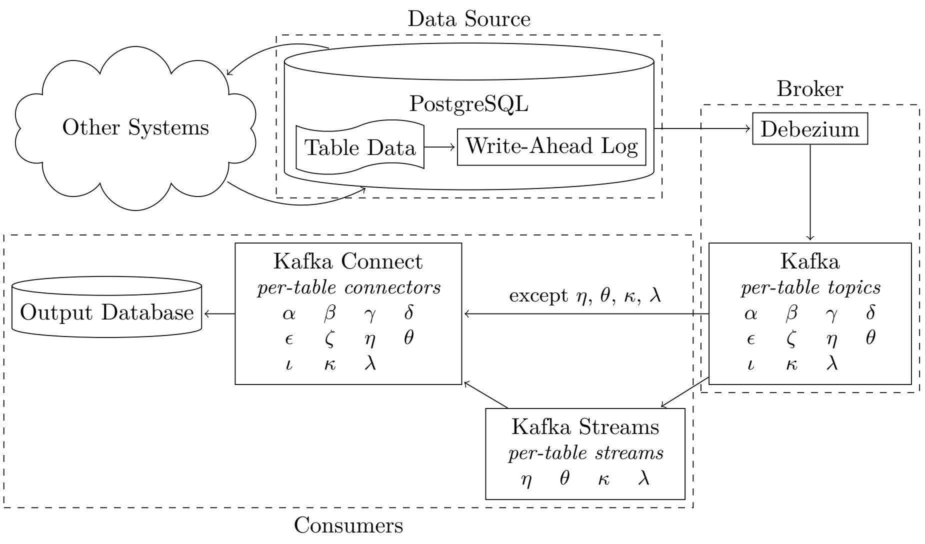

Figure 1.2: An Outline of the CDC System presented in this Theses.

can define an ordered unbounded sequence Δ of all changes to T, which is the stream of changes to T.

Let us define a sum of changes as the sequential application of data change events to some relation, such that $\delta + \epsilon$ is some data change, defined as the sequential application of δ and then ϵ, for any δ and ϵ being two sequential data change events in Δ. Let us consider α as the sum of all changes in Δ:

$$\alpha = \sum_{\delta \in \Delta} \delta \; .$$

It is trivial to assert that $\alpha = T$, which is to say that a table may be re-composed from the sequential application of its data change events.

Considering all of the above, we can conclude that the streaming of data change events of any table T is an alternative representation of T.

1.5 Notation

In the following chapters we'll make use of standard notation for Relational Algebra, best described in [8, Chapter 4].

Additionally, further notation will be introduced in order to allow for clarity between table representations, and formal discussion of aggregation queries.

1.5.1 Source and Destination Tables

Every table has one representation in the source database and another representation in the destination database. Thus simply referring to some table ω may be ambiguous, given the identical names of the two representations.

To avoid such ambiguity, let us introduce the following notation:

(a)

m	n
a	1
b	5
a	3

(b)

m	n
a	1
a	3

m	n
b	5

(c)

m	n
a	4

m	n
b	5

(d)

m	n
a	4
b	5

Table 1.1: Relations of Examples 1, 2, and 3.

- source $[\omega]$ and dest $[\omega]$ refer to the representation of some table ω at the source and destination databases respectively;
- source $[*]$ and dest $[*]$ refer to the representations of all tables collectively (i.e. the whole databases), at the source and destination databases respectively.

It should be noted that the source tables are spread between different logical, as well as physical locations; i.e. different relational databases, not hosted on the same server. This, though, should be of little interest: as better explained in the following chapters, the different source tables are represented into as many topics on the same Kafka cluster (cf. §2.1), abstracting away from the source. Therefore, source $[\omega]$ refers to the source representation of table ω, regardless of where, even logically, it is stored.

1.5.2 Aggregation Related Functions

Common constructs of Relational Algebra do not provide a way in which one can express aggregation queries.[9]

Effectively, this renders the notation defined in this section into something that is not Relational Algebra, since it does not satisfy the closure property, as shown later. Nonetheless, I argue that there can be a strict separation between relational algebra and the notation that will shortly be introduced. Also, a well-defined way for going back and forth between these different notations will be laid down.

Definition 1 (Grouping generation)**.** Let G be the grouping generation operator, ω any relation, and c, the grouping key, a column of ω. Let V be the set of all the values of the grouping key in ω. The grouping $G_c(\omega)$ is defined as the set of relations (groups) obtained performing all the possible selections from V:

$$G_c(\omega) = \{g : g = \sigma_{c=v}(\omega), \forall v \in V\} \ .$$

Example 1. Let ω be the relation shown in Table 1.1.a. $G_m(\omega)$ is shown in Table 1.1.b.

Observation 1. If $\mathcal{R}$ is the set of all relations, it is evident from the above definition that $G_c(\omega) \notin \mathcal{R}$ for any c and ω. Thus, the output of the grouping generation operator is *not* a relation.

[9]See remark in [8, p. 154].

Since every construct of Relational Algebra satisfies the closure property with respect to $\mathcal{R}$, the G operator from Definition (1) cannot be considered part of Relational Algebra.

Definition 2 (Grouping with multiple keys)**.** Let C be a set of grouping keys, and ω any relation. $G_C(\omega)$ is defined as

$$G_C(\omega) = \bigcup_{c \in C} G_c(\omega) \; .$$

Observation 2. A grouping with multiple keys, is still a grouping (i.e. a set of relations).

Definition 3 (Grouping selection)**.** Let H be a grouping, C the set of grouping keys of H, *f*, and h two aggregation functions[10], and c_i, and c_j two columns of any group in H. The grouping selection operation σ^* is defined as the grouping

$$\sigma^*_{f(c_i),h(c_j),\ldots}(H) = \Big\{ \big\{ \langle c : c \in C, f(g.c_i), h(g.c_j), \ldots \rangle \big\} : g \in H \Big\} \; .$$

Example 2. Continuing from Example 1, let us consider $\sigma^*_{\sum n}(G_m(\omega))$, as shown in Table 1.1.c.

Observation 3. The output of the grouping selection function σ^* is a grouping. Each one of the groups is a single instance relation.

Definition 4 (Grouping merge)**.** Let H be a grouping. The merging operation is defined as

$$M(H) = \bigcup_{g \in H} g \; .$$

Example 3. Continuing from Example 2, let us now consider $M\Big(\sigma^*_{\sum n}(G_m(\omega))\Big)$, as shown in Table 1.1.d.

Proposition 1. *A merged grouping is a relation.*

Proof. Notwithstanding Observation (1); from Definition (1): $\forall g \in G_c(\omega), g \in \mathcal{R}$, since it is defined as a selection.

A union of relations is a common operation of Relational Algebra, thus the result of $M(H) \in \mathcal{R}$, for any grouping H. $\square$

[10]Here "aggregation functions" is used to indicate some functions that produce a result from a succession. E.g. $\sum$, min, and so on.

Chapter 2

Technologies Employed

In this Chapter, we will go over the different technologies that compose the system built for SpazioDati, as it was outlined in Figure 1.2.

2.1 Apache Kafka

Before stating what Apache Kafka is, the reader will find it clearer if we introduce the log data structure first. That is, an unbounded, append-only sequence of records,[1] ordered by time. For the purposes of a log sequence, let us disregard a precise notion of time; instead, let us loosely assert that subsequent messages were recorded (and received) subsequently.

Kafka, at its core, is a distributed system which handles logs, whilst providing desirable features for such systems: fault-tolerance, high throughput, low latency, and responsiveness to unwanted network partitioning.

Kafka itself is a cluster of independent broker systems, which operate together making use of the Paxos[2] protocol to provide Kafka's functionalities.

Architecturally, as outlined in Figure 2.1, the overall system entails the presence of some Producers (which send records to the cluster), and Consumers (which read records from the cluster). These two sets of systems, which do not strictly pertain to Kafka but base their operations on its features, are completely decoupled from one another, even if, generally it is by the combination of them that a particular purpose is achieved. This is a desirable characteristic of the overall system, which enables interoperability between even vastly heterogeneous producer and consumer systems.

In Kafka's terminology, which is common in the field, records belong to topics; i.e. named logs. A Producer will instruct Kafka to store a record and that it pertains to a particular topic. Conversely, a Consumer will ask Kafka to read a record from a particular topic, at some specific offset.

A topic is further divided into so called partitions.[3] Generally, this division is most useful when considering the capability of some greater system, which is

[1] Or messages, or events, etc.

[2] Cf. [4].

[3] Therefore, to be precise, each single topic partition is a log and a topic is a set of logs. Nonetheless, in our use-case, as discussed shortly, we are not interested into splitting topics into multiple partitions, thus leaving little space to imprecision.

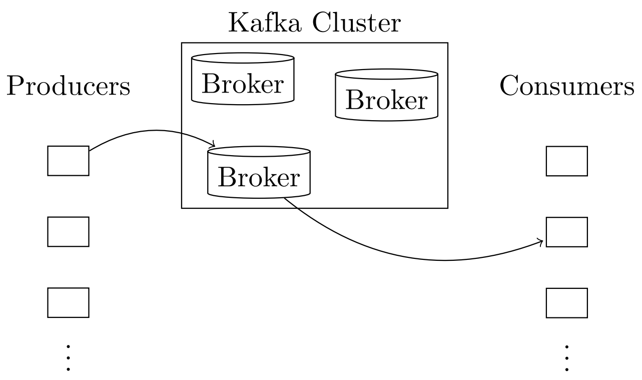

Figure 2.1: General overview of Kafka Architecture.

in part composed by Kafka, to meet increasing computational demands. Such is achieved by dividing homogeneous records into different partitions that are read by separate Consumers, thus enabling greater simplicity in the addition of Consumers, therefore expanding the computing capability assigned to the consumption of records.

Since each partition is a log in itself, records are inherently ordered, but ordering *between* partitions is not guaranteed; hence, partitioning topics, in our use-case, is not a viable option for increasing computational power, since ordering of data change events is key when reconstructing the source tables (cf. §1.4). Meeting greater computational demands will therefore have to be achieved through different means, such as increasing the computing power assigned to individual Consumers.

In our use case each table that is being reproduced from the data source, will be assigned to a different topic. Each table-topic will be read from separate Consumers, as better explained in the following §2.1.1.

One interesting aspect of Kafka's design is the pervasive and efficient usage of persistent memory, described in detail in [1, §4.2, and §4.3]. Seeing how disk read operations can have a throughput similar to that of a computer network,[4] the following statement from Kafka's Documentation makes a lot of sense:

> [...] Rather than maintain as much as possible in-memory and flush it all out to the filesystem in a panic when we run out of space, we invert that. All data is immediately written to a persistent log on the filesystem without necessarily flushing to disk. In effect this just means that it is transferred into the kernel's pagecache.

Thus, instead of spawning a memory management subsystem for its own purposes, Kafka relinquishes control and relies on the OS to handle its persistent memory. Kafka also uses the [5, `sendfile`(2)] system call for greater efficiency in I/O operations when receiving from or sending records over the network. A `sendfile` call directs data from an input file descriptor to an output file descriptor, without leaving the kernel-space, thus saving the programmer from having an additional buffer in user-space.

[4]Cf. [3].

2.1.1 Kafka Connect

From Kafka's documentation:

> Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other systems. It makes it simple to quickly define connectors that move large collections of data into and out of Kafka.[5]

Kafka Connect provides two abstractions over very common use-cases, namely Source and Sink Connectors. Effectively, with regards to the terminology previously introduced, a Source Connector is a Producer and a Sink Connector is a Consumer. The switch in terminology is due to the fact that any Connector is entirely managed by the Kafka Connect runtime, implying that some of their logic is managed by Connect. Thus, developers are inevitably more restricted in their design choices over Connectors; nonetheless, this restriction provides greater simplicity in the definition of Producers and Consumers that follow the common pattern stated in the above quotation, which is the rationale for Kafka Connect itself.

In order to allow external control, Kafka Connect provides a REST interface utilized for starting, stopping, pausing, and resuming Connectors, along with more functionalities designed to manage Kafka Connect itself. The rest of the operations are all managed internally, without the need for manual control.

In our case, we will have Debezium Source Connectors (cf. §2.3) for each source database, and as many Sink Connectors as there are tables in the output database. The code Sink Connectors use was purpose-built and can be found in Listing A.6, which will be explained in greater detail in the following chapters. For the purposes of this Section, it will suffice to say that the listed code generates a Sink Connector Task, which is responsible for handling the transfer of data out of Kafka and into the output database.

2.1.2 Kafka Streams

Kafka Streams is a library that conveys various utilities for developing systems that operate with Kafka, most importantly it addresses systems that act both as consumers and as producers at the same time. In particular, this underlies the concept of a stream, intended as the computation carried out by a chain of Producers that write to certain topics, Consumers that read from said topics as well as produce to other topics, and so forth.

Recalling our previous assertions on the duality of streams and tables (cf. §1.4), let us utilize such notion to state the usage of Kafka Streams in our application. Since a stream can be interpreted as a table, modifications to a stream constitute changes to a table, thus the operations of relational algebra can be adapted and applied to streams as well. I will not endeavor into a formal demonstration of such notion, nonetheless I will try to convince the reader that such is true in the code provided.

In our case, Kafka Streams are employed for the aggregations regarding certain tables, that will be described in §3.1. The code for components of the proposed system that make use of Kafka Streams are presented in Listings A.2, A.3, A.4, and A.5.

[5] 1, §8.1.

For instance, let us consider table η. In Aggregation 3.1, defined in the next Chapter, we have a grouping[6] with keys user, time, and class. In Listing A.2, lines 29 – 39, we utilize the Kafka Streams APIs to achieve the operation $G_{\text{user, time, class}}(\text{source}\,[\eta])$. The subsequent grouping selection[7] is performed on lines 48 and 49.

By looking at this example, difference with the notation proposed in Chapter 1 is evident. While the operator σ^* points to the result of the aggregation, for instance in this case $\sum$, on line 49 we compute the value one data change event at a time. This difference is in line with the duality expressed in §1.4.

2.2 PostgreSQL

Explanation of what a relational database is, is intentionally left to the several books[8] that provide a more thorough explanation than any that could possibly be expected in a section of a Theses. Additionally, the great majority of the intended audience for this document already has knowledge on this topic, therefore there is no strong rationale for which to indulge in such a broad explanation. Nonetheless, in this Section, particular aspects of the PostgreSQL relational database will be presented, when they pertain contents of this Theses.

2.2.1 Write Ahead Log

The Write Ahead Log (WAL) is a core component of PostgreSQL servers, and similar implementations are present in most relational databases. Its function is to log transactional data, i.e. data referred to some transaction, as defined in [8, §16.2].

Such a log provides data integrity. Power loss, OS failure, or hardware failure[9] can impede successful flushing of all data pages to disk, but writing the transactional changes themselves on a log, an operation far less costly due to the inherent sequential writing, is sufficient to later recover the data to a consistent state. As the name suggests, data is logged in the WAL before it is written to persistent storage, which is the reason why integrity is preserved.

Some further remarks in PostgreSQL's Documentation:

> Using WAL results in a significantly reduced number of disk writes, because only the log file needs to be flushed to disk to guarantee that a transaction is committed, rather than every data file changed by the transaction. The log file is written sequentially, and so the cost of syncing the log is much less than the cost of flushing the data pages. This is especially true for servers handling many small transactions touching different parts of the data store. Furthermore, when the server is processing many small concurrent transactions, one fsync of the log file may suffice to commit many transactions.[10]

For a more detailed explanation, refer to [7, Chapter 29].

[6]Recall Definition 1.

[7]Recall Definition 3.

[8]E.g. [8].

[9]Except when related to the persistent storage itself. Evidently, should such be the nature of hardware failure, no action can provide data integrity.

[10]From [7, §29.2]

2.2.2 Logical Replication

In addition to what's already been stated, the aforementioned WAL is used by PostgreSQL in order to provide the feature of logical replication. It is a process by which external systems can consume data from the WAL, thus essentially replicating the database contents.

Data is extracted and logically decoded;[11] next it is exported through a replication slot, which essentially transmits the data to some external system that can then use it according to its needs.

2.3 Debezium

Debezium is a Source Connector (cf. 2.1.1), developed by Red Hat.[12] It possesses the capability of extracting data change events from various relational database management systems, including PostgreSQL. Then, regardless of the kind of data source, it writes data to Kafka topics.

In particular, when extracting data from a PostgreSQL server, Debezium utilizes the means described in the previous Sections 2.2.1, and 2.2.2.

In our case, Debezium is configured to listen (by means of replication slots) to changes on tables from the source databases; specifically there is one Debezium Connector for each source database.[13] Then, data regarding each individual table is sent to a specific topic, as already stated in §2.1.

As will be stated in later chapters,[14] the independence of parallel processing between separate table-topics is an important aspect of the presented system's implementation.

2.4 Scala

Scala is the programming language of choice for a substantial part of work done for the proposed system's implementation. The selection of source code displayed in Appendix A is entirely written in Scala.

The following are some of Scala's features, as noted from its authors:

- It's a high-level language;
- It's statically typed;
- Its syntax is concise but still readable – we call it *expressive*;
- It supports the object-oriented programming (OOP) paradigm;
- It supports the functional programming (FP) paradigm;
- It has a sophisticated type inference system;
- Scala code results in .class files that run on the Java Virtual Machine (JVM);
- It's easy to use Java libraries in Scala.[15]

[11] I.e. transformed in a format that is suited to be understood by systems other than PostgreSQL; see [7, Chapter 48] for details.

[12] See [2].

[13] In fact, a single Debezium Connector cannot simultaneously connect to multiple databases.

[14] See 3.2.1.

[15] From [9, Prelude: A Taste of Scala].

Personally, I have learned Scala whilst fulfilling the curricular internship mentioned in Chapter 1; a learning process which I remarkably enjoyed.

Chapter 3

Remarks on the Data

We will assume the following, without proof:

Assumption 1. source $[*]$ is inherently consistent.

The consistency of the source database should be assumed because a replication of a database can only be as consistent as the original database, i.e. there is no way to have consistent data, starting from inconsistent data.

Additionally, such is safe to assume, since the database constraints are checked at the source by the PostgreSQL server.

Generally, $\text{dest}\left[\alpha\right] := \pi_S \left(\text{source}\left[\alpha\right]\right)$ over some subset S of its columns, with one exception for table γ:

$$\text{dest}\left[\gamma\right] := \pi_{\text{dest}[\alpha].\text{id},\text{source}[\gamma].*} \left(\text{source}\left[\gamma\right] \bowtie_{\alpha.\text{email}=\gamma.\text{email}} \text{dest}\left[\alpha\right]\right)$$

3.1 Aggregations

$$\begin{aligned} &\text{dest}\left[\eta\right] := M\big(\sigma^*_{n:=\text{count}(*),C:=\sum\text{cost}}\big(\\ &\quad G_{\text{user, time, class}}\big(\text{source}\left[\eta\right]\big) \\ &\big)\big) \end{aligned} \tag{3.1}$$

3.2 On Constraints

3.2.1 On Foreign Key Constraints

Chapter 4

Time Travel Functionality

4.1 Updates to Primary Keys

delete + insert

4.2 Empty Validity Ranges

When dealing with ranges, one should account for the fact that $\emptyset$ is indeed a valid range:

$$[t, t) = \emptyset, \forall t$$

This has the inherent meaning, in our domain, that two different records for the same table were issued at the exact same time t.

At first, this may seem an issue that could possibly be dismissed as very unlikely. Nevertheless, the presence of database transactions makes the significance of this scenario become evident. Within a transaction, different queries are evaluated and later performed as a single atomic[1] operation, thus resulting in several records being issued at the same instant.

using database system clock ensures positive time drift

[1]??? explain atomic

Chapter 5

Future Work

5.1 Keeping up to date

With the source db changes

5.2 Metrics

5.3 Batching

5.4 Squashing Transactions Together

With Kafka Streams

Chapter 6

Conclusion

Bibliography

[1] *Apache Kafka 2.7 Documentation.* Apache Software Foundation. URL: `http://kafka.apache.org/27/documentation.html` (visited on January 12, 2021).

[2] Debezium Community. *Debezium Documentation.* Red Hat. URL: `https://debezium.io/documentation/reference/1.3/index.html` (visited on January 12, 2021).

[3] Adam Jacobs. “The Pathologies of Big Data”. In: *ACM Queue* 7 (6). URL: `https://queue.acm.org/detail.cfm?id=1563874` (visited on January 12, 2021).

[4] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Transactions on Computer Systems 16* (May 1998).

[5] *Linux Programmer’s Manual.* Linux Foundation. URL: `https://man7.org/linux/man-pages/` (visited on January 12, 2021).

[6] Eric Murphy. *Distributed Data for Microservices – Event Sourcing vs. Change Data Capture.* Debezium, Red Hat. URL: `https://debezium.io/blog/2020/02/10/event-sourcing-vs-cdc/` (visited on January 12, 2021).

[7] *PostgreSQL 12 Documentation.* The PostgreSQL Global Development Group. URL: `https://www.postgresql.org/docs/12/` (visited on January 12, 2021).

[8] Gehrke J. Ramakrishan R. *Database Management Systems.* Third edition. McGraw-Hill, 2003.

[9] *Scala Book.* URL: `https://docs.scala-lang.org/overviews/scala-book/introduction.html` (visited on January 12, 2021).

[10] University of Trento. *FAQ per tirocini avviati tramite ufficio Job Guidance.* Italian. March 23, 2020. URL: `https://www.unitn.it/alfresco/download/workspace/SpacesStore/e9af2533-6b75-4f2b-844e-6c3bd3828ceb/FAQ_TIROCINI_04_06_2020.pdf` (visited on January 12, 2021).

Appendix A

Listings of Source Files

A selection of the source code that runs the system presented in this Theses, is here listed.

In previous chapters, the name of `class` has been assigned to several relation attributes; note that since `class` is a reserved keyword in Scala, it will be replaced by `klass` throughout this Appendix, without implying any change of meaning.

Listing A.1: source/eu.spaziodati.metrics/streams/StreamsLoader.scala

```scala
package eu.spaziodati.metrics.streams


class StreamsLoader {

}

object StreamsLoader {
  def main(args: Array[String]): Unit = {
    val configPath = args(0)
    val streams = List[StartableStream](
      new kappaStream,
      new etaStream,
      new thetaStream,
      new lambdaStream,
    )
    var threads = List[Thread]()
    for (stream <- streams) {
      Runtime.getRuntime.addShutdownHook(stream.
    getShutdownHook)
      threads ::= new Thread(stream.name) {
          override def run(): Unit = {
            stream.start(configPath)
          }
        }
    }
    for (thread <- threads) {
      thread.start()
    }
    for (thread <- threads) {
      thread.join()
    }
  }
}
```

Listing A.2: source/eu.spaziodati.metrics/streams/etaStream.scala

```
package eu.spaziodati.metrics.streams

 . . .

class etaStream extends StartableStream {
  override val config: etaConfig = new etaConfig
  override val logger: Logger = LoggerFactory.getLogger(
    getClass)
  override val name = "eta"

  override def createStreamTopology(builder:
    StreamsBuilder): Topology = {
    builder.stream(config.inputTopic, Consumed.`with`(new
     JsonSerde[etaDebeziumKey], new JsonSerde[Debeziumeta
    ]))
      .filter(
        (_, record) =>
          record != null &&
            record.after.isDefined &&
            record.after.get.klass.isDefined &&
            record.after.get.status.isDefined &&
            record.after.get.cost.isDefined &&
            record.after.get.user.isDefined &&
            // Filter out rows that are not finalised, i.
    e. will get updated
            // status == "done", or "error" if the row is
     finalised.
            List("done", "error").contains(record.after.
    get.status.get)
      )
      .map(
        (key, record) => KeyValue.pair(
          key.id, record.after.get
        )
      )
      .groupBy(
        (_, record: eta) => etaDebeziumOutputKey(
          record.user.get,
          record.day.toLocalDate,
          record.klass.get,
        ),
        Grouped.`with`(
          new JsonSerde[etaDebeziumOutputKey],
          new JsonSerde[eta]
        )
      )
      .aggregate(
        () => new Outputeta(-1, LocalDate.now(), "", 0, 0),
        (aggKey: etaDebeziumOutputKey, newValue: eta, aggValue: Outputeta) => {
          if (aggValue.user == -1) {
            aggValue.user = aggKey.user
            aggValue.day = aggKey.day
            aggValue.klass = aggKey.klass
          }
          aggValue.n += 1
          aggValue.C += newValue.cost.get
          aggValue
        },
        Materialized.`with`(
          new JsonSerde[etaDebeziumOutputKey],
          new JsonSerde[Outputeta],
        )
      )
      .toStream()
      .mapValues(
        record => new etaDebeziumOutput("upsert", record),
      )
      .to(config.outputTopic, Produced.`with`(new JsonSerde[etaDebeziumOutputKey], new JsonSerde[etaDebeziumOutput]))
    builder.build()
  }
}
```

Listing A.3: source/eu.spaziodati.metrics/streams/kappaStream.scala

```
package eu.spaziodati.metrics.streams

 . . .

class kappaStream extends StartableStream {
  override val config: kappaConfig = new kappaConfig
  override val logger: Logger = LoggerFactory.getLogger(getClass)
  override val name = "kappa"

  override def createStreamTopology(builder: StreamsBuilder): Topology = {
```

```scala
builder.stream(config.inputTopic, Consumed.`with`(new JsonSerde[kappaDebeziumKey], new JsonSerde[Debeziumkappa]))
  .filter(
    (_, record) =>
      record != null &&
        record.after.isDefined &&
        record.after.get.klass.isDefined &&
        record.after.get.user.isDefined &&
        record.after.get.sub_class.isDefined &&
        // Only interested in "credits_buckets" (-> "c") category.
        record.after.get.klass.get == "c"
  )
  // Only interested in records that have consumed some credits.
  .filter(
    (_, record) => {
      val Right(postSave) = circe.jawn.decode[kappaPostSave](record.after.get.post_save)
      postSave.cost.isDefined
    }
  )
  // Extract `after` field and source table ID
  .map(
    (key, record) => KeyValue.pair(
      key.id, record.after.get
    )
  )
  .groupBy(
    (_, record: kappa) => kappaDebeziumOutputKey(
      record.user.get,
      record.day.toLocalDate,
      record.klass.get,
      record.sub_class.get,
    ),
    Grouped.`with`(
      new JsonSerde[kappaDebeziumOutputKey],
      new JsonSerde[kappa]
    )
  )
  .aggregate(
    () => new Outputkappa(-1, LocalDate.now(), "", "", 0),
    (aggKey: kappaDebeziumOutputKey, newValue: kappa, aggValue: Outputkappa) => {
          if (aggValue.user == -1) {
            aggValue.user = aggKey.user
            aggValue.day = aggKey.day
            aggValue.klass = aggKey.klass
        aggValue.sub_class = aggKey.sub_class
       }
       val Right(postSave) = circe.jawn.decode[kappaPostSave](newValue.post_save)
       aggValue.C += postSave.cost.getOrElse(0)
       aggValue
      },
    Materialized.`with`(
     new JsonSerde[kappaDebeziumOutputKey],
     new JsonSerde[Outputkappa]
    )
   )
   .toStream()
   .mapValues(
    record => new kappaDebeziumOutput("upsert", record),
   )
   .to(config.outputTopic, Produced.`with`(new JsonSerde[kappaDebeziumOutputKey], new JsonSerde[kappaDebeziumOutput]))
  builder.build()
 }
}
```

Listing A.4: source/eu.spaziodati.metrics/streams/lambdaStream.scala

```
package eu.spaziodati.metrics.streams

 . . .

class lambdaStream extends StartableStream {
 override val config: lambdaConfig = new lambdaConfig
 override val logger: Logger = LoggerFactory.getLogger(getClass)
 override val name = "lambda"

 override def createStreamTopology(builder: StreamsBuilder): Topology = {
  builder.stream(config.inputTopic, Consumed.`with`(
   new JsonSerde[lambdaDebeziumKey],
   new JsonSerde[Debeziumlambda]
  ))
   .filter(
    (_, record) =>
     record != null &&
      record.after.isDefined
   )
   .map(
    (key, record) => KeyValue.pair(
     key.id, record.after.get
    )
```

```scala
      )
      .groupBy(
        (_, record: lambda) => lambdaDebeziumOutputKey(
          record.email,
          record.day.toLocalDate,
          record.klass,
        ),
        Grouped.`with`(
          new JsonSerde[lambdaDebeziumOutputKey],
          new JsonSerde[lambda]
        )
      )
      .aggregate(
        () => new Outputlambda("", LocalDate.now(),"", 0),
        (aggKey: lambdaDebeziumOutputKey, newValue: lambda, aggValue: Outputlambda) => {
          if (aggValue.email == "") {
            aggValue.email = aggKey.email
            aggValue.day = aggKey.day
            aggValue.klass = aggKey.klass
          }
          aggValue.n += 1
          aggValue
        },
        Materialized.`with`(
          new JsonSerde[lambdaDebeziumOutputKey],
          new JsonSerde[Outputlambda]
        )
      )
      .toStream()
      .mapValues(
        record => new lambdaDebeziumOutput("lambda", record),
      )
      .to(config.outputTopic, Produced.`with`(
        new JsonSerde[lambdaDebeziumOutputKey],
        new JsonSerde[lambdaDebeziumOutput]
      ))
    builder.build()
  }
}
```

Listing A.5: source/eu.spaziodati.metrics/streams/thetaStream.scala

```scala
package eu.spaziodati.metrics.streams

 . . .

class thetaStream extends StartableStream {
  override val config: thetaConfig = new thetaConfig
  override val logger: Logger = LoggerFactory.getLogger(getClass)
  override val name = "theta"

  override def createStreamTopology(builder: StreamsBuilder): Topology = {
    builder.stream(config.inputTopic, Consumed.`with`(
        new JsonSerde[thetaDebeziumKey],
        new JsonSerde[Debeziumtheta]
    ))
      .filter(
        (_, record) =>
          record != null &&
            record.after.isDefined &&
            record.after.get.user.isDefined &&
            record.after.get.status.isDefined &&
            // Filter out rows that are not finalised, i.e. will get updated
            // status == "done", or "error" if the row is finalised.
            List("done", "error").contains(record.after.get.status.get)
      )
      .map(
        (key, record) => KeyValue.pair(
          key.id, record.after.get
        )
      )
      .groupBy(
        (_, record: theta) => thetaDebeziumOutputKey(
          record.user.get,
          record.day.toLocalDate,
        ),
        Grouped.`with`(
          new JsonSerde[thetaDebeziumOutputKey],
          new JsonSerde[theta]
        )
      )
      .aggregate(
        () => new Outputtheta(-1, LocalDate.now(),0),
        (aggKey: thetaDebeziumOutputKey, newValue: theta, aggValue: Outputtheta) => {
          if (aggValue.user == -1) {
            aggValue.user = aggKey.user
            aggValue.day = aggKey.day
          }
          aggValue.n += 1
          aggValue
        },
        Materialized.`with`(
          new JsonSerde[thetaDebeziumOutputKey],
          new JsonSerde[Outputtheta]
        )
      )
      .toStream()
      .mapValues(
        record => new thetaDebeziumOutput("upsert", record),
      )
      .to(config.outputTopic, Produced.`with`(
        new JsonSerde[thetaDebeziumOutputKey],
        new JsonSerde[thetaDebeziumOutput]
      ))
     builder.build()
  }
}
```

Listing A.6: source/eu.spaziodati.metrics/connectors/PostgresSinkTask.scala

```
package eu.spaziodati.metrics.connectors

 . . .

class PostgresSinkTask extends SinkTask {
  private var connection: Connection = _
  private val statementsCache: mutable.Map[(String, String), PreparedStatement] = mutable.Map[(String, String), PreparedStatement]()
  private val logger = LoggerFactory.getLogger(getClass)
  private val timeTravelTables = List("alpha", "beta", "mu")

  override def start(propsJ: util.Map[String, String]): Unit = {
    logger.info("Running with scala version: " + scala.util.Properties.scalaPropOrElse("version.number", "unknown"))
    val props = propsJ.asScala
    val properties = new Properties()
    properties.setProperty("stringtype", "unspecified")
    // Line below doesn't actually do anything, but
    // prevents the Driver from not getting into the Uber-Jar
    DriverManager.registerDriver(
      new org.postgresql.Driver()
    )
    connection = DriverManager.getConnection(props(PostgresSinkConfigConstants.URL), properties)
  }

  override def stop(): Unit = {
```

```scala
    connection.close()
  }

  private def makeQuestionMarks(n: Int): String = List.tabulate(n)(_ => '?').mkString(",")

  private def bindParameters(stmt: PreparedStatement, values: Iterable[AnyRef]): PreparedStatement = {
    var index = 1
    values.foreach({
      case value: ArrayList[String] =>
        val valueArray = connection.createArrayOf("varchar", value.toArray())
        stmt.setArray(index, valueArray)
        index += 1
      case value =>
        stmt.setObject(index, value)
        index += 1
    })
    stmt
  }

  private def doUpsert(table: String, record: Map[String, AnyRef], key: Map[String, AnyRef]): Unit = {
    val stmt = statementsCache.getOrElseUpdate(
      (table, "upsert"),
      connection.prepareStatement(
        s"""  INSERT INTO $table("${record.keys.mkString("\",\"")}")
           |  VALUES (${makeQuestionMarks(record.size)})
           |  ON CONFLICT ("${key.keys.mkString("\",\"")}") DO UPDATE
           |    SET "${record.keys.mkString("\"= ?,\"")}" = ?
           |    WHERE $table."${key.keys.mkString("\"= ? AND " + table +".\"")}" = ?;
           |""".stripMargin
      )
    )
    bindParameters(stmt, record.values ++ record.values ++ key.values).execute()
  }

  private def doUpsertTimeTravel(table: String, recordWithNulls: Map[String, AnyRef], key: Map[String, AnyRef]): Unit = {
    val record = recordWithNulls.filter {
      case (_, v) => v != null
    }
    val stmt = statementsCache.getOrElseUpdate(
      (table, s"tt:upsert(${record.keys.mkString(",")})"),
      connection.prepareStatement(
        // N.B. With autocommit enabled (such is the default)
        // these statements execute within a transaction.
        s"""  UPDATE $table
           |  SET validity = tstzrange(lower(validity), now())
           |  WHERE ${key.keys.mkString("= ? AND ")} = ?
           |    AND current_timestamp <@ validity
           |    AND NOT EXISTS (
           |      SELECT 1 FROM $table
           |      WHERE ${record.keys.mkString("= ? AND ")} = ?
           |        AND current_timestamp <@ validity
           |    );
           |
           |  INSERT INTO $table (${record.keys.mkString(", ")}, validity)
           |  SELECT ${makeQuestionMarks(record.size)}, tstzrange(now(), 'infinity', '[)')
           |  WHERE NOT EXISTS (
           |    SELECT 1 FROM $table
           |    WHERE ${record.keys.mkString("= ? AND ")} = ?
           |      AND current_timestamp <@ validity
           |  );
           |""".stripMargin
      )
    )
    bindParameters(stmt,
      // UPDATE
      key.values ++ record.values ++
      // INSERT
      record.values ++ record.values
    ).execute()
  }

  private def doUpsertlambda(table: String, record: Map[String, AnyRef], key: Map[String, AnyRef]): Unit = {
    val stmt = statementsCache.getOrElseUpdate(
      (table, "email to id"),
      connection.prepareStatement(
        s"SELECT public.user_email_to_id(?) AS user_id;"
      )
    )
    val rs = bindParameters(stmt, List(record("user_id"))).executeQuery()
    rs.next()
  val userId = rs.getObject("user_id")
  if (userId == null) {
    logger.error(s"Got null while converting email '${
 record("user_id")}', in doUpsertlambda.")
  }
  else {
    val mutableRecord = mutable.Map(record.toSeq: _*)
    mutableRecord("user_id") = userId
    val mutableKey = mutable.Map(key.toSeq: _*)
    mutableKey("user_id") = userId
    doUpsert(table, Map(mutableRecord.toSeq: _*), Map(
 mutableKey.toSeq: _*))
  }
}

private def doDelete(table: String, key: Map[String,
 AnyRef]): Unit = {
  val stmt = statementsCache.getOrElseUpdate(
    (table, "delete"),
    connection.prepareStatement(
      s"""  DELETE FROM $table
         |  WHERE ${key.keys.mkString("= ? AND ")} = ?;
 """.stripMargin
    )
  )
  bindParameters(stmt, key.values).execute()
}

private def doDeleteTimeTravel(table: String, key: Map[
 String, AnyRef]): Unit = {
  val stmt = statementsCache.getOrElseUpdate(
    (table, "tt:delete"),
    connection.prepareStatement(
      s"""  UPDATE $table
         |  SET validity = tstzrange(lower(validity),
 now())
         |  WHERE ${key.keys.mkString("= ? AND ")} = ?
         |    AND current_timestamp <@ validity;
         |""".stripMargin
    )
  )
  bindParameters(stmt, key.values).execute()
}

private def extractAfterRecord(data: mutable.Map[Object
 , Object]): Map[String, AnyRef] = {
  data("after").asInstanceOf[util.Map[Object, Object]].
 asScala.map { case (k, v) =>
    (k.toString, v)
  }.toMap
```

```scala
  }

  private def extractTs(data: mutable.Map[Object, Object]): Long = {
    data("source").asInstanceOf[util.Map[Object, Object]].asScala.map {
      case (k, v) => (k.toString, v)
    }.map {
      case ("ts_ms", ts) => Some(ts.asInstanceOf[Long])
      case _ => None
    }.filter(x => x.isDefined).head.get
  }

  override def put(recordsJ: util.Collection[SinkRecord]): Unit = {
    val records = recordsJ.asScala
    var table = "unknown"
    for (record <- records) {
      table = record.topic()
        .split("\\.").takeRight(2).mkString(".")  // Avoid the Debezium's assigned database name
      val key = record.key().asInstanceOf[util.Map[String, AnyRef]].asScala.toMap
      val json = record.value().asInstanceOf[util.Map[Object, Object]].asScala
      if (json != null) {  // Ignore Debezium tombstone events
        var data = Map[String, AnyRef]()
        try {
          data = extractAfterRecord(json)
        }
        catch {
          case e: NullPointerException => // there is no `after` field; discard.
        }
        json("op") match {
          case "c" | "r" =>
            if (timeTravelTables.contains(table)) {
              doUpsertTimeTravel(table, data, key)
            }
            else {
              doUpsert(table, data, key)
            }
          case "u" =>
            if (timeTravelTables.contains(table)) {
              doUpsertTimeTravel(table, data, key)
            }
            else {
              doUpdate(table, data, key)
            }
          case "d" =>
            if (timeTravelTables.contains(table)) {
              doDeleteTimeTravel(table, key)
            }
            else {
              doDelete(table, key)
            }
          case "upsert" => doUpsert(table, data, key)
          case "lambda" => doUpsertlambda(table, data,
    key)
        }
      }
    }
    if (records.nonEmpty) {
      logger.info(s"Processed ${records.size} records for
     table $table.")
    }
  }

  override def version(): String = "1.0.0"
}
```