2021-04-05_IntroductiontodistributedGraphAnalyticsSystems_PDF_compilable

April 5, 2021

1 Introduction to distributed Graph Analytics Systems: GraphX and GraphFrames for Apache Spark

1.1 Abstract

The following study project introduces the main characteristics of graph database systems and graph analytics systems and justifies their applications in practical use cases. Many datasets can most naturally be described via graph data structures and therefore the need for their efficient and effective processing and storage arises regularly. Starting from basic theoretical concepts concerning graph datasets and algorithms, we lay out the key differences between graph database systems (namely Neo4j) and graph analytics systems (namely Apache GraphX, Apache Giraph and GraphFrames for Apache Spark). The latter will be in to focus of this paper. Based on these conceptual foundations we provide example use cases for both GraphX and GraphFrames, showcasing their individual strengths and differences, implemented in both Scala and Python3. Based on existing scientific literature we provide a comparison in terms of performance, scalability and fault tolerance of the systems listed above. We conclude by providing a use-case dependent recommendation of graph analytics systems grounded on both the executed use-cases from above and the literature analysis.

1.2 Environment description

The environement is based on jupyter/all-spark-notebook, Docker image from https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html

The image includes Python, R, and Scala support for Apache Spark. The environement contains: * Installed latest Apache Spark with Hadoop binries * Python support * IRKernel for R support * Apache Toree and spylon-kernel to support Scala code in Jupyter notebooks (YAY!) * ggplot2, sparklyr, and rcurl packages

To the base image we added: * GraphFrames installation with pip * Clone of this git repository

In order to keep the deployment as lightweight as possible and focus on the concepts of the analyzed systems rather than their deployment at scale, we opted for a Apache Spark deployment in local mode. Therefore all uses cases will not be processed in a distributed fashion but rather locally on one machine. If you have a Spark cluster available to you and would like to run our code on that cluster, you can easily do so by making sure that versions align and pointing the Spark Master to the respective Master IP in your cluster.

1.3 Table of Contents

1.4 Introduction of Graph Databases and Graph Analytics Systems

*Sources: * https://engineering.fb.com/2015/06/02/core-data/recommending-items-to-more-than-a-billion-people/ * https://arxiv.org/abs/2103.03227 * https://arxiv.org/pdf/2102.13613.pdf * https://www.scitepress.org/papers/2018/69102/69102.pdf * _https://medium.com/@gauravsarma1992/neo4j-storage-internals-be8d150028db_ * https://www.slideshare.net/thobe/an-overview-of-neo4j-internals * https://arxiv.org/pdf/2012.06171.pdf*

Many systems of academic or economic interest give rise to datasets that can most naturally be described by graphs. Examples include the interaction between agents in a social network, content recommendation , financial crime and fraud detection as well as transaction processing. In order to efficiently store graph data, specific database designs are required that augment existing NoSQL design paradigms to better fit graph datasets.

For the purposes of this paper, a graph database system shall be defined as follows: **Graph database systems** are a subgroup of schemaless NoSQL database systems. Contrary to relational database systems, where the single most basic unit of data information is a record in a predefined relation, graph database systems distinguish two basic units of information: 1. Nodes, representing graph vertices (e.g. teachers, students, etc.) 2. Relationships, representing graph edges that represent the relationship between nodes (e.g. teaches, hears, tutors, etc.)

Both nodes and relationships allow for the classical notion of attributes in relational databases, though without a fixed schema. The aforementioned types of nodes and relationships are called labels and each node or relationships can contain one label value (e.g. a node is labeled as a teacher or student). By inheritance of the characteristics of schemaless databases as well as their augmentation through the additional structures outlined above, the following core characteristics of graph databases shall be noted.

- **Flexible schema** - While relational databases require new tables or alterations in the existing ones to add new types of data, in a graph database we can add new type of vertices and edges without alterations in the previously stored data;

- **Query language** – Relational databases use SQL query language to manipulate the database, but graph databases needed a more powerful query language. There are many different query languages to manipulate specific graph databases like Cypher on Neo4J or AQL on ArangoDB, and we will study to see which one suits best the needs of querying data;

- **Sharding** - One of the key elements to being able to scale NOSQL databases is sharding, where a large dataset can be broken up and distributed across a number of (typically

replicated) shards;

- **Backups** – Graph databases should provide functions for planning, performing and restoring a database backup. A full backup contains all data files and information required to restore a repository to the state it was in at the time of the backup;

- **Multi-model** – A multi-model graph database provides a database with unstructured data, and we can visualize relationships with data like in the form of graphs, key-value pairs, documents or tables;
- **Multi-architecture** - When planning a graph database solution, there are several structural decisions to make. These decisions may vary slightly, depending on which database product you choose.
- **Scalability** - As hardware continues to innovate at a rapid pace (mass storage devices, CPU and networks), this in turn leads to increased capacity for existing database software, enabling growth in the number of transactions per second. There are two approaches to scaling a database: Vertical (Scale Up) and Horizontal (Scale Out). Vertical scaling involves adding more physical or virtual resources to the underlying server hosting the database – more CPU, more memory or more storage. Horizontal scaling involves adding more instances/nodes of the database to deal with increased workload;
- **Cloud ready** – A graph database implemented on cloud is a great feature because it solves scalability problems and provides real-time management.

In the following paragraph some of the technical details of graph databases (in this case Neo4j serves as a representative implementation) will be outlined. As will be detailed in the following sections, a graph generally consists of a set of nodes and edges, both of which can contain a set of attributes. This set of attributes can differ for each node/edge. Each node is stored as an individual node record. Node attributes are stored as a singly-linked lists consisting of key-value pairs, with the first (ordering is arbitrary and subject to insertion order unless explicitly specified otherwise) list item being pointed to by the node record itself. Node records also contain a pointer to the first (again, ordering is arbitrary unless specified otherwise) edge the node is referenced by. Similarly, edges, in the language of Neo4j referred to as relationships, are stored as individual records. Just like nodes, the attributes of a relationship are stored as singly-linked lists. A relationship record contains not only pointers to its start- and end-node, but also to the next/previous relationship of its end-/start-node respectively (again, ordering is arbitrary). This data structure allows for "index free adjacency", meaning there is no need for looking up indices to find relationships between nodes. This marks one major advantage over representing graphs in relational database systems, as such joins tend to become computationally expensive on large datasets. The data types described above are persisted in separate stores on disk, namely Node Stores, Relationship Stores and Property Stores. In the case of Neo4j, large attribute values (e.g. BLOBs) are stored in yet again separate files and merely a pointer to the respective file is stored directly as the value in the singly-linked list. Other peripheral and implementation specific software modules, such as caching, high availability functionality etc. shall not be detailed in the course of this paper.

Graph Analytics Systems operate on top of a (graph-) database and allow for efficient analysis of the underlying data. Most of these systems provide interfaces to common (graph-) databases for frictionless data integration and preprocessing as well as frequently used graph algorithms. Major implementations include Apache Giraph, GraphLab, Blogel and Spark GraphX. The latter will be the focus of this paper, as well as it's successor, GraphFrames.

1.5 Theory and Analysis of Graph Datasets

1.5.1 Introduction to Graph Theory

Depending on the inherent structure of an individual dataset different types of graphs are required to encode its relevant attributes. Some of the most prevalent graph structures include:

Undirected graphs An undirected graph is defined as an ordered pair $ G = (V, E) $, with: * $ V $, a set of vertices * $ E $, a set of edges, where each edges is an unordered pair of vertices

Undirected graphs are commonly used to represent symmetric associations between entities, e.g. the connectedness of cities given a specific road network. In this example each node represents a city and each edge represents a road between two cities. In the example above it is assumed that each road can be traversed in both directions. In order to represent the more general case of including one-way roads in the road network, one needs to augment undirected graphs with directionally encoding.

Directed graph A directed graph is defined as an ordered pair $ G = (V, E) $, with: * $ V $, a set of vertices * $ E $, a set of edges, where each edges is an ordered pair of vertices

Coming back to the previous example, a directed graph allows for the definition of unidirectional roads between cities, e.g. due to road width restrictions. Given the ability to represent all road networks as a directed graph, one might want to perform analyses on upcoming trips. For example, one might be interested in finding the shortest path connecting two cities. To do so, information about distances between cities needs to be incorporated in the graph data structure.

Completing the aforementioned example, Phi associates a distance measure to each edge, representing the length of each connecting road.

Finally, an algorithm for finding the shortest paths is required to make use of the data structures defined above. The following table presents a selection of these algorithms.

1.5.2 PageRank Algorithm

*Sources: * https://blogs.cornell.edu/info2040/2014/11/03/more-than-just-a-web-search-algorithm-googles-pagerank-in-non-internet-contexts/ * https://towardsdatascience.com/pagerank-algorithm-fully-explained-dc794184b4af*

In the following section the PageRank algorithm will be introduced. This algorithm is most commonly applied in order to identify "important" nodes in graph networks based on graph topology. Practical use-cases of include: * Ranking websites by the amount of references to them weighted by reference quality * Ranking accounts in a transaction network by their transaction activity weighted by transacted value * Predicting traffic flow in road networks

To facilitate the weighting of nodes, the PageRank algorithm calculates the stationary state probability distribution of a Markov-Process defined by the given graph network. As a simple example, the following graph shall be regarded. This example assumes the reader to be familiar with the general concepts of Markov-Processes.

The transition matrix of the Markov-Process is set to $ 1/n $ for each outgoing edge of a node, where n is defined as the total count of all outgoing edges of the respective node . Self references are ignored and in a context of weighted edges the transition probabilities are weighted according to the total outgoing edge weights instead. Therefore, all column sums equal one and describe the

probability of moving from a specific column-state to the respective row-state. The graph displayed above is represented by the transition matrix $ P $.

This implies that must be an eigenvector of $ P $ with the corresponding eigenvalue of $ 1 $. To compute, one can make of the power method algorithm, which in turn makes use of the *Frobenius-Perron* theorem. This theorem states that given a non-negative square matrix, there exists only one positive eigenvalue whose eigenvector can be chosen to consist of only positive entries, qualifying this eigenvector as the probability distribution of interest. This eigenvalue also is dominant, meaning it is the eigenvalue of $ P $ with the highest absolute value. Both implications outlined above allow one to use the Power method outlined below to find the eigenvector of interest, where vector b is initialized randomly and $ A=P $.

1.6 Introduction of selected Graph Database and Graph Processing Systems

1.6.1 Apache Spark

*Sources: * Salloum et. al., 2016, "Big data analytics on Apache Spark"*

Apache Spark (short: Spark) is an In-Memory distributed computing framework that is implemented in Scala and serves similar purposes as Apache Hadoop. Though, unlike Apache Hadoop, Spark is performing processing in RAM, making it orders of magnitude faster than its On-Disk counterpart. Spark (in the narrow sense) consists of Spark Core and Upper-Level Libraries. Spark Core handles communication to a Cluster Manager in the context of data ingestion from supported databases. It also provides the Resilient Distributed Dataset (RDD) API, which is consumed by Upper-Level Libraries. Upper-Level Libraries provide functionalities built upon the RDD abstraction that allow users to e.g. instruct Spark to load a dataset and perform analysis on the dataset.

An RDD is an immutable data structure that Spark uses to store (intermediate) results of processing pipelines. As the name would suggest, an RDD is partitioned across Spark Worker nodes and implements automatic fault recovery. Users can explicitly decide whether to store an RDD in memory or on disk of the respective worker nodes. Another data structure in Spark is the DataFrame, which is built on top of the RDD data structure. As the name suggests, a DataFrame is defined in a similar fashion as e.g. Pandas DataFrames, meaning it consists of tabular data in labeled columns with predefined types. This additional information contained in the data allows Spark to further optimize data partitionining and therefore claims to boost performance.

1.6.2 Apache Spark GraphX

*Sources: * https://amplab.cs.berkeley.edu/wp-content/uploads/2014/09/graphx.pdf * https://spark.apache.org/docs/2.1.0/graphx-programming-guide.html#pregel-api*

In general GraphX is a component in Apache Spark technology eco-system for graphs and graph-parallel computation. GraphX reuses Spark RDD concept and provides API for fast and robust development of graphs logic. Therefore, this integrated component simplifies graph analytics tasks and provides the ability to make operations on a directed multi-graphs. GraphX API implements a variant of the Pregel paradigm as well as a range of common graph operations. GraphX API enables the composition of graphs with unstructured and tabular data and permits the same physical data to be viewed both as a graph and as collections without data movement or duplication. Before we describe GraphX more detailed, it make sense to say a couple of words about Pregel. Pregel is a data flow paradigm and system for large-scale graph processing created at Google to solve problems that are hard or expensive to solve using only the MapReduce (e.g. Apache Hadoop) framework.

Pregel is essentially a message-passing interface constrained to the edges of a graph. The idea is to "think like a vertex" — algorithms within the Pregel framework are algorithms in which the computation of state for a given node depends only on the states of its neighbors.

GraphX Programming Abstraction *Sources: * GraphX: Graph Processing in a Distributed Dataflow Framework (https://amplab.cs.berkeley.edu/wp-content/uploads/2014/09/graphx.pdf)*

Property graph as a collection - The vertex collection contains the vertex properties uniquely keyed by the vertex identifier. The edge collection contains the edge properties keyed by the source and destination vertex identifiers. By reducing the property graph to a pair of collections it is possible to compose graphs with other collections in a distributed dataflow framework like Apache Spark. *Graph Computation in distributed dataflow framework* - The normalized representation of a property graph as a pair of vertex and edge property collections allows embedding graphs in Apache Spark framework. A key stage in graph computation is constructing and maintaining the so-called triplets view, which consists of a three-way join between the source and destination vertex properties and the edge properties. Graph-parallel computation in such a framework as Apache Spark is expressed as a sequence of join stages and group-by stages punctuated by map operations. Consequently, these stages are the focus of performance optimization in graph processing systems. *GraphX Operators* - The GraphX programming abstraction extends the Spark dataflow operators by introducing a small set of specialized graph operators e.g. Graph, vertices, edges, triplets, mrTriplets, in case of Pregel API - messages. This operators will be described more detailed in the following sections.

GraphX built on Apache Spark GraphX achieves performance parity with specialized graph processing systems by recasting the graph-specific optimizations on top of a small set of standard dataflow operators in Apache Spark. *Distributed Graph Representation.* As already mentioned GraphX represents graphs internally as a pair of vertex and edge collections built on the Spark RDD abstarction. These collections introduce indexing and graph-specific partitioning as a layer on top of RDDs. The vertex collection is hash-partitioned by the vertex IDs. Whereas, the edges are divided into three edge partitions by applying a partition function (e.g., 2D Partitioning). Co-partitioned with the vertices, GraphX maintains a routing table encoding the edge partitions for each vertex. GraphX inherits the immutability of Spark RDDs and therefore all graph operators logically create new collections rather than destructively modifying existing ones. As a result, derived vertex and edge collections can often share indices to reduce memory overhead and accelerate local graph operations. *Implementation of the Triplets View.* Because the vertex and edge property collections are partitioned independently, the join requires data movement. GraphX performs the three-way join by shipping the vertex properties across the network to the edges, thus setting the edge partitions as the join sites. Furthermore, GraphX introduces a multicast join in which each vertex property is sent only to the edge partitions that contain adjacent edges. The routing table which is associated with the edge collection and constructed lazily upon first instantiation of the triplets view is used for multicast join. As iterative graph algorithms often modify only a subset of the vertex properties in each iteration. Therefore, incremental view maintenance is applied to the triplets view to avoid unnecessary movement of unchanged data. After each graph operation, we track which vertex properties have changed since the triplets view was last constructed. When the triplets view is next accessed, only the changed vertices are re-routed to their edge-partition join sites and the local mirrored values of the unchanged vertices are reused. This functionality is managed automatically by the graph operators. GraphX incorporates two additional query optimizations for the mrTriplets operator: filtered index scanning and automatic join elimination as well as additional optimizations for memory-based shuffle, batching and columnar structure and

variable integer encoding, which we will not describe in details in this study project.

GraphX in Apache Spark 3.1.1 *Sources: * https://spark.apache.org/docs/latest/graphx-programming-guide.html#:~:text=GraphX%20is%20a%20new%20component,to%20each%20vertex%20and%20edge
* Kaggle - online community of data scientists and machine learning practitioners*

In the following based on an example we will demonstrate how to build a property graph, how to use some common Graph Operators and Property Operators, how to compute degree information as well as an simple application of PageRank algorithm using GraphX in Apache Spark 3.1.1.
For this example two data sets were taken from Kaggle. The datasets contain information about several bike stations in USA and trips which were done between this station in 2013-2014. Sofar Apache Spark provides only Scala API for GraphX. Therefore the example is written is Scala and should be executed in spylon-kernel. If the current notebook kernel is not spylon-kernel switch it in the upper right coner of the grafical interface.

———- DISCLAIMER ———-

This code is written in Scala, so please change the notebook kernel accordingly :)

[1]:
```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.functions.col
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.DataFrame

//val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
//spark context is already availiable in the running environment
//read in station dataset from csv file and register it as a table
val station = spark.read.format("csv")
         .option("header", "true") //first line in file has header
         .option("mode", "DROPMALFORMED")
         .load("./02_Datasets/station.csv")
station.registerTempTable("station")
val bikeStations = spark.sqlContext.sql("SELECT * FROM station")
bikeStations.printSchema()

//read in trip dataset from csv file and register it as a table
val trip = spark.read.format("csv")
         .option("header", "true") //first line in file has header
         .option("mode", "DROPMALFORMED")
         .load("./02_Datasets/trip.csv")
trip.registerTempTable("trip")
val tripData = spark.sqlContext.sql("SELECT * FROM trip")
tripData.printSchema()

//make sure each station occur only ones as stations will be the nodes
val justStations = bikeStations
  .selectExpr("float(id) as station_id", "name")
  .distinct()
```

```
//station ids
val stations = tripData
  .select("start_station_id").withColumnRenamed("start_station_id",␣
 ↪"station_id")
  .union(tripData.select("end_station_id").withColumnRenamed("end_station_id",␣
 ↪"station_id"))
  .distinct()
  .select(col("station_id").cast("long").alias("value"))

stations.take(1) // this is just a station_id

//create set of vertices with properties
val stationVertices: RDD[(VertexId, String)] = stations
  .join(justStations, stations("value") === justStations("station_id"))
  .select(col("station_id").cast("long"), col("name"))
  .rdd
  .map(row => (row.getLong(0), row.getString(1)))

stationVertices.take(1)

//create trip edges
val stationEdges:RDD[Edge[Long]] = tripData
  .select(col("start_station_id").cast("long"), col("end_station_id").
 ↪cast("long"))
  .rdd
  .map(row => Edge(row.getLong(0), row.getLong(1), 1))
//add dummy value of 1
stationEdges.take(1)

//build a graph
val defaultStation = ("Missing Station")
val stationGraph = Graph(stationVertices, stationEdges, defaultStation)
stationGraph.cache()

// sanity check
println("Total Number of Stations: " + stationGraph.numVertices)
println("Total Number of Trips: " + stationGraph.numEdges)
println("Total Number of Trips in Original Data: " + tripData.count)
```

```
Intitializing Scala interpreter ...

Spark Web UI available at http://spark-demo-68b9d46d9-m48w9:4040
SparkContext available as 'sc' (version = 3.1.1, master = local[*], app id =␣
 ↪local-1617633065940)
SparkSession available as 'spark'
```

```
root
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- lat: string (nullable = true)
 |-- long: string (nullable = true)
 |-- dock_count: string (nullable = true)
 |-- city: string (nullable = true)
 |-- installation_date: string (nullable = true)

root
 |-- id: string (nullable = true)
 |-- duration: string (nullable = true)
 |-- start_date: string (nullable = true)
 |-- start_station_name: string (nullable = true)
 |-- start_station_id: string (nullable = true)
 |-- end_date: string (nullable = true)
 |-- end_station_name: string (nullable = true)
 |-- end_station_id: string (nullable = true)
 |-- bike_id: string (nullable = true)
 |-- subscription_type: string (nullable = true)
 |-- zip_code: string (nullable = true)

Total Number of Stations: 70
Total Number of Trips: 669959
Total Number of Trips in Original Data: 669959
```

[1]:
```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.functions.col
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.DataFrame
station: org.apache.spark.sql.DataFrame = [id: string, name: string … 5 more
fields]
bikeStations: org.apache.spark.sql.DataFrame = [id: string, name: string … 5
more fields]
trip: org.apache.spark.sql.DataFrame = [id: string, duration: string … 9 more
fields]
tripData: org.apache.spark.sql.DataFrame = [id: string, duration: string … 9
more fields]
justStations: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[station_id: float, name: string]
stations: org.apache.spark.sql.DataFrame = [value: bigint]
stationVertices: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
String)] = …
```

[2]:
```
// Most common distinations from location to location
// Demonstrate usege of the triplets
stationGraph
    .groupEdges((edge1, edge2) => edge1 + edge2)
    .triplets.sortBy(_.attr, ascending=false)
    .map(triplet => "There were " + triplet.attr.toString + " trips fom " +␣
 ↪triplet.srcAttr + " to " + triplet.dstAttr + "." )
    .take(10)
    .foreach(println)
```

```
There were 6216 trips fom San Francisco Caltrain 2 (330 Townsend) to Townsend at
7th.
There were 6164 trips fom Harry Bridges Plaza (Ferry Building) to Embarcadero at
Sansome.
There were 5041 trips fom Townsend at 7th to San Francisco Caltrain (Townsend at
4th).
There were 4839 trips fom 2nd at Townsend to Harry Bridges Plaza (Ferry
Building).
There were 4357 trips fom Harry Bridges Plaza (Ferry Building) to 2nd at
Townsend.
There were 4269 trips fom Embarcadero at Sansome to Steuart at Market.
There were 3967 trips fom Embarcadero at Folsom to San Francisco Caltrain
(Townsend at 4th).
There were 3903 trips fom Steuart at Market to 2nd at Townsend.
There were 3627 trips fom 2nd at South Park to Market at Sansome.
There were 3622 trips fom San Francisco Caltrain (Townsend at 4th) to Harry
Bridges Plaza (Ferry Building).
```

[3]:
```
// as directed graph is used in- and out-degrees of the vertices can be␣
 ↪calculated
// it can be interpreted as number of trips that arrive on a particular station␣
 ↪or departure from it
stationGraph
  .inDegrees // computes in-degree
  .join(stationVertices)
  .sortBy(_._2._1, ascending=false)
  .take(10)
  .foreach(x => println(x._2._2 + " has " + x._2._1 + " in degrees."))

stationGraph
  .outDegrees // out-degree
  .join(stationVertices)
  .sortBy(_._2._1, ascending=false)
  .take(10)
  .foreach(x => println(x._2._2 + " has " + x._2._1 + " out degrees."))
```

```
San Francisco Caltrain (Townsend at 4th) has 63179 in degrees.
```

```
San Francisco Caltrain 2 (330 Townsend) has 35117 in degrees.
Harry Bridges Plaza (Ferry Building) has 33193 in degrees.
Embarcadero at Sansome has 30796 in degrees.
2nd at Townsend has 28529 in degrees.
Market at Sansome has 28033 in degrees.
Townsend at 7th has 26637 in degrees.
Steuart at Market has 25025 in degrees.
Temporary Transbay Terminal (Howard at Beale) has 23080 in degrees.
Market at 4th has 19915 in degrees.
San Francisco Caltrain (Townsend at 4th) has 49092 out degrees.
San Francisco Caltrain 2 (330 Townsend) has 33742 out degrees.
Harry Bridges Plaza (Ferry Building) has 32934 out degrees.
Embarcadero at Sansome has 27713 out degrees.
Temporary Transbay Terminal (Howard at Beale) has 26089 out degrees.
2nd at Townsend has 25837 out degrees.
Steuart at Market has 24838 out degrees.
Market at Sansome has 24172 out degrees.
Townsend at 7th has 23724 out degrees.
Market at 10th has 20272 out degrees.
```

[4]:
```
// application of PageRank algorithm
// returns 10 most significant stations
val ranks = stationGraph.pageRank(0.0001).vertices
ranks.join(stationVertices).sortBy(_._2._1, ascending=false).take(10).foreach(x␣
 ↪=> println(x._2._2))
```

```
San Jose Diridon Caltrain Station
San Francisco Caltrain (Townsend at 4th)
Mountain View Caltrain Station
Redwood City Caltrain Station
San Francisco Caltrain 2 (330 Townsend)
Harry Bridges Plaza (Ferry Building)
Embarcadero at Sansome
2nd at Townsend
Market at Sansome
Townsend at 7th
```

[4]:
```
ranks: org.apache.spark.graphx.VertexRDD[Double] = VertexRDDImpl[954] at RDD at
 VertexRDD.scala:57
```

1.6.3 GraphFrames for Apache Spark

*Sources: * https://livebook.manning.com/book/spark-graphx-in-action/chapter-4/106 * https://docs.databricks.com/spark/latest/graph-analysis/graphframes/index.html*

GraphFrames is a package for Apache Spark that provides DataFrame-based graphs. It provides high-level APIs in Java, Python, and Scala. It aims to provide both the functionality of GraphX and

extended functionality taking advantage of Spark DataFrames in Python and Scala. This extended functionality includes motif finding, DataFrame-based serialization, and highly expressive graph queries. GraphFrames makes use of the Spark SQL component of Spark and its DataFrames API. DataFrames offers much better performance than the RDDs that GraphX uses because of two optimization layers that Spark SQL provides, namely Catalyst and Tungsten.

The following section shall provide an interactive example for the usage of Spark GraphFrames together with PySpark. The dataset at hand consists of Bitcoin transaction graph data downloaded via Google BigQuery. It contains the information of who transacted Bitcoin with whom, naturally manifesting in a graph structure. The datasets in this paper are provided in several sizes, which can be chosen by the user arbitrarily.

First off, we load all dependencies and apply some Jupyter magic in order to make plotting results easier later on. Additionally, we instantiate a Spark Session to interface with Spark. This session is configured to load the graphframes upper-level library on startup, so that we can reference it through the graphframes Python package. To avoid ambiguities, the Python package will hereinafter be referred to as py_graphframes.

———- DISCLAIMER ———-

This code is written in Python3, so please change the notebook kernel accordingly :)

[ ]:
```
import pandas as pd
from graphframes import *
from pyspark import *
from pyspark.sql import *
from pyspark.sql.functions import col
spark = SparkSession.builder.appName('BTC_Analysis').config('spark.jars.
 ↪packages','graphframes:graphframes:0.8.1-spark3.0-s_2.12').getOrCreate()

%pylab inline
```

Next up we proceed to read the dataset into conventional Pandas DataFrames and print some representative records.

[ ]:
```
# Load data downloaded via BigQuery (Dataset: crypto_bitcoin)
df = pd.read_pickle('02_Datasets/Download_Small.pickle')
inputs_outputs = df[df.columns[-2:]].values
print('Amount of Transactions read: ' + str(len(inputs_outputs)))

# Print transaction in readable format
for i in inputs_outputs[:2]:
    print('\n')
    print('-----Inputs-----------------------------')
    for idx in i[0]:
        print(idx['addresses'][0] + ' - ' + str(round(idx['value']/
 ↪100000000,3))) # convert Satoshi to BTC

    print('-----Outputs----------------------------')
    for idx in i[1]:
```

```
        print(idx['addresses'][0] + ' - ' + str(round(idx['value']/
↪100000000,3))) # convert Satoshi to BTC
    print('_____________________________________')
```

Now that we know what our dataset looks like, we parse it into a format py_graphframes is able to transform into a Spark GraphFrame. In the course of doing so we perform several steps: 1. Build DataFrame containing nodes and add a primary key [id] as long 2. Build DataFrame containing edges 3. Identify address clusters (e.g. addresses that belong to the same real entity) via the "Common Spending" heuristic and append cluster information to node DataFrame (in small datasets, the amount of overlaps between spendings tends to be minimal, therefore leading to the amount of interactions and distinct address clusters to coincide, which is what we see here) 4. Build DataFrame containing edges between clusters

[ ]:
```
# Parse data into GraphFrames-readable format
# Assumption: ALL input addresses are parsed as fully connected to ALL output␣
↪addresses

# 1. Nodes
node_set = set()
node_schema = ['id','address','cluster']
for i in inputs_outputs:
    for input_idx in i[0]:
        node_set.add(input_idx['addresses'][0])
    for output_idx in i[1]:
        node_set.add(output_idx['addresses'][0])

node_dict = {y: x for x, y in enumerate(node_set)}
node_data = [(v,k,0) for k,v in node_dict.items()]
print('Amount of distinct addresses: ' + str(len(node_data)))

# 2. Edges
edge_data = set()
edge_schema = ['src','dst','type']
for i in inputs_outputs:
    for input_idx in i[0]:
        for output_idx in i[1]:
            edge_data.
↪add((node_dict[input_idx['addresses'][0]],node_dict[output_idx['addresses'][0]],␣
↪'sent_btc_to'))
print('Amount of distinct mutual interactions: ' + str(len(edge_data)))

# 3. Clustering Algo to cluster nodes based on "Common Spending" heuristic - w/
↪o pyspark or graphframes
idx = 1
node_schema_clustered = ['id','address','address_id'] # id=cluster_id
for i in inputs_outputs:
    # if all none -> All new
```

```
    if all(node_data[node_dict[x['addresses'][0]][2]==0 for x in i[0]):
        for input_idx in i[0]:
            node_data[node_dict[input_idx['addresses'][0]]] =␣
↪(node_dict[input_idx['addresses'][0]],input_idx['addresses'][0],idx)
        idx += 1
    else:
        tmp = min(node_data[node_dict[x['addresses'][0]]][2] for x in i[0] if␣
↪node_data[node_dict[x['addresses'][0]]][2]!=0)
        for input_idx in i[0]:
            if node_data[node_dict[input_idx['addresses'][0]]][2]==0:
                node_data[node_dict[input_idx['addresses'][0]]] =␣
↪(node_dict[input_idx['addresses'][0]],input_idx['addresses'][0],tmp)
            if node_data[node_dict[input_idx['addresses'][0]]][2]!=0 and␣
↪node_data[node_dict[input_idx['addresses'][0]]][2]!=tmp:
                replace_me = node_data[node_dict[input_idx['addresses'][0]]][2]
                node_data[node_dict[input_idx['addresses'][0]]] =␣
↪(node_dict[input_idx['addresses'][0]],input_idx['addresses'][0],tmp)
                for rep in [x for x,v in enumerate(node_data) if␣
↪v[2]==replace_me]:
                    node_data[rep] = (x,node_data[node_dict[rep]][1],tmp)

clustered_node_data_only = [x[::-1] for x in node_data if x[2]!=0]
print('Amount of address clusters acc. to Common Spending Heuristic: ' +␣
↪str(idx-1))

# 4. Build Edges from cluster to cluster (excl. undetected clusters)
clustered_edge_data = set()
clustered_edge_schema = ['src','dst','type']
for edge in edge_data:
    if node_data[edge[1]][2]!=0:
        clustered_edge_data.
↪add((node_data[edge[0]][2],node_data[edge[1]][2],'sent_btc_to'))
print('Amount of distinct mutual interactions of identified clusters: ' +␣
↪str(len(clustered_edge_data)))
```

The next step is to load both the DataFrames describing the nodes and edges of our graph into a Spark GraphFrame. To do so, both Pandas DataFrames are first transformed into Spark DataFrames by providing the records and the respective schema. Afterwards, both Spark DataFrames can be "married" to create a GraphFrame. One thing to note here is that, in order for py_graphframes to successfully create a GraphFrame, the schema of the data describing the edges must contain the columns [src,dst,type], where [type] refers to the label of the respective edge. Additional columns can be provided to define further edge attributes. Similarly, the schema of the data describing edges must contain an [id] column.

In this example, we continue our analysis with the transaction graph aggregated to address clusters.

[ ]:
```
# Load data into Spark GraphFrames
vertices = spark.createDataFrame(clustered_node_data_only,node_schema_clustered)
edges = spark.createDataFrame(clustered_edge_data,clustered_edge_schema)

gf = GraphFrame(vertices,edges)
gf.vertices.show()
gf.edges.show()
```

Using this GraphFrame, we can now perform analysis with the data at hand. The following section provides the following examples: 1. Analysis of address cluster sizes 2. Analysis of address cluster interactions via node degrees 3. Analysis of address cluster importance via PageRank

This analysis is not meant to provide a thorugh analysis of the BTC transaction grph, but rather to showcase the data analysis tools provided by PySpark and GraphFrames for Apache Spark.

[ ]:
```
# 1. Analysis of address cluster sizes
# This section performs analysis on the transaction graph based on research␣
 ↪conducted by
# Dorit Ron, Adi Shamir: Quantitative Analysis of the Full Bitcoin␣
 ↪Transactiongraph, https://eprint.iacr.org/2012/584.pdf


# ---------- Without PySpark+GraphFrames ----------
print("These results are produced solely with Python+Pandas")

# Read clustered nodes in Pandas DataFrame
df = pd.DataFrame(clustered_node_data_only)

# Count addresses per cluster
df = df[[0,1]].groupby([0]).agg('count')

# Filter for only clusters with at leatst 10 addresses
df_filtered = df.loc[df[1]>30]

# Plot histogram of addresses per cluster
df.plot.hist(title='Amount of addresses per cluster (unfiltered)', bins=50)
df_filtered.plot.hist(title='Amount of addresses per cluster (filtered)',␣
 ↪bins=50)
plt.show()

# Print max amount of addresses per cluster in dataset
print("Maximum addresses per cluster in selected dataset: " + str(max(df[1].
 ↪values)))


# ---------- With PySpark+GraphFrames ----------
print("\n\nThese results are produced with PySpark+GraphFrames")
```

```
# Perform GroupBy+Count in Spark DataFrame
spark_df = gf.vertices.groupBy('id').count()

# Read clustered nodes in Pandas DataFrame
pandas_df = spark_df.toPandas().set_index('id')

# Read only clusters with more than 10 addresses in Pandas DataFrame
pandas_df_filtered = spark_df.filter(col('count')>30).toPandas().set_index('id')

# Plot histogram of addresses per cluster
pandas_df.plot.hist(title='Amount of addresses per cluster␣
 ↪(unfiltered)',bins=50, ylabel="Test")
pandas_df_filtered.plot.hist(title='Amount of addresses per cluster␣
 ↪(filtered)',bins=50)

# Print max amount of addresses per cluster in dataset
print("Maximum addresses per cluster in selected dataset: " + str(spark_df.
 ↪groupby().max('count').first().asDict()['max(count)']))
```

Based on the above results we can conclude that most address clusters contain only a small amount of addresses. By filtering for only those address clusters with at minimum 25 addresses to their name, we can see that there is an interesting accumulation of clusters containing around 125-150 addresses.

[ ]:
```
# 2. Analysis of address cluster interaction via node degrees


# ---------- With PySpark+GraphFrames ----------
print("\n\nThese results are produced with PySpark+GraphFrames")

gf.degrees.sort(col("degree").desc()).toPandas().set_index('id').plot.
 ↪hist(title="Address clusters by degree (unfiltered)",bins=25)
gf.inDegrees.sort(col("inDegree").desc()).toPandas().set_index('id').plot.
 ↪hist(title="Address clusters by ingoing degree (unfiltered)",bins=25)
gf.outDegrees.sort(col("outDegree").desc()).toPandas().set_index('id').plot.
 ↪hist(title="Address clusters by outgoing degree (unfiltered)",bins=25)

gf.degrees.sort(col("degree").desc()).filter(col('degree')>5).toPandas().
 ↪set_index('id').plot.hist(title="Address clusters by degree␣
 ↪(filtered)",bins=25)
gf.inDegrees.sort(col("inDegree").desc()).filter(col('inDegree')>5).toPandas().
 ↪set_index('id').plot.hist(title="Address clusters by ingoing degree␣
 ↪(filtered)",bins=25)
gf.outDegrees.sort(col("outDegree").desc()).filter(col('outDegree')>5).
 ↪toPandas().set_index('id').plot.hist(title="Address clusters by outgoing␣
 ↪degree (filtered)",bins=25)
```

Similar to previous results, we note that many address clusters do not interact very much. Most clusters reside below a degree of 5. Though, taking a look at those clusters that are more active, it can be seen that ingoing edges behave differently from outgoing edges. While the amount of clusters with a certain amount of ingoing edges declines exponentially in the amount of ingoing edges, the amount of clusters with a certain amount of outgoing edges declines almost immediately.

[ ]:
```
# 3. Analysis of address cluster importance via PageRank


# ---------- With PySpark+GraphFrames ----------
print("\n\nThese results are produced with PySpark+GraphFrames")

result = gf.pageRank(resetProbability=0.15, tol=0.01)

# group by cluster ids to not list clusters multiple times
result_df = result.vertices.groupBy('id').max()

result_df.select("id", "max(pagerank)").sort(col("max(pagerank)").desc()).show()

result_df.select("id", "max(pagerank)").toPandas().set_index('id').plot.
 ↪hist(title="Address clusters by PageRank (unfiltered)",bins=50)
result_df.select("id", "max(pagerank)").filter(col('max(pagerank)')>5).
 ↪toPandas().set_index('id').plot.hist(title="Address clusters by PageRank␣
 ↪(filtered)",bins=50)
```

The code above also showcases the ease with which an algorithm like PageRank can be executed via Spark GraphFrames. Excluding the adjustements to our specific data structure, one line suffices. A caviat with the implementation of PageRank in GraphFrames though, is that it is not possible to run a weighted version of PageRank out of the box. If we for example wanted to weigh edges according to their value in BTC, we would have to manually implement our own version of PageRank.

1.6.4 Other Systems compared to Spark GraphX

In the following section GraphX is compared to othe graph database systems. Neo4J is a graph database wich was covered in the lecture. The main differencies between Neo4J and GraphX are addressed in the following. As GraphX is not a graph database but graph processing system, it should be rather used as a complement to Neo4J and not as an opposite to it. The possible ways for integration of the two systems are described. Furthermoe, another graph processing system - Giraph is introduced. Giraph and GrapX are comparable systems and three research studies on performance comparison are represented in the following section.

Neo4j *Sources: * Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB https://www.scitepress.org/papers/2018/69102/69102.pdf * Research on architecture and query performance based on distributed graph database Neo4j https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6703387 * Graph Databases: Neo4j Analysis https://www.scitepress.org/Papers/2017/63560/63560.pdf * https://livebook.manning.com/book/spark-graphx-in-action/chapter-1/66*

Neo4J is an open-source graph database implemented in Java. The developers describe Neo4J as

a fully transactional database and a persistent Java engine where data structures in the form of graphs can be stored. Neo4J uses natives graph storage which provides the freedom to manage and store data in a highly disciplined manner. The Neo4j can run in the embedded mode, as well as in server mode. Embedded mode must be understood as an instance that stores all the information on the disk, and not as an in-memory database, that works exclusively with the internal memory of the machine. The embedded Neo4j is ideal for hardware devices, desktop applications, and for embedded applications in servers. Running Neo4j in server mode guarantees a more common way to implement a database, which is what is most used today. On each server there is an embedded instance of Neo4j. Neo4J has many competitive advantages, which makes this software one of the most used ones in this area. The major features of Neo4J are the following: flexible schema, Cypher query language, HHTP API to manage the database, inde support using Apache Lucence, drivers support for Java, Spring, Scala, JavaScript, online backups, exporting of query data to JSON and XML formats, most active graph community in the world, easy to learn and to use, whiteboard-friendly data modelling to simplify the development cycle. Neo4J has community and commercial versions. In the community versio Neo4J doesn't support sharding and has some limitations on the number of nodes, relationships and properties.

Cypher Query Language *Sources: * https://neo4j.com/developer/cypher/*

Cypher is Neo4j's graph query language that allows users to store and retrieve data from the graph database. Cypher aims to make querying graph data easy to learn, understand, and use for everyone, but also incorporate the power and functionality of other standard data access languages. Cypher's syntax provides a visual and logical way to match patterns of nodes and relationships in the graph. It is a declarative, SQL-inspired language for describing visual patterns in graphs using ASCII-Art syntax. Through Cypher, users can construct expressive and efficient queries to handle needed create, read, update, and delete functionality. Cypher is open source. The openCypher project provides an open language specification, technical compatibility kit, and reference implementation of the parser, planner, and runtime for Cypher.

Neo4J not vs. but rather with Spark GraphX *Sources: * https://neo4j.com/developer/apache-spark/*

As already mentioned, Apache Spark GraphX is not a graph database. Instead, it's a graph processing system, which is useful, for example, for fielding web service queries or performing one-off, long-running standalone computations. Because GraphX isn't a database, it doesn't handle updates and deletes like Neo4j, which is graph databases. Graph processing systems are optimized for running algorithms on the entire graph in a massively parallel manner, as opposed to working with small pieces of graphs like graph databases. Therefore, Neo4J is not an opposite technology to GraphX and can be used rather as a complement. For example, when Neo4J needs to process the very large data-sets and real time processing to produce the graphical results/representation it needs to scale horizontally. In this case combination of Neo4J with Apache Spark will give significant performance benefits in such a way Spark will serve as an external graph compute solution. Neo4J can be integrated with Spark in a variety of ways. It can be used to pre-process (aggregate, filter, convert) raw data to be imported into Neo4j. Spark GraphX can also serve as graph processing solution. So, data of selected subgraphs can be exported from Neo4J to Spark, where analytic spects are computed. The computation results are written back to Neo4J and can be used for further operations and Cypher queries.

Neo4j Connector for Apache Spark *Sources: * https://neo4j.com/developer/spark/overview/ * https://neo4j.com/blog/neo4j-3-0-apache-spark-connector/*

The Neo4j Connector for Apache Spark is intended to make integrating graphs together easy. There are effectively two ways of using the connector: - As a data source: read any set of nodes or relationships as a DataFrame in Spark; - As a sink: write any DataFrame to Neo4j as a collection of nodes or relationships, or alternatively use a Cypher statement to process records in a DataFrame into the graph pattern of your choice. The connector currently supports: RDD, DataFrames, GraphX, GraphFrames.

Neo4J - Mazerunner *Sources: * https://neo4j.com/developer/apache-spark/#mazerunner*

Another integration solution is Neo4J Mazerunner, which allows exporting dedicated datasets, e.g. node or relationship-lists to Apache Spark. It supports PageRank, Closeness Centrality, Betweenness Centrality, Triangle Counting, Connected Components, Strongly Connected Components algorithms. After running graph processing algorithms in GraphX the results are written back concurrently and transactionally to Neo4j. One focus of this approach is on data safety, that's why it uses a persistent queue (RabbitMQ) to communicate data between Neo4j and Apache Spark. The infrastructure is set up using Docker containers, there are dedicated containers for Spark, RabbitMQ, HDFS and Neo4j with the Mazerunner Extension.

Spark Graph Module *Sources: * https://databricks.com/session_eu19/graph-features-in-spark-3-0-integrating-graph-querying-and-algorithms-in-spark-graph-continue*

Furthermore Apache Spark promises an integration of the Neo4J's query language - Cypher into Sparkecosystem. According to the articale from Databricks Spark 3 introduces a new module: Spark Graph. Spark Graph adds the popular query language Cypher, its accompanying Property Graph Model and Graph Algorithms to the data science toolbox. Spark Graph interacts with Spark SQL and openCypher Morpheus, a Spark Graph extension allows to manage multiple graphs and provides built-in Property Graph Data Sources for the Neo4j graph database as well as Cypher language extensions.

Apache Giraph *Sources: * https://giraph.apache.org/intro.html * https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920/ * https://arxiv.org/pdf/1806.08082.pdf*

Apache Giraph is an iterative graph processing system built for high scalability. It is originated as the open-source counterpart to Pregel. Giraph is an iterative graph processing framework, built on top of Apache Hadoop. Giraph adds several features beyond the basic Pregel model, including master computation, sharded aggregators, edge-oriented input, out-of-core computation, and more. Giraph is a computing platform, and therefore it needs to interface with external storage in order to read the input and write back the output of a batch computation.

Comparison on Performance, Scaling and Fault tolerance *Sources: * https://amplab.cs.berkeley.edu/wp-content/uploads/2014/09/graphx.pdf * https://engineering.fb.com/2016/10/19/core-data/a-comparison-of-state-of-the-art-graph-processing-systems/ * https://arxiv.org/pdf/1806.08082.pdf*

The underlying Apache Spark technology allows to horizontally scale GraphX which brings obvious performance benefit. A performance study was conucted at Berkley University. The results of the study showed that increasing number of machines from 8 to 32 caused 3x time performance speedup. In regard to fault tolerance Apache Spark brings a benefit as well as it provides lineage-based fault tolerance with negligible overhead as well as optional dataset replication. Another quantitative and qualitative benchmarking study was conducted by Facebook in 2016 GraphX was compared to Giraph. The Facebook researchers were especially interested in measuring the relative performance and ability of the two systems to handle large graphs. While performance was the main focus, they also studied usability and how easy it is to develop applications in the two systems.The experimental setup consists of executing PageRank, Connected Components, and Triangle Counting implementations on various datasets exhibiting different degrees of skewness in their graphs, namely Facebook and Twitter datasets. Based on these experiments the authors conclude that Apache Giraph is the overall more efficient graph processing platform, being able to process up to 50x larger graphs than GraphX, while GraphX is better suited to rapid development of small applications. The main advantages of the GraphX platform are made out to be it's efficiency in terms of development speed and integration efforts. GraphX can read and preprocess data from Hive via an SQL-like interface, whereas Apache Giraph requires extra integration efforts. In addition, GraphX's shell interface allows for rapid testing of simple applications. Experimental analysis of distributed graph systems was contucted in University of Waterloo in 2018. In this study Öszu et al. perform an extensive comparison of graph processing system architectures, focusing on MapReduce (e.g. GraphX, HaLoop) and Bulk Synchronous Processing (BSP; e.g. Apache Giraph, Blogel). The authors determine performance statistics on workloads including PageRank, calculation of weakly connected components and single source shortest paths on datasets including Twitter and World road Network. The authors report an overall superiority of those systems utilizing BSP architecture over MapReduce. Of those BSP systems Blogel-V performed best in most setups. It is stated that GraphX is not efficient for graph workloads or datasets that require large number of iterations. Futhermore, the authors point out that GraphX suffers from load balancing issues when scaling to large datasets. Graphframes is suggested as an alternative, offering more efficient implementations of algorithms for specific tasks.

1.7 Conclusion

In this paper we have introduced the core characteristics of both graph database systems and graph analytics systems and established the symbiotic potential of both archtiectures. While graph dabaase systems allow for efficient storage and data retrieval for graph datasets, graph analytics systems provide efficient means of analytics via predefined algoriths and interfaces to common (graph-) databases for frictionless data integration and preprocessing.

We have performed two example use-cases of GraphX and GraphFrames via their Scala API/Python3 API respectively. These allow the reader to familiarize with the systems at hand and use our implementation as a basis for their own purposes.

For the use cases outlined in this paper, which focus on rapid explorative data analysis, the nature of both GraphX and GraphFrames proves to be beneficial. Especially GraphFrames with its Python3 API and additional abstraction layers allow for plug-and-play integration into existing data processing pipelines and delivers high performance at the same time. For use-cases that are oriented towards production grade data crunching, systems like Apache Giraph and Pregel, which embrace on the BSP paradigm, outperform GraphX and GraphFrames according to existing scientific literature and are therefore more suited for these tasks.