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
Prof. Gabriel Mark Kuper

Student
Daniele Parmeggiani

Academic year 2020/2021

Dedication

Abstract

This Thesis presents work carried out during my curricular internship, which describes 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 storing 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 to pertain the work I've carried out within SpazioDati, all information relevant to their intellectual property has been anonymized so as to not disclose it. The methods for such anonymization are portrayed in §1.1. Nowhere in this document is their customers' data shown or cited, partly, or in whole.

Contents

Chapter 1

Introduction

The work presented in this Thesis pertains to 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 [14] for further details.

[2] Description via `https://datarade.ai/data-providers/spaziodati/profile`, last visited February 18, 2021.

[3] Ibid.

1.1 Non Disclosure

In this document information internal to SpazioDati is referenced. In order to avoid disclosing information relevant to their intellectual property, the relational tables later referenced in this Thesis 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 Thesis 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 Thesis. 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 [9].

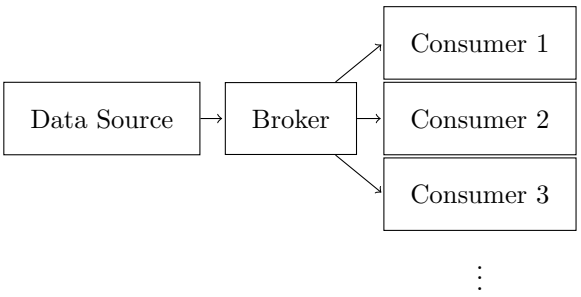


Figure 1.1: A Generic CDC System.

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 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 Thesis, 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. 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.

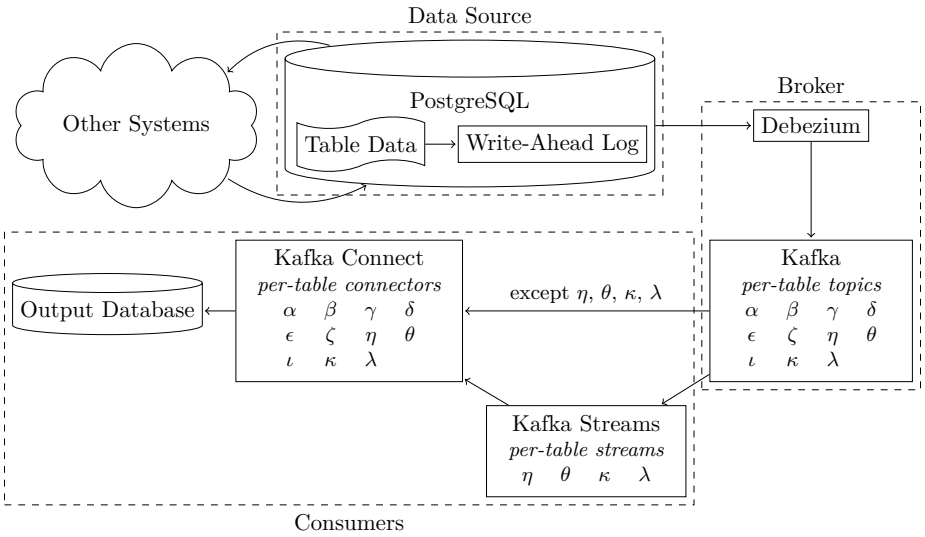


Figure 1.2: An Outline of the CDC System presented in this Thesis.

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 [12, Chapter 4].

Additionally, further notation will be introduced in order to allow for clarity between table representations, as well as 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	$\sum n$
a	4

m	$\sum n$
b	5

(d)

m	$\sum n$
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 [12, 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

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

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 $M\Big(\sigma^*_{\sum n}\big(G_m(\omega)\big)\Big)$, 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. □

Having given evidence for a necessity of an extension to Relational Algebra in order to express aggregations, let us borrow standard notation for use throughout the next chapters. Consider the γ operator as defined in [2, §5.2.4]. It is equivalent to the composition of operators previously defined in this section; for instance consider Example (3) and see that

$$\gamma_{m,\sum n} = M\Big(\sigma^*_{\sum n}\big(G_m(\omega)\big)\Big) \ .$$

[10]Here "aggregation functions" is used to indicate some functions that produce a single result from an input set. 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. [7].

[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.

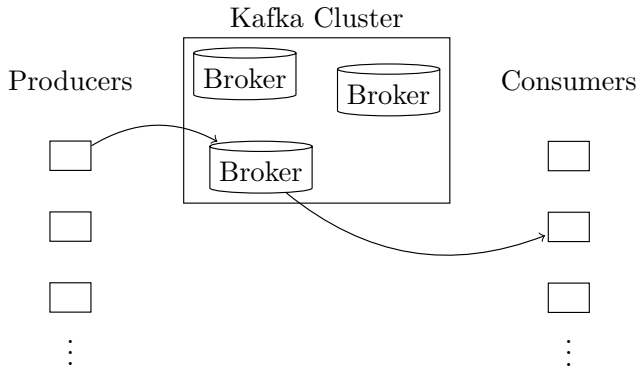


Figure 2.1: General overview of Kafka Architecture.

in part composed by Kafka, to meet increasing computational demands. This 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 [3, §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 [8, `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. [6].

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 a 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 this notion, nonetheless I will try to convince the reader that this 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] 3, §8.1.

For instance, let us consider table η. In Query (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, the 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

While a detailed discussion of the several technologies involved in operating the presented system is the intent of this Chapter, because of the fact that the great majority of the intended audience for this document already has knowledge on this topic and that the discussion itself would be too broad, an explanation of what a relational database is, is intentionally left to the several books[8] that already provide a thorough discussion.

Nonetheless, in this Section, particular aspects of the PostgreSQL relational database will be presented, when they directly pertain contents of this Thesis.

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 [12, §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,

[6] Recall Definition 1.
[7] Recall Definition 3.
[8] E.g. [12], [2].
[9] Except when related to the persistent storage itself. Evidently, should such be the nature of hardware failure, no action can provide data integrity.

> 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 [10, Chapter 29].

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;

[10] From [10, §29.2]

[11] I.e. transformed in a format that is suited to be understood by systems other than PostgreSQL; see [10, Chapter 48] for details.

[12] See [5].

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

[14] See 3.2.1.

- 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]

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

[15] From [13, Prelude: A Taste of Scala].

Chapter 3

Remarks on the Data

In this Chapter several properties of the data at hand will be explored.

Due to specific interests on the particular contents of tables, no exact replica of each table is to be found in the output database. Instead, every table is processed before being sent to its destination, in order to compose the output database to be most useful to the marketing team, whilst considering storage efficiency. In particular, three classes of tables will be introduced, depending on the nature of the transformation carried out: plain, time-traveled, and aggregated.

Plain tables are the simplest to discuss since they can be defined as a projection over a subset S of the source table's columns; formally, for some plain table ω:

$$\text{dest}\,[\omega] := \pi_S\big(\text{source}\,[\omega]\big)\ .$$

Aggregated and time-traveled tables instead require a more detailed discussion, thus let us introduce them here and discuss them in detail in §3.1 and Chapter 4 respectively. As the name suggests, aggregated tables are defined as an aggregation of data contained in the source table. Time-traveled tables are a peculiar representation of their respective source table: they not only retain a current view of the source table, but also store all changes that have occurred to it.

3.1 Aggregated Tables

In this Section, the four aggregated tables that are considered in this CDC system will be presented and the destination tables will be defined in terms of Relational Algebra queries performed on the source tables. Extensive use of the grouping extension to Relational Algebra introduced in §1.5.2 will be made. In addition, several references to code implementing the following aggregations, can be found in this Section.

$$\begin{aligned}\text{dest}\,[\eta] := \gamma_{\text{user, day, class, count}(*)\to n,\ \sum(\text{cost})\to C}\big(\\ \sigma_{\text{isFinal}(*)}\big(\text{source}\,[\eta]\big)\\ \big)\end{aligned} \tag{3.1}$$

$$\begin{aligned} \text{dest}\,[\theta] := \gamma_{\text{user, day, count}(*)\to n}\big(\\ \quad \sigma_{\text{isFinal}(*)}\big(\text{source}\,[\theta]\big) \\ \big) \end{aligned} \tag{3.2}$$

Let us begin with two very similar aggregations. For these tables, we are aggregating operations count and cost based on the user who performed them, the day, and the class of operation, with the latter only being relevant for η.

For these aggregations, let us introduce a new function used to filter out data change events emitted from non-finalized rows: the function $\text{isFinal}\,(*)$ yields truth if the evaluated row will not undergo changes anymore, and it is thus finalized. This aspect is important because if non finalized rows were considered, they would be counted more than once towards the aggregation. For instance, say that some row goes from state A to B: two data change events would be emitted and n would be increased by 2, which is not the intended behavior. In Listings A.2, line 22, and A.5, line 23, you can see this function implemented: rows that are not in a *done* or *error* state are not finalized, thus they are filtered out.

In Figure 3.1 a visualization of Query (3.1)'s aggregation processing is presented in terms of Kafka Streams APIs used. In Kafka's terminology, the content of this Figure is the stream's *topology*; in fact, the data for producing Figure 3.1 was obtained making again use of its APIs.[1] Very similar results were obtained for all other aggregations in this Section.

Comparison of said Figure with code in Listing A.2 yields a direct relationship between methods used and the topology description. In fact they are lines 10 – 63 that generate the above topology. Let us make a detailed comparison only for table η: other tables perform very similar operations.

Consider line 12: the `filter` method is the first filter step in the topology, which implements $\sigma_{\text{isFinal}(*)}$, along with other required technical checks that we may leave out of the discussion.

Going further, on line 24 the `map` method is reflected as the next step after `filter` also on Figure 3.1, again this is used for primarily technical reasons.

Then, the `groupBy` method on line 29, which acts in a similar fashion as SQL's `GROUP BY` clause, is expanded into three steps which eventually output into a new repartitioning topic that Kafka Streams creates on its own. It is used to allow a new keying of the stream, i.e. the source's primary-key is exchanged for the grouping keys.[2]

Next, the `aggregate` method, on line 40, acts similarly to operator σ^*[3] and actually carries out the computation of counting records and summing the cost.

On line 57, the following method `toStream` may seem strange, given that we have since been operating on a stream. Its necessity comes from the fact that Kafka when performing the repartitioning caused by `groupBy`, represents the new data as a table, in order to efficiently carry out the upcoming computations, making again use of the duality between streams and tables. At this point, there is no longer the need to do so, hence the method call.

[1] See [4, Topology#describe]: `https://kafka.apache.org/27/javadoc/org/apache/kafka/streams/Topology.html#describe--`, last visited February 10, 2021.

[2] A computation similar to Definitions (1), and (2).

[3] See Definition (3).

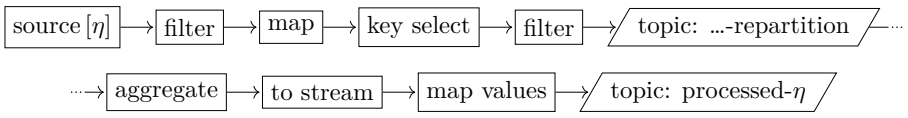


Figure 3.1: Processing of aggregation Query (3.1) in terms of Kafka Streams APIs used.

Finally on lines 58 – 61, `mapValues` renders the data in a manner that the Kafka Connect Sink developed can understand and use to transfer data to the output database, and the `to` method indicates which topic to output the computed records to.

In summary, we have shown how Query (3.1) is transformed into code for Kafka Streams.

$$\begin{aligned}\text{dest}\,[\kappa] := \gamma_{\text{user, day, class, subclass, } \sum(\text{cost})\to C}\big(\\ \quad \sigma_{\text{class}=\texttt{'c'}}\big(\text{source}\,[\kappa]\big) \\ \big)\end{aligned} \tag{3.3}$$

One peculiar aspect of the dest $[\kappa]$ table is the presence of a *class* attribute, which appears in Query (3.3). From the query itself one can rightly argue that since only rows with *class* = `'c'` are kept, the attribute should be discarded altogether: since it's always the same value, it bears no meaningful data. The rationale for keeping the *class* attribute in the output database was that possible future interventions on the system, guided by a change of interest from the marketing team, would require said attribute. Thus, it was decided to have it in the destination table to facilitate possible future works.

$$\begin{aligned}\text{dest}\,[\lambda] := \pi_{\text{dest}[\alpha]\text{.id, day, class, } n}\big(\\ \quad \text{dest}\,[\alpha] \bowtie \gamma_{\text{email, day, class, count}(*)\to n}\big(\\ \quad\quad \text{source}\,[\lambda] \\ \big)\big)\end{aligned} \tag{3.4}$$

The λ table arguably bears the most interesting aggregation query. Not only source data is aggregated, but it is also enriched after aggregation: see join operation in second line of Query (3.4).

The motive for this operation lies in the fact that source $[\lambda]$, for historical reasons, holds the user's email instead of its identifier. In order to have more coherence in the output database, it was decided that in the processing of this table, the email would be converted into an identifier. This is reasonable because a single email is ever only associated with one user.

From the query, dest $[\alpha]$ is used to perform the join operation. This is formally incorrect: as one might expect, if a user is registered and the associated data change has not been processed by the CDC system before a record from source $[\lambda]$ is processed, then the join operation would fail.

While this is true, one should consider that it is very unrealistic. In fact, because of Atoka's application logic, there is a sensible temporal distance between the registration of a user and the first time a record can be emitted from

source $[\lambda]$ pertaining that user. In addition, performing the join query on the source database, which would yield a formally correct operation, would impact the data source's performance, which is undesirable. Therefore, on the grounds of common sense, it was decided to use the more efficient option.

3.2 On Constraints

Enforcing constraints on data is meant to provide consistency to a database. That is, by restricting the set of possible values down to a more sensible subset, the data contained in the database can make more sense. For instance, we may not allow to have users under legal age to be represented in a database, thus we would reduce an *age* attribute from the set of integers down to the set of integers greater than or equal to 18.

With regards to relational databases the constraints generally considered are the following:

1. uniqueness (including primary-key),
2. conditional (including non-null), and
3. referential integrity (foreign-key).

Let us discuss how these constraints are carried from the data source over to the output database, which is a property we are interested in. To do so formally, let the following be assumed:

Assumption 1. source $[*]$ is consistent.[4]

Considering Assumption (1), let us examine the three constraint classes enumerated above. With regards to the first two, the fact that they are carried over to the output database really is a corollary to Assumption (1). Consider some table ω: if the constraint holds for source $[\omega]$ and the data contained in records streamed is received unmodified,[5] then for each unique value in some attribute of source $[\omega]$ there is a unique value in dest $[\omega]$. Similarly, for each condition on any value in some attribute of source $[\omega]$, that condition holds for any respective value in dest $[\omega]$.

With regards to the referential integrity (or foreign-key) constraint, the fact that it holds in the output database is no corollary to Assumption (1). Considering also the amount of foreign-key relations between the tables we are interested in, depicted in Figure 3.2, let us have a lengthier discussion.

[4]Meaning: every constraint which has been set at schema definition, holds. The source database's consistency should be assumed because a database replication can only be as consistent as the original database, i.e. there is no sensible mechanism to reach data consistency, starting from inconsistent data. Additionally, such is safe to assume, since the database constraints are checked at the source by the PostgreSQL server.

[5]This is true for every table that doesn't undergo any processing in the stream, therefore it is not true for aggregated tables. Nonetheless, given the above definition queries, we can state that there is no modification of values, thus the following reasoning holds for aggregated tables as well. This is also true for time-traveled tables.

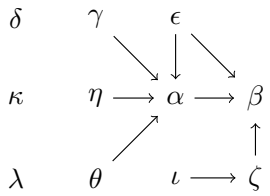


Figure 3.2: Foreign Key Relational Dependencies.

3.2.1 On Foreign Key Constraints

One particular aspect on the processing of streams is that they are processed in parallel. That is, records pertaining to one table are processed in succession,[6] but records pertaining to different tables are processed independently of each other, possibly at the same time.

The problem, depicted in Figure 3.3, which arises with foreign-keys is that, given the parallelism of streams processing, a record may fail to be processed on the grounds that a referenced value has not been processed yet, thus resulting in an error raised by the output database. Therefore, we want records that have this particular requirement to be processed after those records that are required for the destination database's consistency.

Formally, let us denote with $r(f^*)$ a record that contains a reference to some attribute f which is part of another relation, and let $s(f)$ be the record containing the value in attribute f that is referenced in r. A processing failure on r, that is due to a foreign-key constraint, means that the database system is refusing to process r on grounds that the referenced value f^* is not to be found in the relation containing attribute f.

Proposition 2. *Let $r(f^*)$ be a record containing a foreign-key constraint on f. Let r fail to be processed due to the constraint on f. Then*

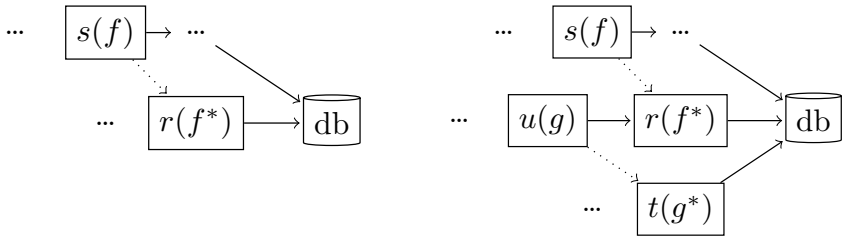
1. *$s(f)$ is an existing record containing the value referenced in $r(f^*)$; and*
2. *$s(f)$ is in the unprocessed part of the stream.*

Proof. (1) Let source $[\omega]$ be the relation which emitted record r and source $[\psi]$ the table referenced in source $[\omega]$ that contains attribute f. For a record to be emitted from a table, there has to be a data change contained in a committed transaction that has taken place in the data source. A transaction can only be committed if every functional dependency within it is satisfied. Thus, the transaction that caused the emission of r, must have had the functional dependency on f satisfied.

It follows that source $[\psi]$ contains the value referenced in $r(f^*)$, otherwise source $[*]$ would not be consistent.[7] Thus, there must be a data change event that contains the value referenced in r, in other words, $\exists s(f)$ record emitted from source $[\psi]$ that satisfies reference f^*.

[6] In particular, they have to: one of the core principles expressed in §1.4 is that data change events emitted from one table are processed sequentially, otherwise their application to the destination table is not guaranteed to be consistent with the source.

[7] Cf. Assumption (1).



(a) (b)

Figure 3.3: (a) Depiction of the foreign-key problem. (b) The same problem recurred.

(2) Since $r(f^*)$ is failing to be processed due to the foreign-key constraint, $s(f)$ cannot be in the portion of stream that has already been processed, else there would be no failure. □

Proposition (2) tells us that a solution can be found, since the record required does exist and is in fact in one of the streams.

Logically, the solution to this problem is to be found in topological sorting algorithms,[8] since the various records that present this issue would yield a dependency graph when combined. In practical terms, since data change events related to each individual table are streamed in parallel, one solution to this problem could be to have the problematic streams hold processing until the referenced records are processed, assuming that progress on every other stream is never halted. Though, this causes a notable degradation of the CDC system's performance.

After careful consideration, it was decided to relax the requirement of having foreign key constraints on the destination database, for the sake of performance. The output database would therefore not guarantee consistency (only in so far as foreign-keys are concerned), instead it would guarantee an *eventual* consistency. That is, assuming progress of every stream, given Proposition (2), the output database would become consistent once all records are processed.

[8] See for reference [1, §9.5.8].

Chapter 4

Time Travel Functionality

In this Chapter, the various concepts regarding the so called *time travel* functionality, how it pertains the presented CDC system, and the issues that arose in implementing this feature, will be discussed. Additionally, some background to this functionality will be presented in the following Section.

A time traveled table is a table in the output database that holds the information of present and past states of its source table. Let us introduce a time travel operator τ, so that for some table ω, similarly to the previous Chapter 3, we can write

$$\text{dest}\,[\omega] := \tau(\text{source}\,[\omega])\ .$$

Before giving a definition of τ, further notation is required.

Let us indicate with r_i a particular configuration of some row r's values, which belongs to table ω, such that r, identified by its primary key, is updated in source $[\omega]$ from r_i to r_{i+1}. This change of values must happen at a particular moment, thus let a data change record r_i^t convey update number i to r's values, emitted at time t.

In Figure 4.1 a visual representation of the above notation is portrayed. In particular, r_0 is the configuration of r's values at insertion time t, while the following are those configurations that r acquires, at successive updates.

With this notation, a definition of τ is possible.

Definition 5 (Time travel operator)**.** Whichever attributes is source $[\omega]$'s primary key composed of, it is extended, in dest $[\omega]$, to contain a validity range; that is, the range of time in which some configuration r_i of source $[\omega]$'s row r was the configuration that could be retrieved from the source database, when reading r; $\forall r \in \text{source}\,[\omega]$.

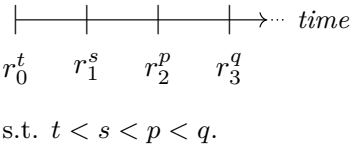


Figure 4.1: Representation of changes to a row r.

r	*validity*
0	$[t, +\infty)$

r	*validity*
0	$[t, s)$
1	$[s, +\infty)$

r	*validity*
0	$[t, s)$
1	$[s, p)$
2	$[p, +\infty)$

r	*validity*
0	$[t, s)$
1	$[s, p)$
2	$[p, q)$
3	$[q, +\infty)$

Table 4.1: Representation in dest $[\omega]$ of the changes outlined in Figure 4.1.

This validity range is expressed as a numerical range of the form $[x, y)$, such that $x, y \in T$, where T is the totally ordered set of all instants of time, present and past. For our purposes, let us say that $\pm\infty \in T$, so that a range of the form $[x, +\infty)$ is intended to mean that some configuration r_i is valid from instant x up to an unknown moment in the future, and a range of the form $(-\infty, x)$ is intended to mean that a validity ends at instant x.

Example 4. To accomplish the changes depicted in Figure 4.1, the resulting validity data in dest $[\omega]$ can be expressed as in Table 4.1. Note how receiving an update causes the old configuration to be retained, and to have its validity range updated.

Let us now draw an algorithm with which operations on the source table are mapped, using time travel logic, to operations in the output database. We do so, by defining the following Operations (4.1), (4.2), and (4.3), which exhaustively comprise all relevant classes of data change events.

$$\text{source}\,[\omega] \to \texttt{INSERT}\ r_i^t \overset{\tau}{\Rightarrow} \texttt{INSERT}\ r_i \text{ s.t. validity} := [t, +\infty) \tag{4.1}$$

$$\text{source}\,[\omega] \to \texttt{UPDATE}\ r_i^t \overset{\tau}{\Rightarrow} \left\{ \begin{array}{l} 1.\ \texttt{UPDATE}\ r_{i-1} \text{ s.t. validity} \cap (-\infty, t) \\ 2.\ \texttt{INSERT}\ r_i \text{ s.t. validity} := [t, +\infty) \end{array} \right\} \tag{4.2}$$

$$\text{source}\,[\omega] \to \texttt{DELETE}\ r_i^t \overset{\tau}{\Rightarrow} \texttt{UPDATE}\ r_{i-1} \text{ s.t. validity} \cap (-\infty, t) \tag{4.3}$$

4.1 Context on Time Travel

Historically, this feature was part of PostgreSQL. It was first part of the core system,[1] and later delivered as an extension.[2] It was eventually removed in version 12, because the `abstime` data type, crucial in the extension's implementation, was also removed in version 12.[3]

In order to bring the feature back to the output PostgreSQL database, we had the following options:

1. making use of the previously working extension code, discard the `abstime` dependency and instead use `tstzrange`;

[1] See `https://www.postgresql.org/docs/6.5/advanced23236.htm`, last visited February 10, 2021.

[2] See `https://www.postgresql.org/docs/8.3/contrib-spi.html`, last visited February 10, 2021.

[3] From [11]: "Remove the timetravel extension (Andres Freund)."

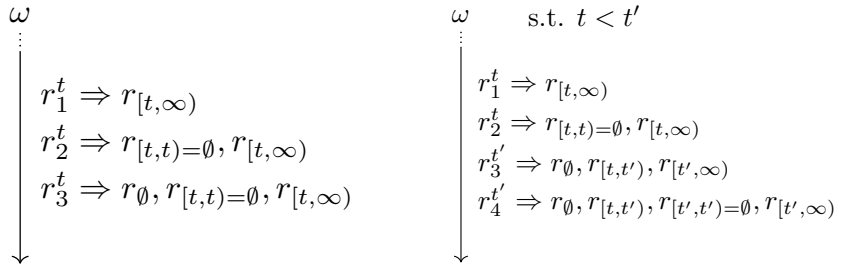


Figure 4.2: Two variations of the empty validity ranges issue occurring.

2. create a trigger, having the server handle time travel logic transparently from the clients; or

3. have highly customized `INSERT`, `UPDATE`, and `DELETE` statements.

With regards to option (1), although achievable per se, the distinct requirement of compiling the library and linking the database against it, makes utilizing AWS RDS[4] not possible, because such custom operations are understandably not allowed.

Option (2) was the one chosen at first. A custom Python trigger was implemented and able to perform Operations (4.1), (4.2), and (4.3). Nonetheless, it was later decided to discard it on the grounds that the Python procedural language was not available on RDS, albeit part of official PostgreSQL distributions, and use of other languages would have made the extension's development more time consuming.

Finally, after discarding the trigger operating on the PostgreSQL server side, option (3) was chosen, thus shifting complexity to the client. The SQL statements that provide the time travel functionality can be seen in Listing A.6: insertion and update can be found on lines 59 – 94, and deletion can be found on lines 129 – 141.

In retrospective, implementation of this feature was where the most effort was directed to.

4.2 Issues with Time Travel

In this Section, the issues encountered while developing the subsystem that provides the time travel functionality, are presented.

4.2.1 Empty Validity Ranges

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

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

[4]Amazon Web Services provides the Relational Database Service. Use of this product was considered a strict requirement for the presented CDC system.
For more information, see `https://aws.amazon.com/rds/`, last visited February 10, 2021.

This has the inherent meaning, in our domain, that two different records for the same table were issued at the exact same time t. Figure 4.2 depicts the issue described in this Section, in two variations: (a) three records originated at the same instant t, and (b) four records were emitted at two different instants.

The issue that arises is the collision of primary keys; that is, the primary key of r can be repeated multiple times, if all the relevant rows at the destination database have different validity ranges.

At first, this may seem an issue that could possibly be dismissed as very unlikely, since in order to occur it requires that records are emitted in the same moment. 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 operation,[5] thus resulting in several records being issued at the same instant; i.e. PostgreSQL writes several changes to the WAL within a single system call, and these changes may, and in certain cases do, contain multiple configurations of r.

The solution to this problem was a temporary one; that is, without any other alternatives being achievable in the internship time frame, it was decided to settle for the following solution. Instead of using the record emission time, utilize the serialization time at the destination database. In fact, using the database's system clock ensures a positive time drift. Because the various operations are performed as transactions themselves, and they are not batched together,[6] the small amount of time between the transactions is sufficient to prevent the problem presented in this Section.

A better solution is presented in §5.4.

4.2.2 Updates to Primary Keys

An update to a time traveled primary key's value is effectively indistinguishable from the consecutive application of a deletion and a subsequent insertion;[7] that is, by following Operations (4.1), (4.2), and (4.3), one cannot distinguish between the two scenarios.

Example 5. Table ϕ's primary key is a single string attribute *id*. Row $q \in \phi$, with $id =$ "Facco", is updated to have $id =$ "Fakko", at instant t, without any issues from the data source's side. Following Operation (4.2), q_{i-1} is updated to have its validity end in t, and q_i, containing the updated value for *id*, is inserted with validity starting in t.

There is no sensible mechanism that solely comprises an inquiry into the contents of dest $[\phi]$, to tell whether at time t an update operation was received for q, or rather a deletion request was received for q, in conjunction with an insertion request for a supposedly different row q'.

Of course, there is a semantic difference between an update and the operations described, even with regards to primary keys. Nonetheless, it would not

[5] From [2, §6.6.3]:

> A transaction is a collection of one or more operations on the database that must be executed atomically; that is, either all operations are performed or none are.

[6] Cfr. §5.3.

[7] Conversely, a deletion and insertion would be indistinguishable from an update, if those two operations were performed inside a transaction, as discussed in the prior Section.

be possible to correlate distinct row configurations, if primary keys were not the driver for this correlation.

Though in the application that pertains to this Thesis updates to primary keys are voluntarily dismissed as not relevant to the use case at hand, it is important to note this limitation of the approach to time data described in this Chapter.

4.2.3 Ignoring Unchanged Rows

In developing this system, another issue arose: since time traveled tables are generally defined as a projection over some subset S of the source table's columns, it is possible for a row to receive an update that contains the previous row configuration. Formally,

$$\text{dest}\,[\omega] := \tau\big(\pi_S\big(\text{source}\,[\omega]\,\big)\big) \Rightarrow \text{it is possible that } \exists r, i \text{ s.t. } r_{i-1} = r_i\ .$$

This is because when an update is performed in source $[\omega]$, PostgreSQL writes the related data onto the WAL, which is in turn read by Debezium. This process is carried out regardless of the fact that the updated value resides in a column that is not part of S, and if that is the case, the received record effectively asks to update a row to the configuration it already has.

When considering other tables, such as plain or aggregated tables, this problem can be trivially dismissed; that is, the update doesn't actually cause data in the output database to change. Nevertheless, when considering time traveled tables, what happens is that the row will be repeated with two validity ranges, the former ending in some instant t and the latter beginning at t.

Thus, while querying the output database one would be induced to think that since the two validity ranges represent a change, then the row was changed, which is not actually the case.

Although this behavior does not in fact go against Definition (5), the relatively diminished semantic significance of the validity range concept, as well as the inefficiency it entails, were sufficient reasons to develop a guarding mechanism against what effectively becomes an unnecessary duplication of data.

This guard can be found in Listing A.6, lines 72 – 76: the `NOT EXISTS` SQL clause is used to check whether, as previously expressed, $r_{i-1} = r_i$.

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] Alberto Montresor et al. *Algoritmi e strutture di dati*. Third edition. De Agostini Scuola, 2014.

[2] Hector Garcia-Molina et al. *Database Systems: The Complete Book*. Second edition. Pearson Education, 2009.

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

[4] *Apache Kafka 2.7 JavaDoc*. Apache Software Foundation. URL: `https://kafka.apache.org/27/javadoc/` (visited on January 12, 2021).

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

[6] 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).

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

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

[9] 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).

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

[11] *PostgreSQL 12 Release Notes*. The PostgreSQL Global Development Group. URL: `https://www.postgresql.org/docs/release/12.0/` (visited on January 12, 2021).

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

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

[14] 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 Thesis, 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

```
1  package eu.spaziodati.metrics.streams
2
3
4  class StreamsLoader {
5
6  }
7
8  object StreamsLoader {
9    def main(args: Array[String]): Unit = {
10     val configPath = args(0)
11     val streams = List[StartableStream](
12       new kappaStream,
13       new etaStream,
14       new thetaStream,
15       new lambdaStream,
16     )
17     var threads = List[Thread]()
18     for (stream <- streams) {
19       Runtime.getRuntime.addShutdownHook(stream.
       getShutdownHook)
20       threads ::= new Thread(stream.name) {
21           override def run(): Unit = {
22             stream.start(configPath)
23           }
24         }
25     }
26     for (thread <- threads) {
```

```
27        thread.start()
28      }
29      for (thread <- threads) {
30        thread.join()
31      }
32    }
33  }
```

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

```
1  package eu.spaziodati.metrics.streams
2
3   . . .
4
5  class etaStream extends StartableStream {
6    override val config: etaConfig = new etaConfig
7    override val logger: Logger = LoggerFactory.getLogger(
       getClass)
8    override val name = "eta"
9
10   override def createStreamTopology(builder:
       StreamsBuilder): Topology = {
11     builder.stream(config.inputTopic, Consumed.`with`(new
        JsonSerde[etaDebeziumKey], new JsonSerde[Debeziumeta
       ]))
12       .filter(
13         (_, record) =>
14           record != null &&
15             record.after.isDefined &&
16             record.after.get.klass.isDefined &&
17             record.after.get.status.isDefined &&
18             record.after.get.cost.isDefined &&
19             record.after.get.user.isDefined &&
20             // Filter out rows that are not finalised, i.
       e. will get updated
21             // status == "done", or "error" if the row is
        finalised.
22             List("done", "error").contains(record.after.
       get.status.get)
23       )
24       .map(
25         (key, record) => KeyValue.pair(
26           key.id, record.after.get
27         )
28       )
29       .groupBy(
30         (_, record: eta) => etaDebeziumOutputKey(
31           record.user.get,
32           record.day.toLocalDate,
33           record.klass.get,
34         ),
```

```
35          Grouped.`with`(
36            new JsonSerde[etaDebeziumOutputKey],
37            new JsonSerde[eta]
38          )
39        )
40        .aggregate(
41          () => new Outputeta(-1, LocalDate.now(), "", 0,
      0),
42          (aggKey: etaDebeziumOutputKey, newValue: eta,
      aggValue: Outputeta) => {
43            if (aggValue.user == -1) {
44              aggValue.user = aggKey.user
45              aggValue.day = aggKey.day
46              aggValue.klass = aggKey.klass
47            }
48            aggValue.n += 1
49            aggValue.C += newValue.cost.get
50            aggValue
51          },
52          Materialized.`with`(
53            new JsonSerde[etaDebeziumOutputKey],
54            new JsonSerde[Outputeta],
55          )
56        )
57        .toStream()
58        .mapValues(
59          record => new etaDebeziumOutput("upsert", record)
      ,
60        )
61        .to(config.outputTopic, Produced.`with`(new
      JsonSerde[etaDebeziumOutputKey], new JsonSerde[
      etaDebeziumOutput]))
62       builder.build()
63    }
64 }
```

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

```
1 package eu.spaziodati.metrics.streams
2
3  . . .
4
5 class kappaStream extends StartableStream {
6   override val config: kappaConfig = new kappaConfig
7   override val logger: Logger = LoggerFactory.getLogger(
      getClass)
8   override val name = "kappa"
9
10  override def createStreamTopology(builder:
      StreamsBuilder): Topology = {
```

```
11      builder.stream(config.inputTopic, Consumed.`with`(new
         JsonSerde[kappaDebeziumKey], new JsonSerde[
        Debeziumkappa]))
12        .filter(
13          (_, record) =>
14            record != null &&
15              record.after.isDefined &&
16              record.after.get.klass.isDefined &&
17              record.after.get.user.isDefined &&
18              record.after.get.sub_class.isDefined &&
19              // Only interested in "credits_buckets" (-> "
        c") category.
20              record.after.get.klass.get == "c"
21        )
22        // Only interested in records that have consumed
        some credits.
23        .filter(
24          (_, record) => {
25            val Right(postSave) = circe.jawn.decode[
        kappaPostSave](record.after.get.post_save)
26            postSave.cost.isDefined
27          }
28        )
29        // Extract `after` field and source table ID
30        .map(
31          (key, record) => KeyValue.pair(
32            key.id, record.after.get
33          )
34        )
35        .groupBy(
36          (_, record: kappa) => kappaDebeziumOutputKey(
37            record.user.get,
38            record.day.toLocalDate,
39            record.klass.get,
40            record.sub_class.get,
41          ),
42          Grouped.`with`(
43            new JsonSerde[kappaDebeziumOutputKey],
44            new JsonSerde[kappa]
45          )
46        )
47        .aggregate(
48          () => new Outputkappa(-1, LocalDate.now(), "", ""
        , 0),
49          (aggKey: kappaDebeziumOutputKey, newValue: kappa,
         aggValue: Outputkappa) => {
50              if (aggValue.user == -1) {
51                aggValue.user = aggKey.user
52                aggValue.day = aggKey.day
53                aggValue.klass = aggKey.klass
```

```
54                aggValue.sub_class = aggKey.sub_class
55              }
56              val Right(postSave) = circe.jawn.decode[
    kappaPostSave](newValue.post_save)
57              aggValue.C += postSave.cost.getOrElse(0)
58              aggValue
59            },
60        Materialized.`with`(
61          new JsonSerde[kappaDebeziumOutputKey],
62          new JsonSerde[Outputkappa]
63        )
64      )
65      .toStream()
66      .mapValues(
67        record => new kappaDebeziumOutput("upsert",
    record),
68      )
69      .to(config.outputTopic, Produced.`with`(new
    JsonSerde[kappaDebeziumOutputKey], new JsonSerde[
    kappaDebeziumOutput]))
70    builder.build()
71  }
72 }
```

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

```
1 package eu.spaziodati.metrics.streams
2
3  . . .
4
5 class lambdaStream extends StartableStream {
6   override val config: lambdaConfig = new lambdaConfig
7   override val logger: Logger = LoggerFactory.getLogger(
    getClass)
8   override val name = "lambda"
9
10  override def createStreamTopology(builder:
    StreamsBuilder): Topology = {
11    builder.stream(config.inputTopic, Consumed.`with`(
12      new JsonSerde[lambdaDebeziumKey],
13      new JsonSerde[Debeziumlambda]
14    ))
15      .filter(
16        (_, record) =>
17          record != null &&
18            record.after.isDefined
19      )
20      .map(
21        (key, record) => KeyValue.pair(
22          key.id, record.after.get
23        )
```

```
24        )
25        .groupBy(
26          (_, record: lambda) => lambdaDebeziumOutputKey(
27            record.email,
28            record.day.toLocalDate,
29            record.klass,
30          ),
31          Grouped.`with`(
32            new JsonSerde[lambdaDebeziumOutputKey],
33            new JsonSerde[lambda]
34          )
35        )
36        .aggregate(
37          () => new Outputlambda("", LocalDate.now(),"", 0)
    ,
38          (aggKey: lambdaDebeziumOutputKey, newValue:
    lambda, aggValue: Outputlambda) => {
39            if (aggValue.email == "") {
40              aggValue.email = aggKey.email
41              aggValue.day = aggKey.day
42              aggValue.klass = aggKey.klass
43            }
44            aggValue.n += 1
45            aggValue
46          },
47          Materialized.`with`(
48            new JsonSerde[lambdaDebeziumOutputKey],
49            new JsonSerde[Outputlambda]
50          )
51        )
52        .toStream()
53        .mapValues(
54          record => new lambdaDebeziumOutput("lambda",
    record),
55        )
56        .to(config.outputTopic, Produced.`with`(
57          new JsonSerde[lambdaDebeziumOutputKey],
58          new JsonSerde[lambdaDebeziumOutput]
59        ))
60      builder.build()
61    }
62  }
```

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

```
1  package eu.spaziodati.metrics.streams
2
3   . . .
4
5  class thetaStream extends StartableStream {
6    override val config: thetaConfig = new thetaConfig
```

```
7   override val logger: Logger = LoggerFactory.getLogger(
     getClass)
8   override val name = "theta"
9
10  override def createStreamTopology(builder:
     StreamsBuilder): Topology = {
11    builder.stream(config.inputTopic, Consumed.`with`(
12        new JsonSerde[thetaDebeziumKey],
13        new JsonSerde[Debeziumtheta]
14    ))
15      .filter(
16        (_, record) =>
17          record != null &&
18            record.after.isDefined &&
19            record.after.get.user.isDefined &&
20            record.after.get.status.isDefined &&
21            // Filter out rows that are not finalised, i.
     e. will get updated
22            // status == "done", or "error" if the row is
      finalised.
23            List("done", "error").contains(record.after.
     get.status.get)
24      )
25      .map(
26        (key, record) => KeyValue.pair(
27          key.id, record.after.get
28        )
29      )
30      .groupBy(
31        (_, record: theta) => thetaDebeziumOutputKey(
32          record.user.get,
33          record.day.toLocalDate,
34        ),
35        Grouped.`with`(
36          new JsonSerde[thetaDebeziumOutputKey],
37          new JsonSerde[theta]
38        )
39      )
40      .aggregate(
41        () => new Outputtheta(-1, LocalDate.now(),0),
42        (aggKey: thetaDebeziumOutputKey, newValue: theta,
      aggValue: Outputtheta) => {
43          if (aggValue.user == -1) {
44            aggValue.user = aggKey.user
45            aggValue.day = aggKey.day
46          }
47          aggValue.n += 1
48          aggValue
49        },
50        Materialized.`with`(
```

```
51            new JsonSerde[thetaDebeziumOutputKey],
52            new JsonSerde[Outputtheta]
53          )
54        )
55        .toStream()
56        .mapValues(
57          record => new thetaDebeziumOutput("upsert",
    record),
58        )
59        .to(config.outputTopic, Produced.`with`(
60          new JsonSerde[thetaDebeziumOutputKey],
61          new JsonSerde[thetaDebeziumOutput]
62        ))
63      builder.build()
64    }
65 }
```

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

```
1 package eu.spaziodati.metrics.connectors
2
3  . . .
4
5 class PostgresSinkTask extends SinkTask {
6   private var connection: Connection = _
7   private val statementsCache: mutable.Map[(String,
      String), PreparedStatement] = mutable.Map[(String,
      String), PreparedStatement]()
8   private val logger = LoggerFactory.getLogger(getClass)
9   private val timeTravelTables = List("alpha", "beta", "
      mu")
10
11   override def start(propsJ: util.Map[String, String]):
      Unit = {
12     logger.info("Running with scala version: " + scala.
      util.Properties.scalaPropOrElse("version.number", "
      unknown"))
13     val props = propsJ.asScala
14     val properties = new Properties()
15     properties.setProperty("stringtype", "unspecified")
16     // Line below doesn't actually do anything, but
17     // prevents the Driver from not getting into the Uber
      -Jar
18     DriverManager.registerDriver(
19       new org.postgresql.Driver()
20     )
21     connection = DriverManager.getConnection(props(
      PostgresSinkConfigConstants.URL), properties)
22   }
23
24   override def stop(): Unit = {
```

```
25      connection.close()
26    }
27
28    private def makeQuestionMarks(n: Int): String = List.
       tabulate(n)(_ => '?').mkString(",")
29
30    private def bindParameters(stmt: PreparedStatement,
       values: Iterable[AnyRef]): PreparedStatement = {
31      var index = 1
32      values.foreach({
33        case value: ArrayList[String] =>
34          val valueArray = connection.createArrayOf("
       varchar", value.toArray())
35          stmt.setArray(index, valueArray)
36          index += 1
37        case value =>
38          stmt.setObject(index, value)
39          index += 1
40      })
41      stmt
42    }
43
44    private def doUpsert(table: String, record: Map[String,
        AnyRef], key: Map[String, AnyRef]): Unit = {
45      val stmt = statementsCache.getOrElseUpdate(
46        (table, "upsert"),
47        connection.prepareStatement(
48          s"""  INSERT INTO $table("${record.keys.mkString
       ("\",\"")}")
49                |  VALUES (${makeQuestionMarks(record.size)})
50                |  ON CONFLICT ("${key.keys.mkString("\",\"")
       }") DO UPDATE
51                |    SET "${record.keys.mkString("\"= ?,\"")}"
        = ?
52                |    WHERE $table."${key.keys.mkString("\"= ?
       AND " + table +".\"")}" = ?;
53                |""".stripMargin
54        )
55      )
56      bindParameters(stmt, record.values ++ record.values
       ++ key.values).execute()
57    }
58
59    private def doUpsertTimeTravel(table: String,
       recordWithNulls: Map[String, AnyRef], key: Map[String,
        AnyRef]): Unit = {
60      val record = recordWithNulls.filter {
61        case (_, v) => v != null
62      }
63      val stmt = statementsCache.getOrElseUpdate(
```

```
64        (table, s"tt:upsert(${record.keys.mkString(",")})")
    ,
65        connection.prepareStatement(
66          // N.B. With autocommit enabled (such is the
    default)
67          // these statements execute within a transaction.
68          s"""  UPDATE $table
69              |  SET validity = tstzrange(lower(validity),
    now())
70              |  WHERE ${key.keys.mkString("= ? AND ")} = ?
71              |    AND current_timestamp <@ validity
72              |    AND NOT EXISTS (
73              |      SELECT 1 FROM $table
74              |      WHERE ${record.keys.mkString("= ? AND
    ")} = ?
75              |        AND current_timestamp <@ validity
76              |    );
77              |
78              |  INSERT INTO $table (${record.keys.mkString
    (", ")}, validity)
79              |  SELECT ${makeQuestionMarks(record.size)},
    tstzrange(now(), 'infinity', '[)')
80              |  WHERE NOT EXISTS (
81              |    SELECT 1 FROM $table
82              |    WHERE ${record.keys.mkString("= ? AND ")}
     = ?
83              |      AND current_timestamp <@ validity
84              |  );
85              |""".stripMargin
86        )
87      )
88      bindParameters(stmt,
89        // UPDATE
90        key.values ++ record.values ++
91        // INSERT
92        record.values ++ record.values
93      ).execute()
94    }
95
96    private def doUpsertlambda(table: String, record: Map[
      String, AnyRef], key: Map[String, AnyRef]): Unit = {
97      val stmt = statementsCache.getOrElseUpdate(
98        (table, "email to id"),
99        connection.prepareStatement(
100         s"SELECT public.user_email_to_id(?) AS user_id;"
101       )
102     )
103     val rs = bindParameters(stmt, List(record("user_id"))
      ).executeQuery()
104     rs.next()
```

```
105     val userId = rs.getObject("user_id")
106     if (userId == null) {
107       logger.error(s"Got null while converting email '${
      record("user_id")}', in doUpsertlambda.")
108     }
109     else {
110       val mutableRecord = mutable.Map(record.toSeq: _*)
111       mutableRecord("user_id") = userId
112       val mutableKey = mutable.Map(key.toSeq: _*)
113       mutableKey("user_id") = userId
114       doUpsert(table, Map(mutableRecord.toSeq: _*), Map(
      mutableKey.toSeq: _*))
115     }
116   }
117
118   private def doDelete(table: String, key: Map[String,
      AnyRef]): Unit = {
119     val stmt = statementsCache.getOrElseUpdate(
120       (table, "delete"),
121       connection.prepareStatement(
122         s"""  DELETE FROM $table
123            |  WHERE ${key.keys.mkString("= ? AND ")} = ?;
      """.stripMargin
124       )
125     )
126     bindParameters(stmt, key.values).execute()
127   }
128
129   private def doDeleteTimeTravel(table: String, key: Map[
      String, AnyRef]): Unit = {
130     val stmt = statementsCache.getOrElseUpdate(
131       (table, "tt:delete"),
132       connection.prepareStatement(
133         s"""  UPDATE $table
134            |  SET validity = tstzrange(lower(validity),
      now())
135            |  WHERE ${key.keys.mkString("= ? AND ")} = ?
136            |    AND current_timestamp <@ validity;
137            |""".stripMargin
138       )
139     )
140     bindParameters(stmt, key.values).execute()
141   }
142
143   private def extractAfterRecord(data: mutable.Map[Object
      , Object]): Map[String, AnyRef] = {
144     data("after").asInstanceOf[util.Map[Object, Object]].
      asScala.map { case (k, v) =>
145       (k.toString, v)
146     }.toMap
```

```
147   }
148
149   private def extractTs(data: mutable.Map[Object, Object
       ]): Long = {
150     data("source").asInstanceOf[util.Map[Object, Object
       ]].asScala.map {
151       case (k, v) => (k.toString, v)
152     }.map {
153       case ("ts_ms", ts) => Some(ts.asInstanceOf[Long])
154       case _ => None
155     }.filter(x => x.isDefined).head.get
156   }
157
158   override def put(recordsJ: util.Collection[SinkRecord])
       : Unit = {
159     val records = recordsJ.asScala
160     var table = "unknown"
161     for (record <- records) {
162       table = record.topic()
163         .split("\\.").takeRight(2).mkString(".")  //
       Avoid the Debezium's assigned database name
164       val key = record.key().asInstanceOf[util.Map[String
       , AnyRef]].asScala.toMap
165       val json = record.value().asInstanceOf[util.Map[
       Object, Object]].asScala
166       if (json != null) {  // Ignore Debezium tombstone
       events
167         var data = Map[String, AnyRef]()
168         try {
169           data = extractAfterRecord(json)
170         }
171         catch {
172           case e: NullPointerException => // there is no
       `after` field; discard.
173         }
174         json("op") match {
175           case "c" | "r" =>
176             if (timeTravelTables.contains(table)) {
177               doUpsertTimeTravel(table, data, key)
178             }
179             else {
180               doUpsert(table, data, key)
181             }
182           case "u" =>
183             if (timeTravelTables.contains(table)) {
184               doUpsertTimeTravel(table, data, key)
185             }
186             else {
187               doUpdate(table, data, key)
188             }
```

```
189           case "d" =>
190             if (timeTravelTables.contains(table)) {
191               doDeleteTimeTravel(table, key)
192             }
193             else {
194               doDelete(table, key)
195             }
196           case "upsert" => doUpsert(table, data, key)
197           case "lambda" => doUpsertlambda(table, data,
      key)
198         }
199       }
200     }
201     if (records.nonEmpty) {
202       logger.info(s"Processed ${records.size} records for
      table $table.")
203     }
204   }
205
206   override def version(): String = "1.0.0"
207 }
```