Course Project: Querying over Property Graphs

1st Sarthak Joshi
Ph.D., CDS department
Indian Institute of Science

2nd Burhanuddin Kamalapur Wala
M.Tech., CSA department
Indian Institute of Science

***Abstract*—Various fields use property graphs to organise data from multiple sources. The stored data is often queried to access valuable information. But with fast increasing amounts of newer data, these graphs have become more and more complex over time. Therefore, it is necessary to devise a scalable method to query them efficiently. In this project, we have designed a method to query property graphs using Spark and the Graphframes library.**

***Index Terms*—Property Graph, Big Data, Spark**

I. Introduction

A property graph associates a key-value pair properties to a graph's vertices and edges on top of its underlying structure. The vertices of the graphs are used to represent objects while the edges are used to represent the relationship across those objects. Through this, a large amount of varied information can be stored as a network of related objects while both the objects and the relations have their individual parameters stored as key-value pairs. Property graphs are used in various forms depending on the type of information the vertices and edges represent. They can be used as knowledge graphs, social networks, financial transactions, etc. Therefore, a large variety of fields employ them for various purposes.

A. Motivation

Throughout the fields in which they are employed, property graphs are used to extract information about its nodes (or vertices) from a complex set of relationships (or edges). For this purpose, property graphs are queried to obtain a set of nodes that satisfies a set of relationships. As these graphs have grown from increasingly complex and varied amounts of data, querying them quickly has become more and more difficult while the queries have also gotten more complicated. Each individual query has a higher number of relations it must satisfy as well as a higher number of unknown variables it must generate values for. Therefore, it is essential to have a fast and scalable method to query these graphs.

Our primary objective is to efficiently query these graphs using Apache Spark. We use the Graphframes library for the same. We are using this big data platform due to its iconic resilient distributed datasets or RDDs, which are a collection of elements partitioned across the nodes that can be operated on in parallel. RDDs are lazily generated when they are needed using a lineage map which allows for many logical optimizations to be added in the deterministic output generation workflow before the final result is computed. Spark also offers SparkSQL which is an efficient SQL querying mechanism that can be used to create and modify Spark Dataframes. Spark Dataframes are essentially a set of RDDs abstracted into a table-like format. This format allows for querying using standard SQL queries which are offered as functions in the platform. The Graphframes library further extends the RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge. This precisely fits what we require to represent our property graph.

B. Our Contribution

The Graphframes library generates a Graphframe object. It takes a vertex dataframe containing all the information about each vertex and a edge dataframe containing all the information about each edge as input. Queries on property graphs, called path queries, allow the user to define a sequence of predicates over vertices and edges that should match along a path in the graph. These queries can then be extended to obtain specific results from the resulting set of possible values that each vertex and edge can take while satisfying the structural pattern specified in the path. We can further break this down into a set of pairwise relationships where we need to obtain all possible values of all unknown variables. Our basic approach would involve using the Graphframes library's motif finding technique to generate a path table containing all the possible vertices and edges that satisfy the structural pattern specified by the query. Then we would apply filters to all the edges and vertices as specified by the query. Finally the filtered path table will be used to generate the output by applying generic SQL operations. As queries get more complicated, we can break them down into smaller queries, execute in parallel and merge the outputs at the end.

II. Related Work

There exist popular query models over property graphs and associated query engines [1]–[3]. SPARQL offer a simple and highly flexible syntax, but is costly to execute in practice for large graphs [4]. Others offer more strict query primitives over property graphs, such as finding paths, patterns and reachability to a vertex while scaling using a distributed execution model for scaling to larger graphs [5]. Our work is exists a middle point between the two as we

use a distributed execution model for scaling but also offer a more flexible syntax due to our query decomposition stage. This is explained in a later section.

It has been shown that an SQL-like implementation outperforms non-SQL implementations for querying property graphs [11]. This is because the SQL platforms offer a more powerful set of tools to easily identify useful information and prune the impossible outputs. While the SQL implementations are much more structured, it is still compatible with the way property graphs are defined, i.e., a set of key-value pairs per vertex and edge which are arranged in an underlying graph structure. We use this implementation to our advantage using the Graphframes library which stores a graphframes object as a combination of a vertex and edge dataframe. The dataframes can individually hold these key-value pairs as column entries such that each entry of the vertex dataframe would have a unique id column as a key and other properties of that vertex as value columns. Meanwhile, each edge can have two keys identifying its corresponding vertices and the rest of the rest of the columns could be the properties of the edge as values columns.

There exist other implementation of utilizing RDDs in Spark to essentially cache the query for quickly returning queries in the future [6] (called faceted search there). However, they are limited to knowledge graphs which is a subset of the property graphs where we are applying our model. Furthermore, they do not utilize the efficient tools provided by the GraphX library and instead try to manually store the graph information in a less optimal manner. They also only consider exactly matching queries for use when a new query arrives and do not attempt to apply the newer query on top of the older query or the lineage of the older query.

There have also been attempts to partition a large-scale property graph's vertices to perform efficient distributed query processing [10]. In there, the graph is divided into sections in which paths are computed by a common partition. However, in the cases where many paths involve edges across partitions, the communication cost can get very high. Furthermore, this communication cost can vary greatly across graph to graph, partition to partition as it is dependent on the number of edges that cross the partition boundaries. Partitioning along query types at run time is more efficient as the number of the types are limited and the same types require the same set of comparisons to be performed to confirm if an edge satisfies an input property. This allows for easier load balancing and faster execution per partition and less communication across partition as only the final results of the local computations are used for communication.

III. Problem statement

We are given an input as an array of conditions that need to be applied to a structural graph pattern which is also part of the input. We need to obtain certain information from the graph out of all the vertices and edges that satisfy those conditions. We need to provide the user an interface to easily input their structural pattern and the conditions on each vertex and edge which we would then apply to the graph to obtain the output.

IV. Workflow

Before the querying begins, the graph dataset needs to be appropriately modified to allow for the creation of the Graphframe object. Graph datasets, like the LBDC social network benchmark, consist of a number of vertices belonging to different classes and a number of edges representing different relation types. Each vertex of a unique class and each edge representing a unique relation is stored in a table. The vertex table for each class contains an "id" column. The edge table for each relationship contains two columns containing the ids of the the source and destination.

The Graphframe object requires a vertex dataframe containing all the information about all the vertices and an edge dataframe, containing all the information about all the edges. The vertex dataframe must have a column named "id" that can be used to uniquely identify any vertex. The edge dataframe must have a column named "src" and a column named "dst" that contain the unique ids of the vertices. To facilitate this, we create the vertex dataframe as a table consisting of columnns from all the vertex tables.

The table is populated with entries from vertex tables of all the classes. Columns that do not exist for a particular vertex class are left null in their corresponding entries. Similarly, the edge dataframe is created as a table consisting of columns from all the edge tables. The table is populated with entries from edge tables representing all the relationships. Columns that do not exist for a particular edge relationship are left null in their corresponding entries.

Following that we add the id, src and dst columns to the vertex dataframe and edge dataframe respectively. If these column names already exist, then they are renamed to a different name. This is because the id may be unique for a class but not necessarily across multiple classes. The id column for a vertex is defined as a combination of the unique id within a class and the class name of that vertex. This guarantees uniqueness of that vertex among all the vertices across all the classes. The src and dst columns are populated with the corresponding class unique id and class name combination of the source vertex and the destination vertex.

Our scalable implementation of querying large property graphs consists of four phases, namely, Query Decomposition, Motif Finding, Filtering and Result Generation.

A. Query Decomposition

Path queries, as their name suggests, output paths from the property graph that satisfy the input properties. These input

properties are essentially an operation on the key-value pairs stored in the vertices and edges of the property graph that give a boolean (True/False) output. If the output is True, then the property is considered satisfied. We can further break down any such path query into a set of edges connecting a pair of vertices in which all 3 satisfy some input properties. This is equivalent to dividing the path into individual one-to-one links. If the path query involves multiple possible paths (an OR condition), we break it down into two sets of separate path queries which are both forwarded to the next phase.

B. Motif Finding

Once the query has been broken into a set of single edge queries, they are represented as a set of edges with matching vertices signifying the a link from one edge to another through a common vertex. Graphframes provides a method named find that takes a motif finding string as an input and returns a path dataframe as an output. We use this method to generate the path table that is used to compute the output.

The find method takes a motif finding string as an input. A motif finding string is essentially a representation of a path as a combination of single edges. Its basic format is "(x)-[e]->(y)". Here x represents the source vertex, y represents destination vertex and e represents the edge connecting them. All the vertices are enclosed by round brackets and all the edges are enclosed by square brackets. Using this basic string as an input would give an output path table which would essentially be an edge list. It would have three columns, namely x, e and y. The entries of this table will be all the possible combinations of vertices and edges that satisfy this structural pattern. Therefore, without any further constraints, this will be an edge list as the pattern represented in this string is a single edge which would then output all the existing edges from the graph. The individual columns x and y would have entries that consist of an entire row from the vertex dataframe. Similarly, e would have entries that consist of an entire row from the edge dataframe.

This motif finding string can be further expanded by concatenating multiple strings with a semicolon delimiter. This would expand the path table to include all the possible combinations of the outputs of the two concatenated strings. Therefore, the output for the string "(x)-[xy]->(y) ; (a)-[ab]->(b)", would be a cross product of the edge list with itself as all the edge outputs of the first string would be extended using all the outputs of the second string. This would result in a path table with six columns for the labels in string.

Furthermore, the labels used to define the vertices and edges in this string can be used to uniquely identify them. Therefore, by using a common label for some vertex in two different strings would only create a single column for that label which would allow for an entire path to be created using individual edges. For example, the string "(x)-[xy]->(y) ; (y)-[yz]->(z)" would generate a path table output representing the a path of length two. Here, there would only be five columns and with y representing the intermediate vertices in the path. Therefore, all the cases in which y did not have an outgoing edge in the basic string would be automatically removed. This is equivalent to computing the path tables for the two strings and then joining them using y as the join key. This can be expanded infinitely to obtain a path table for any given length. More complicated cases, like multiple paths converging to one vertex or multiple paths diverging from one vertex can also be decomposed into strings representing paths of various lengths, converging at or diverging from a common vertex.

C. Filtering

From the possible set of vertices in edges obtained as a result of individual motif strings, represented as a path table, we next filter out all the edges that do not satisfy the input property. As described previously, each entry of the path table is the entire row of the vertex or edge that it is representing from their corresponding vertex and edge dataframes. As this path table is also a dataframe, standard SparkSQL querying can be used on it. Therefore, first, each of the edges in the path are filtered such that they satisfy the conditions defined in the input. These conditions can be checked using the SparkSQL filter method. The individual entries of a row of the edge dataframe can be accessed using the "." operator. For example, let the edge dataframe have a column called "relationship" that represents the unique name of the relationship between the two vertices which that edge connects. Now let us say form a graphframe object and use the basic motif finding string to get a path table called "path" which is essentially an edge list with vertices x and y, and edge e. Now if we need to filter out only the edges that have have the relationship named "knows", we can apply the filter method of SparkSQL to filter out the edges that do not satisfy the given condition. This command would look like out=path.filter(col("e.relationship")=="knows").

By applying this condition to individual edges we filter out entire paths from the path table. These paths are all the possible paths and only some of them are needed as while they satisfy the underlying structural pattern of the path query, they do not satisfy the conditions that are set on the individual edges. Following this, we can similarly filter out paths that do not satisfy the conditions on the individual vertices. After this filtering process, each entry in the path table would represent the a path that matches the structural pattern of the defined in the input and also satisfies all the conditions defined in the input.

D. Result Generation

With all the paths satisfying all the properties available, the path table can now be extended with more columns that are derived from the filtered entries. It can also have its individual vertex and/or edge column entries that are rows of vertex and edge dataframes respectively get expanded into individual columns representing the columns from the vertex and edge

dataframes. From here on, the desired output can be generated using standard SparkSQL operations by new deriving columns using the columns of the path table, further filtering, mapping or adding new entries by checking for conditions on the newly derived columns, repeating the process until the data required in the output is fully generated and selecting the columns required in the output.

V. Implementation

Using the graphfames library we can search path queries but sometimes we may want to filter out nodes based on attributes of vertex or relation of edges which can not be implemented using motif strings. Hence, we facilitate a way to search these kinds of queries efficiently using graphframes and sparkSQL. To provide a simpler interface for inputting queries, a component class has been created. Each component in the query represents a vertex or node and a next edge associated with it. Last node will have the next edge as null. To query graphs, users can create each component individually and then merge them to make a chain of path (similar to linked list) or alternatively we have provided a function to form a query out of array where certain rules need to be followed.

The following steps need to be followed to create query using components:

1) Create component objects for each of the node and its next edge, In case of last node next edge will be None.
2) To create component object use the command, component(conditions=[], v_name="", e_name="", relation="", next=None). The required parameters here are:
   a) v_name: Represents column name which we get in output data frame corresponding to vertex.
   b) e_name: Represents column name which we get in output data frame corresponding to edge.
   c) relation: Values of relation on edge.
   d) conditions: Attribute values of particular vertex to be satisfied
   e) next: Pointer to next component.
3) Merge these components to form chain of components using the function merge_components(*components, edge_names=[], relations=[]). The required parameters here are:
   a) components: Comma separated heads of component chain
   b) edge_name: names of edges between two consecutive components.
      i) Length of this array should be the number of components-1.
      ii) Empty array represents the names of all edges are "".
   c) relations: Relation on edges between two components in chain.
      i) Length of this array should be the number of components-1.
      ii) Empty array represents the names of all edges are "".
4) Call the function searchByComponents(component) to get results.

The following steps need to be followed to create queries using arrays:

1) Make an array with n elements, each element will represent a vertex and next edge.
2) Each element in the array is a tuple of size 4.
3) Array = [element_1, element_2,....,element_n]
4) element_k = [vertex_name, conditions, edge_name, relation]
   a) vertex_name (string): v_name of component
   b) edge_name (string): e_name of component
   c) relation (string): Relation corresponding to edge
   d) conditions (array): conditions on different attributes [[att1, val1], [att2, val2], [att3, val3]
5) Call the function searchByArray() to get results

These two methods will help user to write queries easily and define them precisely. There are also two functions which can be used to get queries in the form of a string by passing component:

1) get_query_string_without_conditions(component) =¿ query
   a) component: First component of the chain
   b) output: Outputs query in string format for graphframes (motif finding string)
2) get_query_string_with_conditions(component) =¿ (query, conditions, relations)
   a) component: First component of the chain
   b) output: Outputs query, conditions and relations:
      i) query: Query in string format for graphframes (motif finding string)
      ii) conditions: Array of conditions on vertices.
      iii) relations: Array of relations on edges between vertices

Using the above mentioned methods, we can allow the user to generate whatever structural pattern they require and allow them to apply all the required conditions to each vertex and edge.

VI. Experiments

We have tested this setup on one of the sample graphs generated with LBDC's Social Network Benchmark's publically available generators. LDBC's Social Network Benchmark (LDBC SNB) is an effort intended to test various functionalities of systems used for graph-like data management. For this, LDBC SNB uses the recognizable scenario of operating a social network, characterized by its graph-shaped data. LDBC SNB consists of two workloads that focus on different functionalities: the Interactive workload (interactive transactional queries) and the Business Intelligence workload (analytical queries). We have used the Interactive workload.

A. Graph Description

The LBDC SNB's interactive workload has a complex set of vertices and relationships as represented in Figure 1 taken from the official documentation.

It consists of a dynamic and a static set of vertices. The dynamic set of vertices would often get added/removed/modified in a social network. These include the vertex classes Person, Forum, Post, Comment and Message (which consists of both Posts and Comments). The static set of vertices would generally remain the same for long periods of time. These consist of Company, University, Organization (which consists of both University and Company), Tag, TagClass, City, Country, Continent and Place (which consists of City, Country and Continent). The vertex classes are related through a variety of relationships represented as edges some of which have their own properties like creationDate and creationTime.

B. DataFrame Creation

The Graph is provided as a multi-level directory consisting of multiple tables stored in the .csv format. Some folders in the directory with the same name as the vertex class store tables containing information about the vertex entries of that class. Other folders of the name format, "class_relation_class" store tables containing information about edges connecting the vertices that represent relationships across two vertex classes. We merge all the columns across all the vertex classes (drawn from directory names) to create a combined vertex dataframe. The columns that do not exist for a particular vertex class are left null for all entries of that vertex class. Similarly, we merge all the columns across all the edge relationship types (drawn from the directory name) into a combined edge dataframes. The columns that do not exist for a particular edge relationship type are left null for all entries of that edge relationship type. Furthermore, the unique id for each vertex is defined as a combination of its class id and its class name. Finally the GraphFrame is generated using these two merged DataFrames.

C. Querying the Graph

We ran three of the Interactive complex queries defined in the LBDC SNB official documentation [12]. The first query (IC 1) involves listing the details of Persons who at related to the Person with the given input id with 1, 2 or 3 steps of knows relation. To solve this, we searched for a structural pattern corresponding to a path of length 3, filtered the knows relation for each vertex, filtered the source vertex to have the input id and then merged all distinct Persons in the other vertex into a single column entry along with another column representing their distance from the source. Finally, we expanded the table into it into having all the required columns to be outputted and distributed the Person entries (full rows) across those columns.

The second query (IC 2) involves listing details of the Person who is known by the Person with the input id and all of their recently created messages which are newer than an input date. To solve this, we searched the structural pattern of two edges converging to the same vertex and filtered the edge relationships of knows and hasCreator from them. Then the input id is filtered from the source vertex of the knows edge and the creation date threshold is checked in the source verted of the hasCreator edge. Then the required output details are given after expanding the table to include required columns from the full row entries.

The third query (IC 7) involves finding the most recent likes on any of input Person's Messages, finding Persons that liked any of input Person's Messages, and the latency in minutes between creation of Messages and like. To solve this we searched the structural pattern of a path of length 2, filtered the first edge to have the relation likes and the second edge to have the relation hasCreator. Then the destination vertex is filtered to have the input Person id. Then the output is obtained by extracting required columns from the full row entries. The latency in minutes is computed as the difference between the creationDate of the message (stored as part of the middle vertex entry) and the creationDate of the like (stored as part of the likes edge).

We were able to successfully execute the queries in Google Colab Pyspark environment and obtain the desired outputs from them.

VII. Conclusion

In this project, we have designed a querying method for property graphs which would be implemented using Apache Spark with the GraphFrames library. We have provided the user with an interface to generate the required structural patterns for their queries with the required conditions and successfully generated the desired outputs.

VIII. Future Work

There is much work that can be further done on this project which involves optimizing the techniques to detect and filter the condition that would have the most impact on the table first, comparing the performance and the scalability with other Graph Querying platforms and exploiting the persistent and immutable nature of RDDs (and thus DataFrames) to cache the output for a given query input to be easily returned later and even helping in computing some of the results of a similar query.

References

[1] D. Chavarría-Miranda, V.G. Castellana, A. Morari, D. Haglin, J. Feo, Graql: A query language for high-performance attributed graph databases, in: IEEE IPDPS Workshops, 2016.

[2] J. Shi, Y. Yao, R. Chen, H. Chen, F. Li, Fast and concurrent RDF queries with RDMA-based distributed graph exploration, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16, 2016, pp. 317–332.

[3] J. Zhou, G.V. Bochmann, Z. Shi, Distributed query processing in an ad-hoc semantic web data sharing system, in: IEEE IPDPS Workshops, 2013.

[4] SPARQL query language for RDF, 2008, URL http://www.w3.org/TR/rdf-sparql-query/

[5] M. Sarwat, S. Elnikety, Y. He, M.F. Mokbel, Horton+: A distributed system for processing declarative reachability queries over partitioned graphs, sPVLDB 6 (14) (2013)

[6] A. Mukhopadhyay, H. Kim and K. Anyanwu, "Scalable Exploratory Search on Knowledge Graphs Using Apache Spark," 2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2018, pp. 154-159, doi: 10.1109/WETICE.2018.00036.

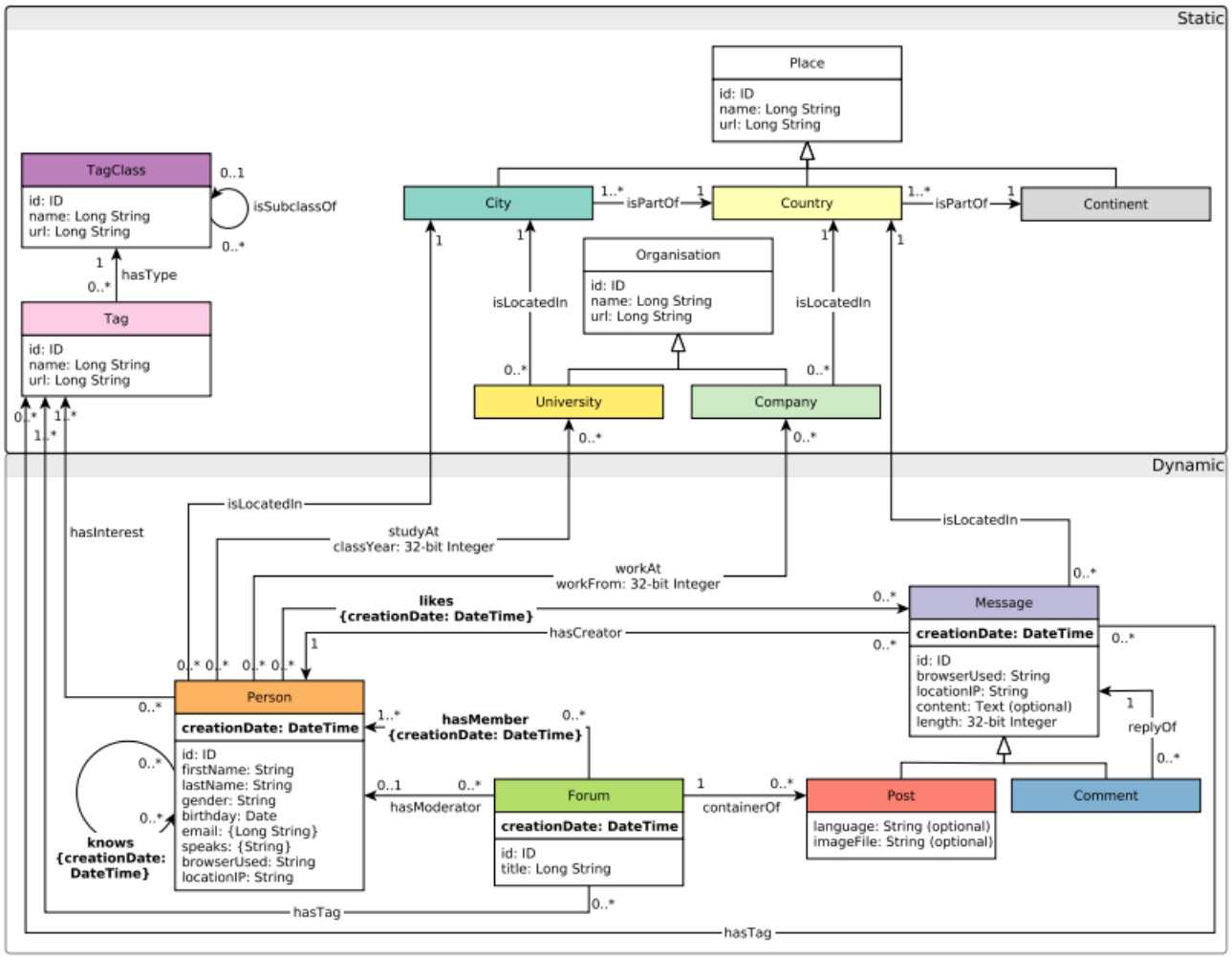

Fig. 1. LBDC SNB Graph Structure

[7] The LDBC Social Network Benchmark (Version 0.3.2), Technical Report, Linked Data Benchmark Council, 2019.

[8] Neo4J graph platform, 2020, URL https://neo4j.com/.

[9] A. Sharp, et al., Janusgraph, 2020, https://janusgraph.org/.

[10] H. Pang, P. Gan, P. Yuan, H. Jin and Q. Hua, "Partitioning Large-Scale Property Graph for Efficient Distributed Query Processing," 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, pp. 1643-1650, doi: 10.1109/HPCC/SmartCity/DSS.2019.00225.

[11] D. Anikin, O. Borisenko and Y. Nedumov, "Labeled Property Graphs: SQL or NoSQL?," 2019 Ivannikov Memorial Workshop (IVMEM), 2019, pp. 7-13, doi: 10.1109/IVMEM.2019.00007.

[12] https://arxiv.org/pdf/2001.02299.pdf