UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and
Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

Titolo

Sottotitolo (alcune volte lungo - opzionale)

Supervisor
Prof. Gabriel Mark Kuper

Student
Daniele Parmeggiani

Academic year 2020/2021

Dedication

Contents

Abstract

In Appendix A the listings of relevant source code are provided.

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, the accuracy and freshness of the data are critical success factors: 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 internship have been discouraged, and should have only been performed in case of necessity. Such was not the case for the work presented here. See [3] for further details.

[2] Description via `https://datarade.ai/data-providers/spaziodati/profile`, last visited January 12, 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 said usage of the product, employing activities 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 [1].

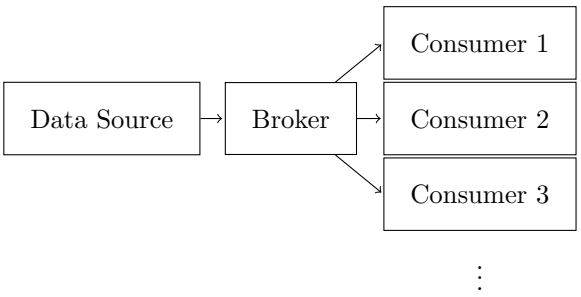

Figure 1.1: A Generic CDC System.

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.

The broker is intended as a delivery mechanism for messages, by which we mean content (i.e. in our case, change data) interchanged between systems. 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.

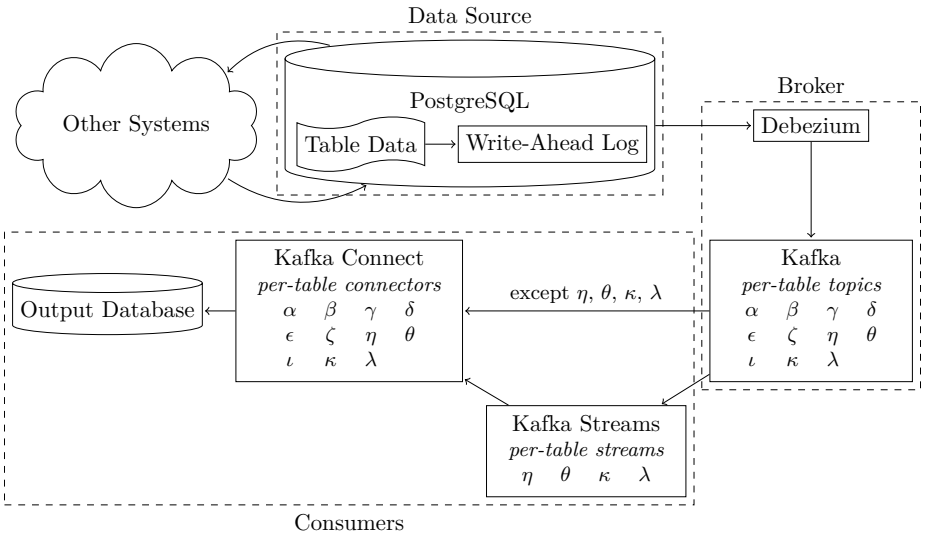

Figure 1.2: An Outline of the CDC System presented in this Theses.

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 [2, 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 [2, 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 $M\Big(\sigma^*_{\sum n}(G_m(\omega))\Big)$, 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. □

[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

2.1 Apache Kafka

2.1.1 Kafka Connect

2.1.2 Kafka Streams

duality of streams and tables

2.2 PostgreSQL

2.2.1 Write Ahead Log

2.3 Debezium

2.4 Scala

Chapter 3

Remarks on the Data

We will assume the following, without proof:

Assumption 1. $\text{source}\,[*]$ 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, $\text{dest}\,[\alpha] := \pi_S\,(\text{source}\,[\alpha])$ over some subset S of its columns, with one exception for table γ:

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

3.1 Aggregations

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

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] 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).

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

[3] 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

Listing A.1: source/python-example.py

```
1 import numpy as np
2
3 def incmatrix(genl1,genl2):
4     m = len(genl1)
5     n = len(genl2)
6     M = None #to become the incidence matrix
7     VT = np.zeros((n*m,1), int)  #dummy variable
8
9     # compute the bitwise xor matrix
10    M1 = bitxormatrix(genl1)
11    M2 = np.triu(bitxormatrix(genl2),1)
12
13    for i in range(m-1):
14        for j in range(i+1, m):
15            [r,c] = np.where(M2 == M1[i,j])
16            for k in range(len(r)):
17                VT[(i)*n + r[k]] = 1;
18                VT[(i)*n + c[k]] = 1;
19                VT[(j)*n + r[k]] = 1;
20                VT[(j)*n + c[k]] = 1;
21
22                if M is None:
23                    M = np.copy(VT)
24                else:
25                    M = np.concatenate((M, VT), 1)
26
27                VT = np.zeros((n*m,1), int)
28
29    return M
```

Listing A.2: source/dir/scala-example.scala

```
1 package examples
2
```

```
3  object Persons {
4
5    /** A list of persons. To create a list, we use Predef.
       List
6     *  which takes a variable number of arguments and
       constructs
7     *  a list out of them.
8     */
9    val persons = List(
10     new Person("Bob", 17),
11     new Person("John", 40),
12     new Person("Richard", 68)
13   )
14
15   /** A Person class. 'val' constructor parameters become
16    *  public members of the class.
17    */
18   class Person(val name: String, val age: Int)
19
20   /** Return an iterator over persons that are older than
       20.
21    */
22   def olderThan20(xs: Seq[Person]): Iterator[String] =
23     olderThan20(xs.elements)
24
25   /** Return an iterator over persons older than 20,
       given
26    *  an iterator over persons.
27    */
28   def olderThan20(xs: Iterator[Person]): Iterator[String]
       = {
29
30     // The first expression is called a 'generator' and
       makes
31     // 'p' take values from 'xs'. The second expression
       is
32     // called a 'filter' and it is a boolean expression
       which
33     // selects only persons older than 20. There can be
       more than
34     // one generator and filter. The 'yield' expression
       is evaluated
35     // for each 'p' which satisfies the filters and used
       to assemble
36     // the resulting iterator
37     for (p <- xs if p.age > 20) yield p.name
38   }
39 }
40
41
```

```
42 /** Some functions over lists of numbers which
       demonstrate
43  *  the use of for comprehensions.
44  */
45 object Numeric {
46
47   /** Return the divisors of n. */
48   def divisors(n: Int): List[Int] =
49     for (i <- List.range(1, n+1) if n % i == 0) yield i
50
51   /** Is 'n' a prime number? */
52   def isPrime(n: Int) = divisors(n).length == 2
53
54   /** Return pairs of numbers whose sum is prime. */
55   def findNums(n: Int): Iterable[(Int, Int)] = {
56
57     // a for comprehension using two generators
58     for (i <- 1 until n;
59          j <- 1 until (i-1);
60          if isPrime(i + j)) yield (i, j)
61   }
62
63   /** Return the sum of the elements of 'xs'. */
64   def sum(xs: List[Double]): Double =
65     xs.foldLeft(0.0) { (x, y) => x + y }
66
67   /** Return the sum of pairwise product of the two lists
       . */
68   def scalProd(xs: List[Double], ys: List[Double]) =
69     sum(for((x, y) <- xs zip ys) yield x * y);
70
71   /** Remove duplicate elements in 'xs'. */
72   def removeDuplicates[A](xs: List[A]): List[A] =
73     if (xs.isEmpty)
74       xs
75     else
76       xs.head :: removeDuplicates(for (x <- xs.tail if x
       != xs.head) yield x)
77 }
78
79
80 /** The main class, the entry point of this program.
81  */
82 object Fors {
83
84   def main(args: Array[String]) {
85     // import all members of object 'persons' in the
       current scope
86     import Persons._
87
```

```
88      print("Persons over 20:")
89      olderThan20(persons) foreach { x => print(" " + x) }
90      println
91
92      import Numeric._
93
94      println("divisors(34) = " + divisors(34))
95
96      print("findNums(15) =")
97      findNums(15) foreach { x => print(" " + x) }
98      println
99
100     val xs = List(3.5, 5.0, 4.5)
101     println("average(" + xs + ") = " + sum(xs) / xs.
        length)
102
103     val ys = List(2.0, 1.0, 3.0)
104     println("scalProd(" + xs + ", " + ys +") = " +
        scalProd(xs, ys))
105   }
106
107 }
```