Hasso Plattner Institut
IT Systems Engineering
Universität Potsdam

Bachelorarbeit

GraphXplore: 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. This graph 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 GraphXplore

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 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 project 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:

Listing 1: Subject

```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()
)
```

The first three attributes (*id*, *master*, *datasource*) constitute the primary key in the Cassandra databases. The partitioning key that is used to decide on which Cassandra instance to store a given *Subject* entry is the *master* ID, which makes sure that a master node and all its data source slave nodes are stored on the same machine. *id* and *datasource* are used as clustering keys, which control the sorting inside the data partitions produced by database queries. They are also easily queryable when specific IDs or data source entries for a given master ID are needed.

When loading Cassandra data into a Spark job, simple fields (e.g. Strings) need to be wrapped in an Option in case that the field contains no value (otherwise there would be an error), which isn't necessary for the fields containing Maps or Lists, which would just contain an empty data structure in that case.

The *properties* field contains all the extracted data from the various data source extraction and normalization steps in the Ingestion pipeline. It is structured in a way to make storing arbitrary data possible, by allowing to store a List of Strings for each property key. This way, a list of eMail addresses can be stored as well as a single postal code and they can be treated the same way.

The *relations* field is structured in a two-dimensional map. The first level is addressed by the target *subject* ID, while the second level's keys are the specific relation type. The value that is returned when the field is queried by the target ID and relation type is the relation attribute, which is mostly used to store a confidence measure (a value from 0 to 1) or the count of the relation type between the current *subject* and the target.

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.

The version data structure that was used is structured like this:

Listing 2: Version

```
case class Version(
  version: UUID = UUIDs.timeBased(),
  var program: String,
  var value: List[String] = Nil,
  var validity: Map[String, String] = Map(),
  var datasources: List[String] = Nil,
  var timestamp: Date = new Date()
)
```

It is stored inside the collections of the *history* fields of the *Subject* data structure and contains useful meta information for restoring old versions, modeling temporal validity (attributes only being valid for a certain timespan) or filtering out data from certain data sources. They also contain the Spark program's name that created the relevant changes and the time they were made.

Additionally, the *version* table contains a list of all changes to the datalake, making it traceable which programs were ran at what time, which table they wrote to and what data sources they processed. Using this, the Curation interface can display the annotated version list, and run Spark jobs that restore the whole data lake to a previous version or compute the changes that a version made in the *subject* table.

3 Using Apache Spark and GraphX for Graph Analysis

This chapter will briefly discuss the reasons for using Spark's GraphX API alongside the Apache Cassandra database. Then the transformations that are necessary to extract GraphX-compatible data from the data lake will be detailed. The following chapters will build on the techniques described in this one, so that only the relevant graph processing sections need to be shown later on.

3.1 Why Spark, Cassandra and GraphX?

The subject table contains millions of entries. In order to efficiently extract a graph and calculate the results of complex graph algorithms, a parallel processing framework like Apache Spark is a really important measure to take into consideration when runtimes of multiple hours (or even days) aren't desired. Cassandra is a database that is up to the challenge of delivering the input for and storing the results of these computations without much hassle (like serializing to CSV or Parquet files would involve). It is also well integrated into the Spark API and can take care of complex nested data structures like e.g. the properties and relations attributes of the *Subject* case class. GraphX is used simply because it enables the description of the Graph algorithms discussed in this paper in an abstract and terse fashion. It also acts as a way to operate on compress graph information without always compressing and restoring the relevant data in every stage of the respective Spark job. Using the Pregel operator contained in GraphX, iterative graph algorithms can be broken down into parallelizable messages that are sent along the edges of the graph.

TODO continue

3.2 Transforming *Subjects* into vertex and relation data

For GraphX, node IDs always have to be ints or longs in order to tightly pack the node and relation data for efficient parallel processing. This however imposes some limitations on the data that is to be imported into the system. As all of our vertices are instances of the *Subject* case class, they are referenced by their respective UUIDs, which contain 128 Bit of information. In order to store this in a long, either the 64 most significant bits (MSB) have to be taken, which might lead to collisions when two UUIDs collide in their first halves, or the hashCode of the UUIDs has to be used to take them into full consideration.

The latter approach also works for strings and pretty much any data type that can be expressed in Scala, so it is the one which was selected for the transformation.

Because of the irreversible nature of both methods of ID compression, a mapping from new ID to the old UUID needs to be created and joined with the calculated data in each step where the old UUIDs are needed (e.g. when querying additional information about the subjets or when exporting the graph data).
Because GraphX lets the vertex and edge attributes have any type and structure, it makes sense to extract the relevant subject data when querying it in the beginning of a Spark job, to keep this joining down to a minimum.

Another step that needs to be executed is the extraction of the relevant relations from the corresponding attribute in the *Subject* data structure. This involves flattening a two-dimensional map (a 'map of maps') into many tuples of the two map keys, relation target UUID and type, and the relation attribute. This then needs to be augmented by the source UUID in order to make it useful for GraphX. All UUIDs need to be converted into longs along the way so that the edges can still be interpreted when they are processed on the worker nodes.

The *transformToGraph* function from *TODO* takes care of this process: *TODO* Listing

3.3 The Pregel operator in GraphX and its uses

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

8 Conclusion and outlook

References