Hasso Plattner Institut
IT Systems Engineering
Universität Potsdam

**Bachelorarbeit**

# User-aided Pattern Search and Analysis on Business Graphs

## Nutzergestuetzte Graphanalyse und Mustersuche auf Unternehmensgraphen

Milan Gruner
milangruner@gmail.com

Eingereicht am <TBD>

Fachgebiet Informationssysteme
Betreuung: Prof. Dr. Felix Naumann, Michael Loster, Toni Gruetze

## Abstract

Costructing a graph made up of thousands of businesses may be hard, but actually making sense of it is a lot harder. With huge amounts of data potentially being integrated into the data lake every day, automatic methods for finding interesting spots in the graph are needed. This paper discusses different approaches that can be taken to extract useful knowledge from such a graph.

# Contents

# 1 Introduction

This paper describes the basic approach of Pattern Search and Analysis on the graphs generated as a part of the Ingestion bachelor project at HPI Potsdam. It deals with the experiments I performed to evaluate certain algorithms and approaches to summarize automatically generated graphs in the ball park of tens of millions of nodes in a large and mostly unconnected and sparse graph. It contains businesses, places and persons and can be augmented freely by adding additional data sources (or manual data input) of your own. The data is visualized and controllable from the Curation interface, which was a project developed alongside the Ingestion pipeline. Curation generates graphs on-the-fly that contain the statistics data that was written as a side effect of the numerous Spark jobs in the Ingestion pipeline and gives a lot of insight into what's going on in the "data lake". All of the experiments contained in this paper were conducted using either the Curation interface, the Apache Zeppelin interactive web shell or the Spark shell.

Most of the topics in this paper deal with graph-related topics, so the Spark GraphX framework (in Scala) will mostly be used to describe the algorithms that were employed, but it can be freely seen as functional pseudo code and adapted to other popular graph processing frameworks (such as Giraph etc.) For UI-related or user-oriented algorithms, JavaScript (or JSX, so partly using React/ Redux etc.) is the listing language, as visual problems are more easily expressed in this tree-oriented but still functional style (and also because it's used in Curation). Whereever possible, ES2016 Syntax is used for the brevity and the similarity to the functional approach of Scala and the rest of the example code.

Also one of the purposes of the Ingestion and Curation projects was to enable the execution of modern text mining, deduplication and parallel data analysis techniques to the german business landscape. This means that some of the examples will be in German (but they will be translated if needed for understanding the technique).

## 1.1 Usage Scenarios of Ingestion, Curation and [this paper]

The process of classic risk analysis which is still majorly employed in today's banks is mostly a model-oriented process that heavily relies on speculation or predictions.
The techniques that Ingestion, Curation and this paper (TODO: project name?) offer are universally usable to analyze huge amounts of structured or unstructured data in the form of WikiData, JSON, CSV or similar data as well as newspaper articles, messages (eMail, instant or otherwise) or similar data sources.
The results of this can either be viewed as a graph or a relational model using the conversion algorithms briefly discussed later in this paper. Because [this system] has automatic as well as user-driven processes that control the flow, categorization and importance

grading of all the data in the data lake, it may be effectively deployed in a bank or similar institution as a data collection, curation and analysis software stack.
It is meant to be heavily extensible and allows the simple integration of new data sources as well as algorithms and data structures in general. The pipeline itself is very modular and is composed of many stand-alone Spark jobs that mostly read and write from or to the Cassandra database. For more information on this technique, see [Nils' paper on Cassandra].

### 1.2 Pattern Analysis vs. PageRank

TODO

## 2 Data structures for business entities

### 2.1 The *subject* data structure

The central data structure that exists inside the datalake is called *subject*. It contains all the final businesses, industry sectors, countries or persons that are the result of the computations in the Ingestion project.
This is what it looks like in Scala:

```
case class Subject(
  var id: UUID = UUID.randomUUID(),
  var master: UUID,
  var datasource: String,
  var name: Option[String] = None,
  var aliases: List[String] = Nil,
  var category: Option[String] = None,
  var properties: Map[String, List[String]] = Map(),
  var relations: Map[UUID, Map[String, String]] = Map(),

  var master_history: List[Version] = Nil,
  var name_history: List[Version] = Nil,
  var aliases_history: List[Version] = Nil,
  var category_history: List[Version] = Nil,
  var properties_history: Map[String, List[Version]] = Map(),
  var relations_history: Map[UUID, Map[String, List[Version]]] = Map()
)
```

### 2.2 A versioning scheme that stands the test of time

Versioning is needed for the automatic system that includes various pipelines that write to lots of different tables in Cassandra during their runtime, but at the end it is expected that all data will flow back into the central 'data lake' that is the *subject* table, described earlier in this chapter. This means that fully automated processes may and will write millions of data entries to the datalake at an hourly basis (or even more frequent), so if any breaking changes are introduced by these, they need to be rerolled quickly. This means that each version isn't saved as a diff like in version control systems (e.g. git), but rather every single version has a copy of all the attributes, a timestamp and the name and version of the program that modified it. As our whole pipeline is made up of individual small Spark jobs that write mostly into their own Cassandra tables, the error in the data can be quickly located and traced to the corresponding source code.

Because our data domain included events, attributes and relations that had temporal validity parameters attached to them, we had to realize a versioning scheme that was compatible with the column-family oriented storage of Cassandra. As everything is distributed across the cluster here, a diff-oriented versioning would be hard to keep up, so our team decided on a deeply structured deep copy backup system for our versions.

The version data structures are stores for each attribute and each individual key of the subject entries, which makes restoring individual columns or single data entries efficient and easily parallelizable.

## 3 Using Apache Spark and GraphX for Graph Analysis

### 3.1 Motivation: Why Spark, Cassandra and GraphX?

### 3.2 The Spark GraphX vertex and edge data structures

### 3.3 Transforming *Subjects* into GraphX compatible relation data

### 3.4 Writing efficient Spark GraphX code

# 4 Graph Summarization

## 4.1 What users actually want to see

## 4.2 Compressing graph information to the bare minimum

## 4.3 Presenting graph data appealingly

## 4.4 Related work

# 5 Pattern Search

## 5.1 Discerning patterns from randomness

## 5.2 Pattern types and their applications

## 5.3 Related work

# 6 Pattern Analysis

## 6.1 User-aided approaches for Pattern Categorization

## 6.2 Pattern importance measures

## 6.3 Machine Learning Models for analyzing user feedback

## 6.4 Related work

# 7 Benchmarks and Experiments

## 7.1 GraphX vs. Plain RDDs

## 7.2 Design decisions and trade-offs

## 7.3 Technical challenges

UUID reduction to store in Longs

# 8 Conclusion and outlook

## References