SciPi: A Recommender System for Scientific Publications

Andrew Cachia
Department of Artificial Intelligence
University Of Malta
andrew.cachia.12@um.edu.mt

Joseph Azzopardi
Department of Artificial Intelligence
University Of Malta
joseph.azzopardi.06@um.edu.mt

***Abstract*—This study uses the MAG (Microsoft Academic Graph) dataset to build a recommender system. The system is built using Spark and Neo4j, and runs on Azure using HDinsight and IaaS services. The proposed solution uses the dataset to build a knowledge graph, *SciPi*, using key academic entities to store their information and more importantly the inter-relationships. The recommender system uses the SciPi graph to extract two types of recommendations, including recommending similar papers based on similarity, and also recommending potential collaborations based on strong collaborations clusters.**

I. Introduction

The aim of this project is to create a recommender system dealing with scientific papers and the scientific community. A recommender system is used to aid the users by providing relevent suggestions based on the user's interest. The final product should be able to achieve the following two objectives:

- When given a particular paper Id, the system should be able to give a reasonably accurate result set of the top 5 most similar or relevant papers to that particular paper. These recommendations will be comprised of a number of factors relating to the paper, including the paper's top keywords, the author, institution, and the journal.
- When given a particular author, the system should be able to indicate a number of potential co-authors likely to result in a successful collaboration, and with whom the author has had no collaboration yet.

The chosen dataset from which we will be extracting the required information is the Microsoft Academic Graph (MAG) [3], a comprehensive dataset containing over 219 million scientific papers and over 468GB of data. This represented a major challenge in this assignment. Due to the huge size of the dataset, extracting particular information such as recommendations from it in it's current table-format state would be near impossible using traditional methods. The data is composed of a number of tab delimited text files where each file represent all the data of a particular SQL relationship. The textfiles are of various sizes ranging from a few MB to the largest file being 146GB.

Our aim is to utilize Big Data processing techniques which would allow us to extract all the necessary information within a reasonable time frame, and store the resulting data in an efficient format to allow fast retrieval of these recommendations in the future.

Apart from parsing, extracting the data, and storing it in an efficient format, we also intend to extract underyling information from the dataset through the keywords. The objective here is to analyze each paper's abstract, and utilize the Term-Frequency Inverse-Document Frequency (TFIDF) algorithm to determine the top keywords for each paper. This will allow us to classify the papers based on their keywords, which can then be used for analyses when generating the recommendations.

The rest of the paper is orgnaised as follows. Section 2 provides an introduction to related research on recommender systems, Section 3 provides a detailed description of the methodology and approach used, and Section 4 concludes with a discussion on the conclusions made and possible future work.

II. Related Research

Recommender systems can be split into two common approaches, namely *Content* and *Collaborative Filtering* based approaches [8]. The content-based approach uses features that describe the item, and looks for other items that have *similar* features. In a scientometric context these features can be characteristics describing the paper items and include information such as keywords, authors etc. On the other hand the collaborative filter model is more complex and does not use the characteristics of the items, but instead uses other measures or ratings about the items which are generally attributed to user behaviour and point of view. Recommender systems have gained a lot of popularity in e-commerce and e-business service as they were proven to enhance commercial activity and usability of products [10].

Huang et al [8] developed a recommendation system based on a graph model using a combination of both approaches in a *hybrid* approach for book recommendations. The authors use a two stage approach where at first a similarity measure is computed between the book and customers nodes on their own, then use these weighted similarity relationships together with purchase transactions to build a graph. Another similarity measure is then computed for recommendations in the second stage. Huang et al compare the different recommender approaches and showed that the hybrid approach provided

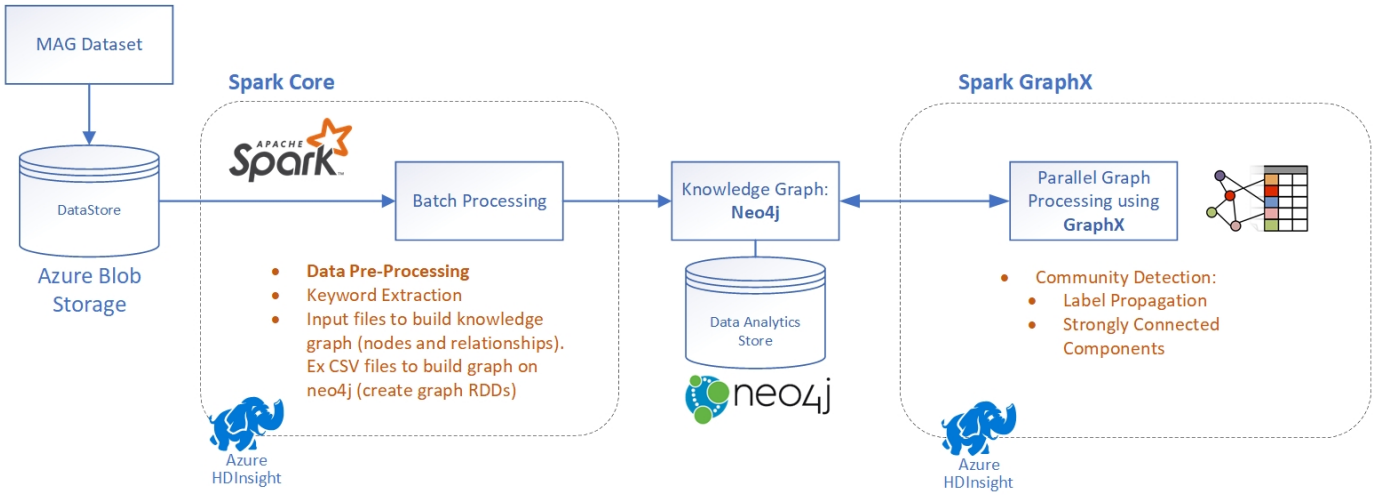


Fig. 1: Spark Architecture used on Azure HDinsight

consistently better results. Hoang et al [1] also analyse academic data and make use of a graph structure to create collaboration recommendations. A weighted directed graph is used to discover potential collaborators by analysing the direct and indirect connections between different scientists.

Arnab et al [3], who are also contributors to the MAG research, have used six different academic entities (authors, papers, and so on) to create the MAS (Microsoft Academic Service) graph to demonstrate how a recommender system can exploit the relationships across all these entities to offer suggestions. Arnab et al try to extract recommendations by using the possible 'missing' connections in the MAS graph that would create a fully connected graph. The authors have also used the MAS entity graph to create an academic search engine based on the Bing engine.

Cung and Jedidi [2], create recommendations for an eCommerce system using a graph database stored in Neo4j, and employ techniques such as the minimum spanning tree or cosine similarity to compute the recommendations.

In our scenario one of the objectives of the recommender system is to find papers that are similar, in order to suggest similar papers to the interested reader given an input paper that the reader finds interesting. This recommendation will be based on content based approach relying on some the features related to papers. Graph theory seems to be a popular choice when building recommender systems [1], [3], [7], [8], [9], and therefore a similar approach is being proposed. The graph contains relationships of salient information related to the papers, as a central component in the graph.

III. Methodology

A. Architecture

The setup to run the SciPi application is illustrated in Figure 1. Apache Spark is the main platform that will be used to construct and analyse the knowledge graph. Spark was chosen for a number of reasons including: i) it is an ecosystem of cluster-computing tools that are integrated with the same platform making it easier to transition across different aspects of the data pipeline, ii) it includes multiple connectivity APIs, iii) support for integration with Neo4j using the *neo4j-spark-connector* and iv) it is widely used, and is well documented. The proposed architecture will be hosted on Microsoft Azure using the HDInsight service. The architecture used for this assignment includes the following components:

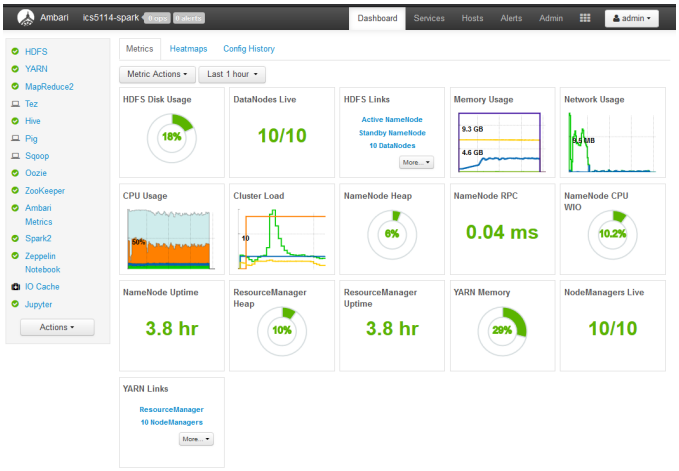


Fig. 2: HDinsight Cluster Dashboard

- **Data Source:** Azure Blob Storage will be used as the main data store as this provides cheap storage and fast data access through Azure. The MAG dataset was pushed directly onto our datastore through Microsoft Academic Services in order to avoid handling (upload/download) of big data components. Blob storage can be accessed directly by the Spark cluster using the Windows Azure Storage Blob (WASB) protocol. WASB is an extension on HDFS which makes the Blob store available to all nodes within the cluster *similar* to HDFS. However, the

storage layer is independent of the Spark nodes which is very convenient and flexible way to persist the data after deleting the Spark cluster, allowing us to save on costs. It is important to note that read/write access from/to Blob storage was very fast and GBs of data could be transferred in a few minutes.

- **Spark Core:** A number of Head and Worker Nodes were used to build the Apache Spark cluster and execute batch processing of very large textfiles to extract the required information. Azure HDinsight provides a flexible way to increase the worker nodes as required for bigger jobs. HDinsight builds the Spark cluster on top of Hadoop HDFS and therefore uses YARN for resource management and scheduling. The MAG dataset is updated approximately every 2 weeks, meaning that the changes are not frequent and more bulky. This scenario does not require any streaming services since the *MAG updates* can be easily handled by the same batch processing job handling the main dataset.
- **Analytical Data Store:** The Neo4j graph database platform was used to store the relationships between the selected components of the graph. The schema for the nodes and relationships extracted is depicted in Figure 3. A graph database provides the advantages to easily query the relations between the academic entities. This is the main reason for chosing a graph database as opposed to other NoSQL or SQL databases. A graph databases was chosen as we are interested in the relationships between different entities, such as relationships between the papers and keywords, conference instances, journals etc. Neo4j is also the central analytic datastore and can provide previously computed results in near real time.
- **GraphX:** Spark was also used to process data from Neo4j in order to faciliate the large amount of nodes and relationship that need to be processed. Therefore the solution leverages the Spark ecosystem and parallelization to execute clustering algorithms on sub-graphs extracted from Neo4j.

The cluster was setup with Spark 2.3 and consisted of 10 worker nodes and 2 head nodes. The machine type $D3V2$ was chosen for the worker node, with each VM having 4 cores and 14G RAM. This machine type is optimised for HDinsight. The HDsight dashboard for the cluster used is illustrated in Figure 2.

The graph schema for the SciPi solution is illustrated in Figure 3, and includes 7 academic entities including papers, authors, and keywords amongst others. We have chosen these entities in order to build a knowledge graph as we believe this approach offers much more flexibility to extract useful scientometric information - for example, most prominent authors or publishing house, timeline views, etc. However for the purposes of this assignment the graph will be used to tackle the objectives listed in section 1. The knowledge graph was built using Neo4j Community version 3.5.5, which was also hosted on Microsoft Azure on same datacentre region. This allowed fast network access between the Spark cluster and Neo4j since it will use the same datacentre backbone connection. The knowledge graph is the central part of the solution where key information extracted from the MAG dataset is stored through relationships of linked nodes. At the centre of the graph are the *paper* nodes that have outlink relationships to various academic entities such as the publishers, keywords, and the authors. The entities used are similar to those used by Arnab et al [3]. The time information is stored as a property in the paper nodes and can be used to filter out the scientific contributions by year. The main connections that will be used for analysis are the paper-to-authors, and the paper-to-keywords relationships, whilst the other nodes are used to enhance the output information.

Neo4j was chosen as the central database for a number of reasons including i) Neo4j provides visulation tools such as the Neo4j Browser that can easily show relationships between authors, or links across different papers ii) Neo4j includes a number of packages for graph mining techniques that can be applied to the sub-graph for direct processing, and iii) Neo4j provides a 'standard' and easy approach to the property graph using *Cypher* query language.

Fig. 3: Graph Schema for Neo4j Knowledge Graph

The data pipeline implemented is summarised in Figure 4 and will be discussed in the next sections.

B. Collecting and Analysing the Data

As illustrated in Figure 4, the first step was that of collecting the data. Although the data was available from the Microsoft website, downloading and storing the data was proving to be a challange due to the huge file size. The traditional approach

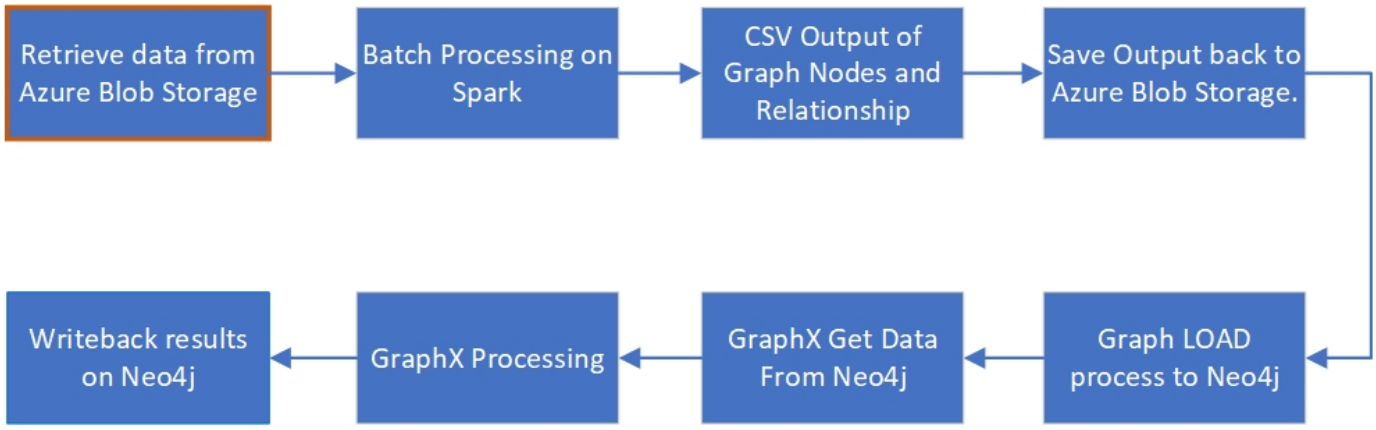


Fig. 4: SciPi Spark pipeline

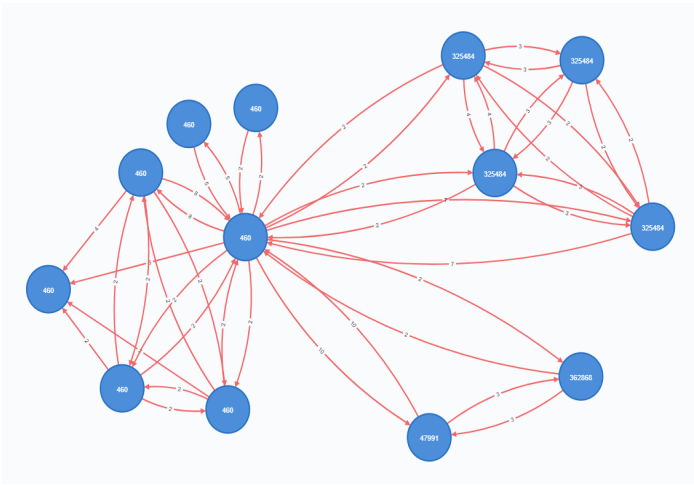


Fig. 5: Author Collaborations

of simply downloading a few files locally and running some experiments was not possible, as the files were batched in chunks of around 20Gb each.

Microsoft however does offer a service where, upon request, they can transfer the files into a blob storage container on your Microsoft Azure account. We therefore opted for this approach. We then utilized the Azure Blob Storage SDK connector in Python to connect to this container, and download smaller chunks of around 500MB of each file, to obtain small samples of the dataset which we could use to investigate the data, and build a development environment on the local machines. The files formats are a mix of tab-delimited and JSON files.

C. Parsing and Transforming the Data

The next step was to now begin parsing the files to extract the data we would like to retain. In order to perform such a large batch processing task, we decided to opt for *PySpark on Apache Spark*. The large benefit that Spark would bring is that it would allow us to create a cluster of multiple nodes to allow parallel processing, and was shown to be significantly faster than other batch processing libraries such as Hadoop [6].

Besides simply parsing and collecting data, we also needed to extract information from this data to allow further analysis and provide better recommendations. To achieve this, we opted to define the top 5 keywords from each paper. We fed the abstract for each paper through our TFIDF algorithm, in order to determine a score for each word. The words with the the top 5 highest TFIDF scores for each paper were selected as the keywords for that paper.

Keyword Extraction The TFIDF algorithm works by determining the frequency with which a word appears in a paper, as opposed to the number of papers in which that word appears. So to summarise:

- Words with a **high** TFIDF score would appear multiple times in one or a few papers, but do not appear in the vast majority of papers.
- Words with a **low** TFIDF score would appear very frequently in multiple papers.

The TFIDF formula is as follows:

$$tfidf = tf \cdot idf \tag{1}$$

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{2}$$

$$idf_i = log(\frac{D}{d_i}) \tag{3}$$

Where n is the number of times a word j appears in document i, or in each document k, D is the total number of documents, and d is the number of documents in which the word i appears.

This technique therefore easily filters out stop words such as 'and', 'then', 'because', and so on. This technique also easily helps pinpoint the important and relevant features of each paper.

Cluster of Authors

The authors are co-authors if they have participated in the same published work. Neo4j does not support bi directional relationships, and therefore two directed edges were created to represent the directional relationship. For a given paper that has n author, nPr co-author relationships are created.

One of the objectives is to find strong collaborations between different authors. In order to emulate a strong relationship, the number of collaborations was computed for every author pair. Therefore the strong collaborations can be filtered by setting a threshold for the minimum number of collaborations that constitute a strong relationship. This filtering process also helps to filter out less important and interesting relationships, say authors that collaborated only once, and also helps to reduce the number of relationships created, and consequently the size of the database. We also think it is a fair assumption that authors that collaborated between each other genuinely know each other. Author relationships and their collaborations (minimum 2) are illustrated in Figure 5. As can be noted in this diagram there are 13 author nodes grouped into 4 groups. The strongly connected groups are clearly visibly with id "460" and "325486".

A similar approach was taken by Newman [9], where the author focused on specific research domain and created a network of scientists, where a link between scientists is present if they had coauthorship in at least one scientific paper. Newman also used a graph model to analyze the author collarborations and calculate a number of statistical values on the networks.

Within Spark, both RDDs and Dataframes were utilized. RDDs were used at the start of the spark jobs to build the RDDs from textfiles, and parse the raw data, however these were then immediately transformed in a *Dataframe* format. This format allowed us to take advantage of the underlying Spark infrastructure, and run queries using the Spark SQL library which would also automatically parallelize data across the cluster. Dataframes are also a distributed collection of data similar to RDDs, however data is organised and structured using named columns which are easier to reference. As highlighted by [6] and [7] the computations based on dataframes versus that of functional RDD can be up to $2-5\times$ times faster. For these reasons most of the batch processing used to build the knowledge graph use the dataframe model.

Spark has the functionality of adding User Defined Functions (UDFs), which are custom SQL functions to allow the developer to write custom functions when the built in Spark functions do not meet the requirement. A UDF function was initially used to compute permutations for the collaborations of authors. Although the UDF functions provide more flexibility they are highly inefficient and slow down spark job executions. When the co-author network computation was changed to a dataframe inner join instead of a UDF permutation function, the efficiency was drastically improved. A spark job submitted on a development workstation using 25MB of raw text was not able to finish in more than 1 hour, or worst failed to complete the job. When changing the UDF function to use Spark inbuild dataframe functions such as inner join, a 500 Mb job was completed in 8.2 minutes (using Ubuntu VM with 2 cores (i5200) and 6G of RAM). The difference in performance stems from the fact that the UDF is seen as a 'black box' function, one that Spark *cannot* optimise since it cannot be converted into native Spark instructions.

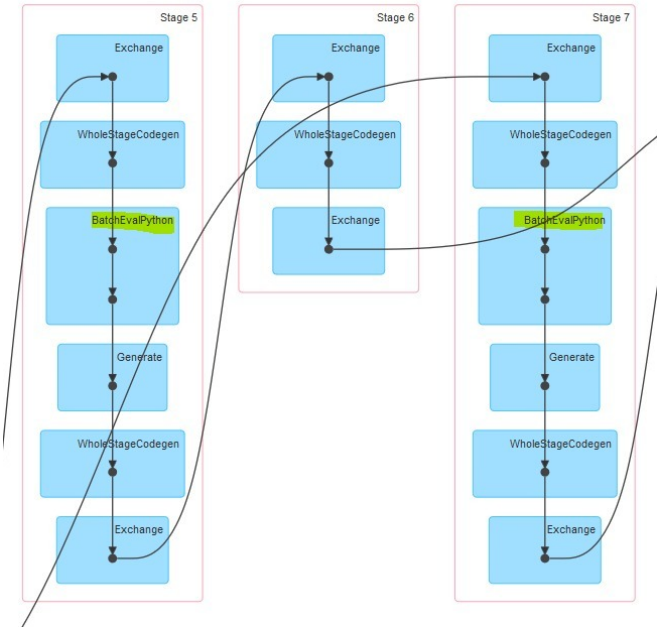


Fig. 6: Spark Stages DAG Visualisations

In another instance UDFs were used as part of the TFIDF to parse the JSON dictionary of keywords and count the number of times the word appears. When running the job in a one node cluster in a development environment there was no significant difference in processing time between the UDF and non-UDF function. However when the UDF function was submitted on the 10-node HDinsight cluster it became a significant bottleneck and the *stage* would have taken several hours to finish. This occured because Spark could not parallelize this task since the function is seen as a 'black box', and it was observed that during processing of this stage only one worker node (out of 10) was being utilised. This was noted as the job stage included an *BatchEvalPython* task as illustrated in Figure 6. The UDF function was rebuilt using Spark built-in function in order to execute job in a timely manner. As it was noted, in most cases there is a way to rewrite the code to avoid using UDFs.

D. Importing data to Graph Database

Once the batch processing was done, the results were saved as CSV files directly in our Azure Blob Storage account as highligthed in pipeline of Figure 4. The output files generated from batch processing had a size of approximately 85GB in total. This includes all nodes and relationships required to build the graph. Once done, another script was run that would

import the files into Neo4J. Cypher was utilized to insert the data into the graph database and create the necessary relationships. The following example demonstrates the use of Cypher to insert a set of keywords from a CSV file and create a relationship to the respective author. Since a very large number of nodes and relationships were created, the Neo4j LOAD-CSV bulk loading tool was used, and records were executed in batches of 1000.

```
USING PERIODIC COMMIT 1000
LOAD CSV FROM csvfile AS line
MERGE (p:Paperid:line[0], name:line[0])
MERGE (k:Keywordname:line[1])
CREATE (p)-[:ContainsTFIDF:line[2]]->(k)
```

Each of the nodes and relationships had their own seperate CSV files. Each node/relationship type had approximately 200 files each of around 70MB per file. In turn each of the files could include millions of nodes or relationships. This proved to be a major challenge to upload all this data in a timely manner. The knowledge graph was built in a containerised environment running on an Azure IaaS VM with allocated 8 CPU cores and 64 GB of RAM. The neo4j container was configured with a $pagecache = 30GB$ and $heap_max = 30GB$ to optimse the memory configuration. The page cache is used for caching graph data after reading from disk, and also to cache indexes, whilst the heap parameter is the memory reserved for query execution.

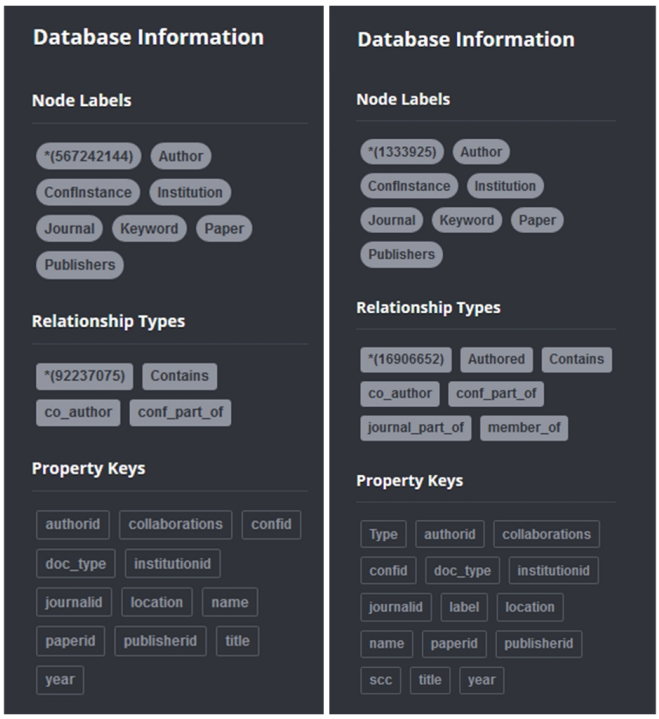


Fig. 7: Partial MAG Database (left), Sample Database (right)

The nodes were uploaded first and the process created around 600 million nodes as shown in Figure 7 (left). Since some of the files were very large (close to 350MB) some manual intervention was required in order to restart or continue the import process. All of the node identities or names were indexed to improve performance. However when trying to create the relationships it was proving to be a very slow process. The relationships were being created successfully however, since each file could contain $3.4M$ records and each record required at least two $MATCH$ queries followed by a $MERGE$, it proved to be a very slow process where it could take more than 10 hours to finish one file. The 'slowness' is also attributed to the large number of nodes that were already created in the database, since a large number of indexes were created. This was not experienced when creating the knowledge graph using samples of the MAG dataset in the test environment. Therefore due to the vast amount of data and limited resources available it was not possible to finalize the creation of the whole knowledge graph. The Neo4j database can support billions of nodes so we still believe that it was the best analytical data store for this requirement. The cost of the used Azure resources was also signficant, and therefore it was difficult to scale up the Neo4j resource. The large number of nodes creates another difficult to query or perform computations on the graph. These queries could take a lot of time and consume all available 30GB or heap memory and would eventually stall the Neo4j application.

A smaller version of the knowledge graph (Figure 7 right) was built in order to extract results . The neo4j import scripts were modified to load $100,000$ papers only, together with all the relationships depicted in Figure 3. Although limited in content, this graph was then used to extract recommendations and collaborations.

E. Generating Recommendations

Relevant Papers Recommendation: Once the data was loaded into the graph database with all the respective nodes connected, we could begin to generate recommendations. The algorithm of choice was the **Jaccard Similarity Algorithm** (also referred to as the *Jaccard Index*), which determines how similar two sets are based on their overlapping features. The aim of this system is that, when given a particular paper id, the query should return a list of papers with the highest Jaccard Index, signifying those papers as being very similar and most relevant to the given paper. The Jaccard similarity was chosen as it is used to find the similarity between two sets, whereas other similar measures such as *Cosine Similarity* use the magnitude of the relationships to find similarity. The Jaccard similarity between the paper-keyword relationships of two papers can be found using (III-E), where A represents the set of keyword that paper1 is related to, and similarly B represent the set of keyword that the second paper includes. The Jaccard similarity compares the overlap between the sets of connected nodes A and B.

$$Jaccard\,Similarity = \frac{A \cap B}{A \cup B} \tag{4}$$

To generate the Jaccard Score, the query will compare the number of similar keywords and authors the different papers contain, as well as whether they belong to the same journal or institution.

The resulting Cypher query to generate the recommendations is as follows:

```
MATCH (p1:Paper paperid: '<paperid>')-[]->(n)
WITH p1, collect(id(n)) AS p1ConnectedNodeIds
MATCH (p2:Paper)-[]->(n) WHERE p1 <> p2
MATCH (p2)-[]->(n2)
WITH p1, p1ConnectedNodeIds, p2, collect(id(n2)) AS
p2ConnectedNodeIds
RETURN p1.paperid AS from,
p2.paperid AS to,
algo.similarity.jaccard(p1ConnectedNodeIds,
p2ConnectedNodeIds) AS similarity
ORDER BY similarity DESC
LIMIT 5
```

After running the query for a particular paper id, the top 5 most similar nodes would be displayed as follows:

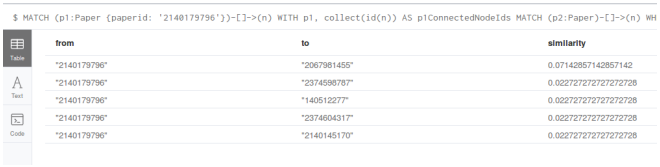


Fig. 8: Recommendation results for paper id '2140179796'

When taking the top recommendation and analysing the nodes, we can tell that the two papers are indeed strongly connected, as shown in Figure 9. In fact, it seems as though one paper is a follow up to the other.

Author Collaboration Recommendations: Once the data was pre-processed and loaded to Neo4j, the data can be easily queried to extract subgraphs from the knowledge graphs and in turn perform graph processing to find other paper recommendations using graphX.

The data can be analysed by using Neo4j directly, however since we are using a single instance of Neo4j, and scalability is limited, it would take a lot of time and resources to process such a large number of nodes and relationships. Therefore the role of Neo4j is limited to extract the data using Cypher queries, whose results are used as input to graphX job.

The data can be loaded into GraphX using the Neo4j-Spark-Connector. Although there is a lot of feature overlap across the APIs, there are still some limitations in python as the Neo4j-Spark-Connector is only currently available in Scala. The connector can be added to a Spark cluster by adding an additional jar in the jars repository. The connector is enabled on the Spark context and enables the use of the Neo4j Bolt protocol to transfer graph data to and from the Neo4j servers.

Therefore the subgraph for the co-author relationship can be queried and loaded directly during a spark job. Once the data is

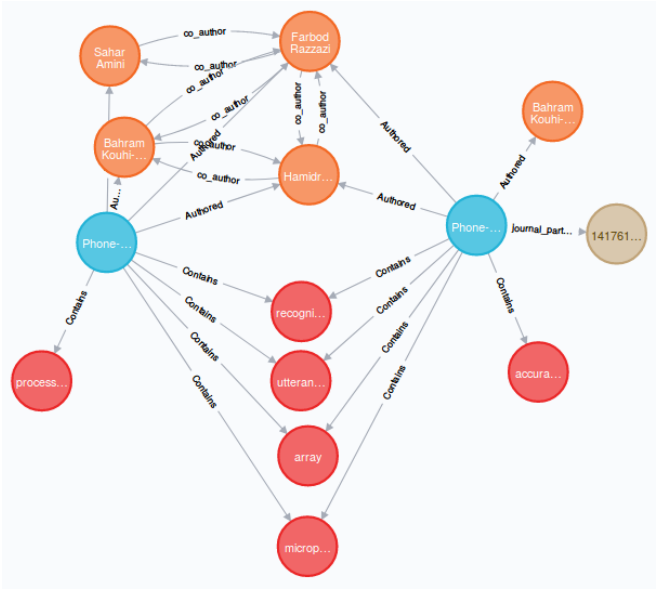


Fig. 9: Top Recommended Paper for paper id '2140179796'. Paper nodes are in blue, keywords in red, authors in orange and journal in yellow.

loaded into a graphX instance, created from the loaded Neo4j data, the graphX object is used to process the data in parallel streams across the Spark cluster. Once the data is processed the Neo4j connector enables the user to push data back to Neo4j. However in the current release of the connector (2.1.0-M4), only one property can be written back to each vertex in the sub-graph at the same time. This is not a major limitation as in most cases one parameter is only required to be pushed back such as the *rank* in PageRank, or *label* groups after LabelPropagation algorithm.

In the last stages of the pipeline, a subgraph of the authors was loaded into graphX and the community detection algorithms were used for unsupervised clustering. The community detection algorithms are computed using the graphX Pregel variant API, since a vertex-centric approach is better suited for these iterative algorithms. The subgraph was imported to GraphX through the neo4j-spark-connector using a cypher like query that queries all authors and co_author relationships. The following two algorithm were considered to tackle this problem; i) Label Propagation (LP), and ii) Strongly Connected Components (SCC).

Label Propagation (LP): The label propagation method was proposed by Raghavan [5] as a community detection technique. A community can be dened as a group of nodes that are similar to each other with characteristics that isolate them from the other group of nodes. The SciPi graph includes the collaborations done by vast number of authors. Therefore a community in this aspect would represent a set of authors that have worked together. The co_author relationships hold the number of collaborations as a property within the relationship and therefore weaker connections can be filtered out by setting

a *collaboration threshold* to a higher value. In our example a single collaborations were not created in Neo4j and therefore the collarboration threshold was set to 2.

The label propagation algorithm is initialised by assigning a random label number to each node in the graph. The labels are propagated to neighboring nodes at each super-step and each node adopts the label that occurs for most times on its neighbouring nodes. The process of matching the label is iterated over a number of times and as the labels propagate in the network, nodes form clusters by reaching an agreement on a unique label. Label Propagation is an unsupervised technique as the number of communities and their sizes are not know from before. Although LP is not a very expensive computation, its convergence may not be guaranteed or it can result in uninformative results such as having single community. For our implementation we have used 20 iterations.

Strongly Connected Components (SCC): The SCC algorithm can be used to find subgraphs where every vertex of this subgraph is connected to all other vertices within the subgraph. In order for the sub-graph to be strongly connected, a particular node within the subgraph must have a path to reach all the nodes in the subgraph. The SCC can therefore be used to find communities of authors that are strongly connected. The SCC algorithm was also computed in graphX by loading the data from the Neo4j database. The SCC algorithm is commonly used in social networks to find people with the same interests (likes). Similarly in this scenario authors are linked (bi-directional) with collaborations.

In order to find and visualise different communities, Neo4j will be used to query a particular researcher. After the LP or SCC is computed the author node is assigned a particular label, or group. After finding the respecting label or group of the author of interest, the database can be queried to provide a list of all the authors have the same label/group. This approach has the advantage that the communities are determined in a batch process, and once the process is executed the information can be easily queried directly from Neo4j in real time without the requirement of running additional Spark jobs.

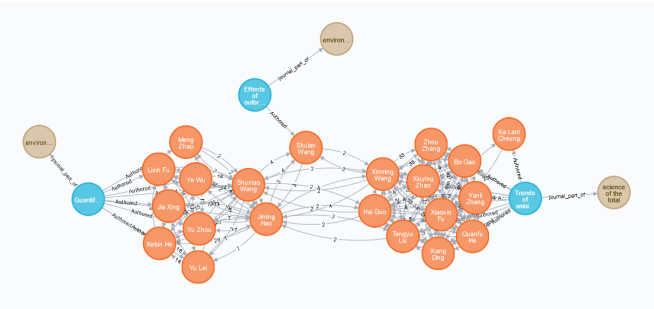


Fig. 10: Collaboration Network example: Authors-Papers-Journal

```
$ MATCH (a:Author) RETURN a.scc, COUNT(a.scc) as MemberSize ORDER BY MemberSize DESC
```

a.scc	MemberSize
100017	138704
100168	32
352523	26
126152	21
221021	21
106782	20
126917	17
177256	17
104326	16
105492	16
112360	16
121216	16
161316	16
186786	16
230866	16
289991	16

Started streaming 109347 records after 930 ms and completed after 1048 ms, displaying first 1000 rows.

(a) SCC Groups

```
$ MATCH (a:Author) RETURN a.label, COUNT(a.label) as MemberSize ORDER BY MemberSize DESC
```

a.label	MemberSize
233374	43529
224780	6873
140171	5294
323716	2065
284439	1772
153590	548
214194	510
316089	506
234877	494
220524	450
313521	382
179086	367
220850	334
337154	306
132088	299
139018	280

Started streaming 135235 records after 1085 ms and completed after 1238 ms, displaying first 1000 rows.

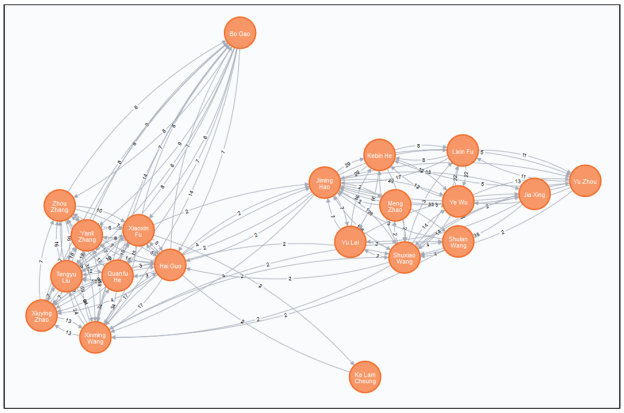
(b) LB Groups

Fig. 11: Clustering Groups

Figure 10 below shows an example of a collaboration network having the same LB label. The actual values for collaborations were computed using all the MAG dataset, however since a sample of the papers was taken as described earlier, only a small number of papers are shown for this cluster.

The grouping results from SCC do not fair as good as the grouping results from LB. Figure 11a shows that SCC has grouped most of the nodes into one big cluster, whilst the LB groupings have more group segregations. The best value for LB iterations, i, for the network would need to be identified by computing the LB algorithm for different values of i.

In order to find possible collaborators, Cypher queries can be used to find the all other authors that have the same label or group as the given author. Once the list is filtered from the Author nodes, the possible collaborators includes those authors who the given author has never collaborated with. The indegree for the authors in the group is computed so that the most popular authors in the group are returned as potential collaborators. For the group shown in Figure 12a, the author *Bo Gao* is considered. The authors on the left-hand side are



(a) Authors Group Visualisation

```
$ MATCH (b:Author {authorid: "2566047383"}) WITH b.label AS group MATCH (a:Author {label: group}) WHERE
```

a.name	Collaborations_in_Group
"Jiming Hao"	26
"Shuxiao Wang"	24
"Kebin He"	21
"Congrong He"	10
"Ye Wu"	8

(b) Top 5 Recommend collaborators

Fig. 12: Results for author in group *"140171"*

already collaborators of Bo Gao, whilst the authors on the right are all his possible collaborators. The indegree filters out the least collaborative authors so that the top collaborative authors can be returned. The top 5 recommendations are shown in Figure 12b.

Dynamics in Publication Networks: To track the dynamics in publications networks the collaborations Cypher query is extended to find out the journal or conference preferences of the groups of authors. After filtering all the authors in the group, all the papers and their conference/journal relationships are matched, and the results are collected and aggregated to show the year, event, and papers submitted as shown in Figure 13. Since the results were computed with 100k papers only a limited number of papers from the authors group were available. The queries used can be reference the *Supplementary Material* document for this paper.

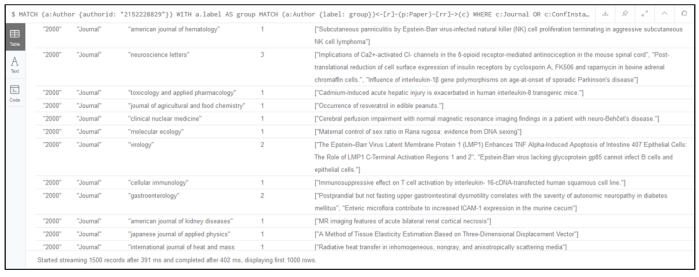


Fig. 13: Timeline

IV. Conclusion and Future Work

As noted during the implementation of *SciPi*, the approach when handling large volumes of data can become complex quite fast and traditional methods can start failing quickly. Spark turned out to be very effective in processing the large MAG dataset and extracting the relevant information to create a basic recommender system. Its parallelisation allowed efficient processing of the huge quantity of data.

Although the complete Neo4j knowledge graph was too big to host on our limited resources, we still believe that Neo4j is one of the best options for an analytic store for SciPi. Storing and querying connections between the different nodes turned out to be very effective. The main challenges arose due to the relatively slow processing times when attempting to load huge volumes of data. Given adequate resources, using the Neo4j enterprise edition and running Neo4j across a cluster would have most probably solved a number of our issues.

Although our architecture did not make use of the HDFS local storage, Azure Blob Storage was found to be a great alternative as it provided very fast read and write access whilst also being a cheap and persistent storage for the Spark cluster. As blob storage runs on the cloud, having all the files available from anywhere also proved to be very convenient. Blob storage was definitely not a bottleneck in the SciPi pipeline.

Containerization of all our system through docker also turned out to be very useful, allowing us to port the system wherever it needed to be run with minimal effort. This helped the experimentation process greatly.

Future work would include implementing improvements to the recommender systems. The co-author recommendations could be enhanced using an algorithm that includes additional parameters in the clustering process, such as including the number of collaborations as a weight value. The other entities in the knowledge graph could also be included when generating clusters to improve the clustering ability of the authors.

It is difficult to implement a hybrid or collaborative recommender system without a user database and a system to collect user feedback on which you can base recommendations. This would be one of the possible way to enhance the recommender system in future work.

References

[1] Hoang, D.T., Nguyen, N.T., Tran, V.C. and Hwang, D., 2018. Research collaboration model in academic social networks. Enterprise Information Systems, pp.1-23.

[2] H. Cung and M. Jedidi, "Implementing a Recommender system with graph database", Freiburg University

[3] Sinha, A., Shen, Z., Song, Y., Ma, H., Eide, D., Hsu, B.J.P. and Wang, K., 2015, May. "An overview of microsoft academic service (MAS) and applications." In Proceedings of the 24th international conference on world wide web (pp. 243-246). ACM.

[4] Neo4j-Spark-Connector, (2019) https://github.com/neo4j-contrib/neo4j-spark-connector [Accessed 1 June 2019].

[5] Raghavan, U.N., Albert, R. and Kumara, S., 2007. Near linear time algorithm to detect community structures in large-scale networks. Physical review E, 76(3), p.036106

[6] Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R. and Zaharia, M., 2015. Scaling spark in the real world: performance and usability. Proceedings of the VLDB Endowment, 8(12), pp.1840-1843.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). ACM

[8] Huang, Z., Chung, W., Ong, T.H. and Chen, H., 2002, July. A graph-based recommender system for digital library. In Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries (pp. 65-73). ACM.
[9] Newman, M.E., 2001. Scientific collaboration networks. I. Network construction and fundamental results. Physical review E, 64(1), p.016131.
[10] Ricci, F., Rokach, L. and Shapira, B., 2011. Introduction to recommender systems handbook. In Recommender systems handbook (pp. 1-35). Springer, Boston, MA.