Distributed Graph Processing on the Open Academic Graph Dataset

Han-Chiat, Wong
University of Illinois
at Urbana-Champaign
Illinios, USA
hcwong2@illinois.edu

Manas Kumar, Mukherjee
University of Illinois
at Urbana-Champaign
Illinios, USA
manaskm2@illinois.edu

Paco, Cruz Morales
University of Illinois
at Urbana-Champaign
Illinios, USA
fmc2@illinois.edu

Sudheer Kumar, Kusuma
University of Illinois
at Urbana-Champaign
Illinios, USA
skusuma3@illinois.edu

***Abstract*—The Open Academic Graph dataset is one of the largest academic graphs linking Microsoft Academic Graph (MAG) and AMiner. Using the Open Academic Graph (OAG), one can then perform expanded searches on research publications using meta-data such as keywords, authors and synopsis. The OAG consists of 3 major datasets: Linking Relations, AMiner and MAG. The 2017 version of the dataset is over 143 GBs. Extracting and processing these datasets to construct publication graph via traditional ETL methods is inefficient. Our effort is to construct an effective graph processing pipeline using the latest distributed processing technologies such as Spark and Neo4j.**

Keywords—distributed, graph, Kafka, Spark, Spark Streaming, Neo4j.

I. INTRODUCTION

Cross referencing and citation research are key parts of research publication activities in the academia research community. Often, researchers review and cross reference papers by topics and authors. Over the years, there have been various systems such as Google Scholar and ResearchGate providing online searching of research publications. However, these systems are not comprehensive and effective as they lack functionalities allowing researchers to easily relate topics and authors of the same semantic subjects.

The Open Academic Graph (OAG) dataset was created as the result of the partnership between Microsoft and AMiner. The goal of the partnership is to create a comprehensive academic graph for studying citation network and topic content. The 2017 version of the dataset contains publication meta such as authors and references and many more attributes for over 320 million papers from MAG and AMiner. It also provides a linkage meta dataset contains over 64 million cross-links between MAG and AMiner. The entire dataset is over 143 GB and consists of 3 major components: MAG, AMiner and Linkage. Extracting and processing the entire dataset to build a searchable graph solution using the traditional procedural ETL and aggregation framework is inefficient and time-consuming.

In this paper, we try to address the challenges of extracting, loading and building a searchable academia publication graph system by leveraging the OAG dataset. Since OAG dataset is huge, we choose to use the subset of data to help understand and project the solution backed with our experimental results. The data files can be found in https://aminer.org/open-academic-graph.

II. SCOPE

A. Background

Microsoft Academic Graph (MAG) and AMiner started effort in consolidating publications based on profiles of researchers from various sources with author accuracy of 97.41%.

B. Current State

As detailed in the Introduction section, OAG dataset has become so huge to implement with traditional ETL techniques and opened avenues to solve using big data technologies enabling data processing for further analysis and deduce insights with the limited resources. Each publication contains meta information like author, references, keywords, year of publication etc.

C. Our Contribution

In this paper, we are introducing avenues to process huge dataset into chunk partitioning in distributed and parallel data pipeline to enable easy query and data visualization.

D. Use cases

For academic researchers, the availability of an easy to query graph database will enable to assess the existing corpus of research around the topic of interest, allowing them to navigate through existing relevant research and reducing the risk of inadvertent duplication or inadequate citations.

The proposed system will help journalists, peer reviewers and authorities to track down research that has been tainted by unethical practices like self-references, plagiarism, results manipulation, faked experimental results etc.

Following are User Scenarios producing directed graph representing with nodes and edges with required properties and tabular format of results where suitable.

1. Author Based:
   a. Get the k latest Publications from specific author based on year of publication
   b. Get the popular (more citations) publications from specific author
2. Publication (Paper) Based:
   a. Get popular publication (most referenced)
   b. Get k latest publications based on search criteria(keywords)

E. *Future Extension*

The following are the potential future work we may embark after our initial effort:

1. Extracting each paper, preparing topics using data mining techniques and showcase relevant and latest publications based on search criteria.
2. The relationships uncovered by a graph database can be used by a recommendation engine to ensure that researchers do not waste time trying to find relevant existing papers as background for their research.
3. Provide UX interface for querying the Graph database to retrieve various user case scenarios and showcasing results in terms of Graph Nodes and Tabular data format for easy consumption.
4. Instead of using py2neo driver, we should leverage Spark GraphFrame to process and build vertexes and edges to optimize graph processing.
5. Instead of using py2neo driver, we should use neo4j-spark connector to optimize the performance of Spark integration with Neo4j.
6. Extend graph storage to a scalable cloud solution like Azure Cosmos DB which is a multi-model NoSQL store that supports graph data.

III. Proposed Solution

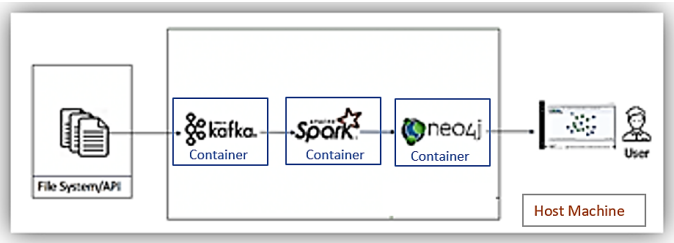
Proposed solution consists of 3 major areas in building the pipeline

Stage1: We'll split the huge datasets into smaller chunks and push through Kafka Queue or read directly from persisting datastore.

Stage2: Processing using Spark-streaming and building aggregating, prepare vertices and edges to the persistence storage.

Stage3: Consume data from persistence storage, build Graph network using Neo4j Graph database and Neo4j browser to visualize graph data.

A. *Workflow*



B. *Technologies*

Following are the potential technologies we are exploring for building the proposed solution:
Programming Language**:** Java, Python
Data ingestion**:** JSON loader, Kafka [2]
Data processing**:** Spark Streaming, Kafka consumer
Data visualization: Neo4j browser
Data persistence: Neo4j Graph Database

IV. Solution Explorations

The canonical workflow shown in Section III A(workflow) served as the basis of the end to end scenario of a minimum viable product (MVP) that we will construct. With that in mind, the success criteria for the MVP is as follows:

a) The system shall ingest over 1 GB of AMiner data concurrently from an on-premise file system into a Massively Parrallel Processing (MPP) pipeline.

b) The pipeline shall consist of micro atomic modules to parse, transform and persist the data into the persistence storage for graph analysis.

c) End users must be able to then exercise simple graph queries, for example, to find related refences for a specific paper based on a topic. The result network graph should be displayed in a web browser.

The decision is to leverage Kafka as the ingestion source since Kafka is designed to handle high throughout message streams. The goal here is to have Kafka producer(source) serves as the source pushing rows of AMiner record into the queue. Each AMiner record is atomic and of JSON format. The consumer(sink) would then spawn Spark jobs to consume the records in queue to the processing. Spark jobs would parse and transform records into the desired graph data format and then store the data into Neo4j, a native graph database. As each record is atomic, we can spawn many Kafka broker instance and Spark jobs to process the data. Neo4j provides a web based interactive query interface that the end user can use craft queries, and the network graph results are displayed in the web interface.

A. *Cloud PaaS vs Containers*

As discussed above, the goal is to build an MPP pipeline to perform concurrent processing of atomic data streams. This notion would be that we need the ability to scale up compute and storage as needed as processing jobs and data volume grow. The bottleneck for compute would be at a) Kafka broker to handle the ingestion b) Spark worker to process the data. On the data storage side, the bottlenecks exist in c) I/O to the write the data to the persistent storage and d) data partitions as data grows. To address the bottlenecks, we investigated several cloud PaaS solutions as these solutions are inherently built with scale in mind. For Kafka, we investigated the AWS Managed Streaming for Kafka (MSK) and Apache Kafka on Microsoft Azure HDInsight. Both PaaS solutions are fundamentally similar with the exception that AWS MSK is in public preview with limited service region footprint while Azure Implementation of Kafka on HDInsight [2] is much more mature from both user experience and programmability perspective in addition to having a more comprehensive service region coverage. Note that HDInsight is the Hadoop PaaS on Azure. Microsoft also offers a different event streaming with Kafka via Azure Event Hubs for Kafka. AWS MSK [3] and Kafka on Azure both required the creation of Kafka clusters in a Virtual Private Network which is essentially to enforce proper

security boundary for authorized clients to communicate with the clusters. As for Spark PaaS, both AWS and Microsoft Azure offer Apache Spark PaaS: Apache Spark on AWS EMR and Apache Spark on Azure HDInsight. These two PaaS solutions are similar, but Microsoft stands out with its Spark offerings by offering Azure Databricks via partnership with Databricks. Azure Databricks[1] offers a unified analytics platform with optimized Apache Spark environment and seamless integration with a variety of services on Azure including SQL Data Warehouse, Cosmos DB and HDInsight. As for DBaaS (Database as a Service), both AWS and Microsoft Azure offer a variety of managed services. However, we found that the Microsoft Azure Cosmos DB to be an attractive service as it is a multi-model database service that supports a variety of data format including JSON and Graph. It also provides multi-region and multi-master reads/writes with 5 different tunable consistency levels. Now, we realized that PaaS services, especially fully-managed services would suffice the scale requirements and resolve the bottlenecks mentioned above, but these services have a high entry cost particularly in the development and testing phases.

Container [4] is a less cost prohibitive approach for development and testing phases. The incremental cost of creating and spinning up docker containers with Apache Kafka [5], Spark [7] and Neo4j [8] for development testing purposes is very minimal (given human capital and system resources stay constant). Given Microsoft Azure and AWS both offer Kubernetes and Container cloud solutions, building our solution using containers would allow our system to scale from on-premise to cloud much more robustly. Hence, given the benefit of low entry cost and portability, the minimum viable solution (MVP) will a container solution with Apache Kafka, Spark and Neo4j.

B. Tiers

a. Producer Client

AMiner dataset consists of several text files with JSON data. In the current implementation, these files are split into smaller chunks with fixed number of lines (1000 each). There is a python client program which reads these JSON files in sequence and push it to a Kafka topic. In the future implementation, the python program might use more advanced techniques like 'Kafka Streams' to process the batch and real time data feeds in a uniform way.

b. Kafka

Kafka and Zookeeper instances are part of the composite docker image. Zookeeper is configured to use port 2181 and the Kafka broker is configured to use port 9092. The producer component connects with the Kafka topic using the following broker via HTTP, for example http://broker:9092.

c. Spark Streaming (Consumer, Persistence)

Kafka Consumer is meant to do the following:

- Consume data from Kafka published topic
- Parse the data and retrieve only the required info
- Generate required Nodes(&properties), Edges(&properties)
- Persist the processed data into Neo4j Graph Database

d. Neo4j – Graph Database

Neo4j is an open-source graph database, implemented in Java described as embedded, disk-based, fully transactional Java persistence engine that stores data structured in graphs rather than in tables. We create the vertices and edges as follows:

- Vertices (nodes): Paper, Author, Keywords
- Relationship(edges): AUTHORED, REFERENCE, CONTAINS

Author -----AUTHORED ----> Paper
Paper -----REFERENCE ----> Paper
Paper -----CONTAINS ----> Keyword

The following is an example of a research paper meta in JSON.

{"id": "53e99837b7602d970205d322", "title": "Notes on approximation", "authors": [{"name": "G. G. Lorentz"}], "venue": "Journal of Approximation Theory", "year": 1989, "page_start": "360", "page_end": "365", "lang": "en", "volume": "56", "issue": "3", "doi": "10.1016/0021-9045(89)90125-1", "url": ["http://dx.doi.org/10.1016/0021-9045(89)90125-1"], "references": ["53e99b56b7602d97023fd8cd"], "keywords": ["computational modeling", "space technology", "computer networks"]}

Each JSON record comes with a unique id for the research paper of publication along with the title, year, language, the list of authors, references and keywords.

e. Data Visualization:

Once processed data is stored in Neo4j Graph database, it can be consumed through various visualization technologies like D3js, Tableau, PowerBI etc. Neo4j supports Cypher Query Language [9] which returns the query results in JSON format that can be further transformed to appropriate format for display in chosen visualization tool.

As part of the MVP, we decided to leverage Neo4j default interactive portal to showcase the query capability and the output in tabular and Graph structure. The Neo4j interactive portal can be accessed via a designated http port, for example, http://localhost:7474/browser

V. Assumptions

As noted prior section, we will containerize Kafka, Spark and Neo4j. The assumption is that the containers will be portable to across machines and cloud providers. The following are the key assumptions:

1. docker-compose [7] will be used to instantiate the containers and the setting the docker-compose.yml file will ensure the containers in individual local environment and cloud environment without the need to reconfigure the

settings such as Network Lan and port settings for Kafka and Spark.

2. User credentials for Neo4j will remain the same for all components to communicate with Neo4j regardless to the environment the containers are running.
3. The py2neo driver used to transform and load data into Neo4j will perform consistently with multiple instances of client consumers.

VI. EXPERIMENTION RESULTS

A. Data Loading and Processing Performance

The performance test on data loading and processing performance was performed on an Ubuntu 18.04 LTS VM with 6 vCPUs, 12 GB RAM and 100 GB storage. The boot, swap and data volumes were mounted entirely on an Intel 600p nvme SSD in the host machine. This setup was to ensure we would get consistent disk IO performance and to reduce bottleneck from disk IO.

The performance test consisted of two parts: 1) baseline test of 5,000 rows of records and 2) final test of 3GB of data consisting 1,000,000 rows publication records. The test environment composed of the following containers:

a. 1-node Kafka zookeeper
b. 1-node Kafka worker
c. 1-node Spark master
d. 1-node Spark worker
e. 1 Neo4j database instance

a. Baseline test: 5000 rows of records

The test procedures were as follows:

i. Kafka Producer:

- Producer code read in the test JSON file consisting 5000 AMiner publication records.
- The producer routine established a connection with Kafka and created a message queue topic, "aminer".
- The code proceeded to push each record to the Kafka queue.

From host machine: python producer.py

ii. Spark Consumer:

- The spark worker was set up with the default settings, i.e. one default executor. Below is the spark-submit invocation of the consumer worker:

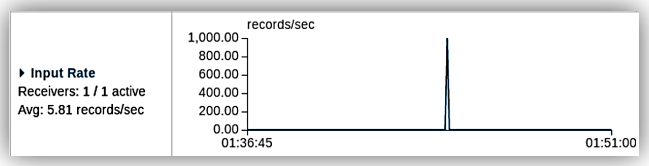
docker exec spark-master bin/spark-submit --verbose --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.3.1 --master spark://spark-master:7077 --executor-memory 1g --num-executors 2 --executor-cores 1 --total-executor-cores 2 /opt/spark/code/consumerSpark2.py

- Spark consumer routine established the connection with the Kafka queue with a 5s fetch window. This means that Spark would fetch data from the queue every 5s and the data fetch would be the batch pushed in to the process queue.
- Spark processed each batch in the queue. The Consumer code processed in the JSON row and passed the JSON record the data processor.

iii. Data Processor (Neo4j):

- Data processor parsed the JSON record and created vertexes and edges.
- Data processor upload the processed data into the Neo4j.

Kafka to Spark Ingestion result:

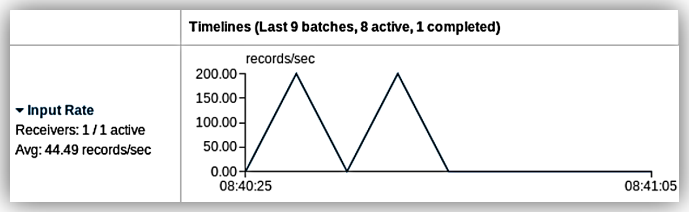


As shown above, the spike in the chart indicates Spark ingested the data from Kafka. The avg ingestion rate averaged 5.81 records/sec. The ingestion rate of 5000 records was under 1 minute. The entire processing including job queue time, transformation and loading of data into Neo4j took about 17 mins. This means that, from end-to-end, the system would process 294 of record per minute. Below shows the processing statistics from the Spark job.

Streaming Statistics
Running batches of **5 seconds** for **17 minutes 6 seconds** since **2019/04/28 01:36:43** (**206** completed batches, **5000** records)

We ran the same workload several times which yielded an average processing time of 15 minutes. Now, this would mean that, with the current design and environment, it would take about 3000 minutes, i.e. 50 hours to load 1,000,000 rows of data. We questioned if there were options to increase performance of the entire process. With the baseline time, out goal was to halve the processing time, i.e. 7-8 mins for 5000 rows of records.

It was evident that the processing bottleneck was surrounding Spark batch processing and Data Processor given that the Kafka-Spark ingestion time was sub-minute. We understood that Spark Streaming was fundamentally batch processing in series. We decided to increase the number of executors to increase task parallelism and thus reducing the overall processing time. For this round of testing, we set the number executors to 2 and manually forced the batch size to be 1000. Below shows the ingestion time averaged 44.49 records/sec.



The processing result per batch improved with the total end-to-end processing time of about 30 seconds per 1000 records as shown the streaming statistics below.

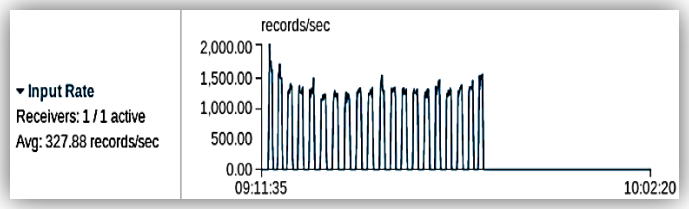
Streaming Statistics
Running batches of **5 seconds** for **30 seconds 761 ms** since **2019/04/28 08:40:19** (**1** completed batches, **1001** records)

Given this result, we expected the final test with 1,000,000 rows of records would to take about 500 minutes or 8 hours of total processing time.

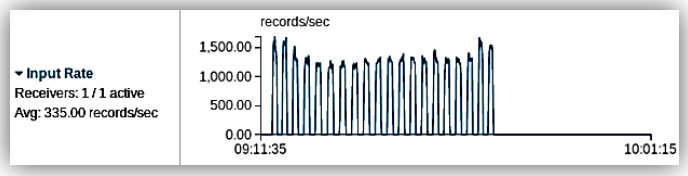
b. Final test: 1,000,000 rows of records

The test procedures were like the baseline test with the exception that we decided to spin up two instances of the Spark Consumer to parallelize the data loading process. The Producer module was updated to send a 50k batch to 2 topics, 'aminer0' and 'aminer1' in a round-robin manner to the Kafka queue. The Spark Consumer instances were configured with 2 executors each and specifically set to fetch the data from the respective topics. We expected having two Customer instances running would halve the estimated total processing time to 4 hours for 1,000,000 rows of data.

Below shows the Kafka to Spark ingestion time for Spark Consumer instance #1. The ingestion rate averaged 327.88 records/sec.



Below shows the Kafka to Spark ingestion time for Spark Consumer instance #2. The ingestion rate averaged 335.00 records/sec.



The total ingestion time for 1,000,000 rows of data was about 25 minutes. However, we observed that the processing time was not as performant. The processing time increased dramatically as Spark continued to process the batches in the stream. Below shows the processing time of some of the completed batches for the respective instances.

Processing time for completed batches for Instance #1:

Batch Time	Records	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)
2019/04/28 09:12:50	8964 records	4.8 h	4.0 h	8.9 h
2019/04/28 09:12:45	8562 records	2.0 h	2.8 h	4.8 h
2019/04/28 09:12:40	10335 records	13 min	1.8 h	2.0 h
2019/04/28 09:12:35	3974 records	0 ms	13 min	13 min

Streaming Statistics
Running batches of **5 seconds** for **15 hours 54 minutes 42 seconds** since **2019/04/28 09:11:31** (**17** completed batches, **48095** records)

Processing time for completed batches for Instance #2:

Batch Time	Records	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)
2019/04/28 09:13:20	7589 records	2.9 h	3.1 h	6.0 h
2019/04/28 09:13:15	8073 records	39 min	2.2 h	2.9 h
2019/04/28 09:13:10	5626 records	1 ms	39 min	39 min

Streaming Statistics
Running batches of **5 seconds** for **15 hours 58 minutes 26 seconds** since **2019/04/28 09:11:34** (**23** completed batches, **36930** records)

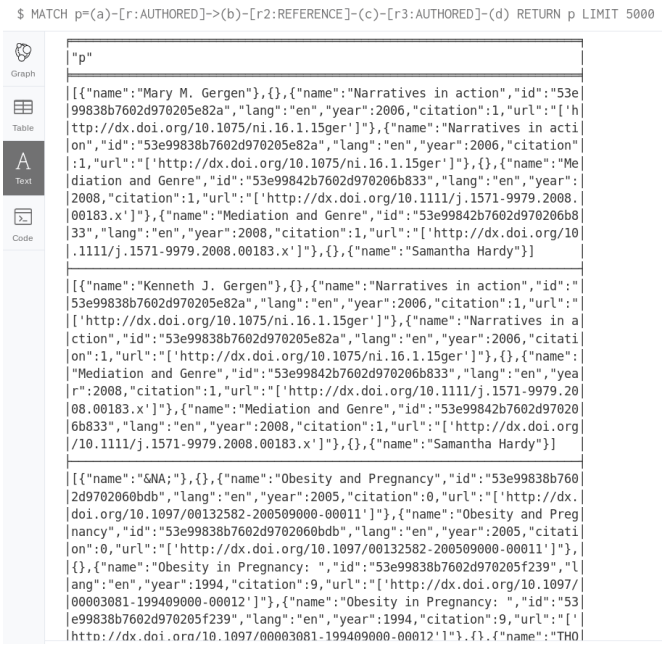
It seemed that after about 8 hours, the system only transformed and loaded 53,123 rows of records. This was far below the expected results as per the baseline test. The processing time continued to decay as time pressed on. After 16 hours, the system processed a total of 85,025 records, far below the expected result. Further investigation indicated the data processing and loading module via py2neo driver were not fully leveraging Spark parallelism. Each row of record was essentially processed in series. Furthermore, it seemed that the ingestion rate to Neo4j via the py2neo driver degraded as data volume increased.

B. Graph network results from AMiner dataset

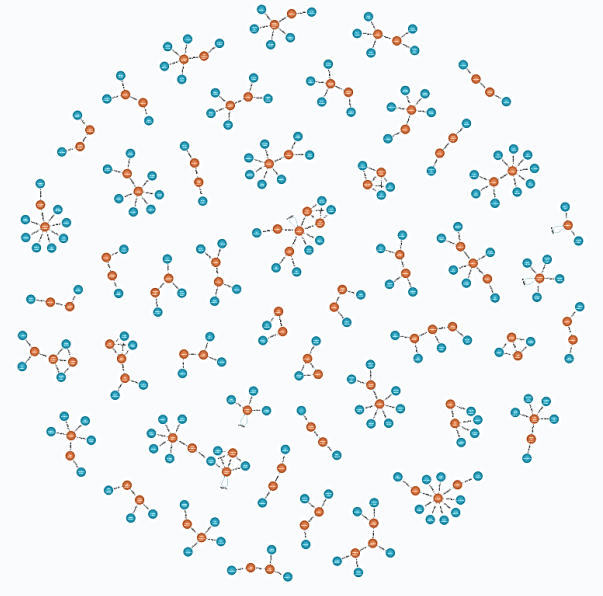
a. Hierarchy relationship

One of the key scenarios that we have established is the ability to query data with hierarchical relationship. For example, we would like to find the authors and their publications along with the referral publications and authors. The query below is an example of such scenario.

```
MATCH p=(a)-[r:AUTHORED]->(b)-
[r2:REFERENCE]-(c)-[r3:AUTHORED]-(d) RETURN
p LIMIT 5000
```



The diagram below shows the graph network for 5000 nodes returned by the query above.

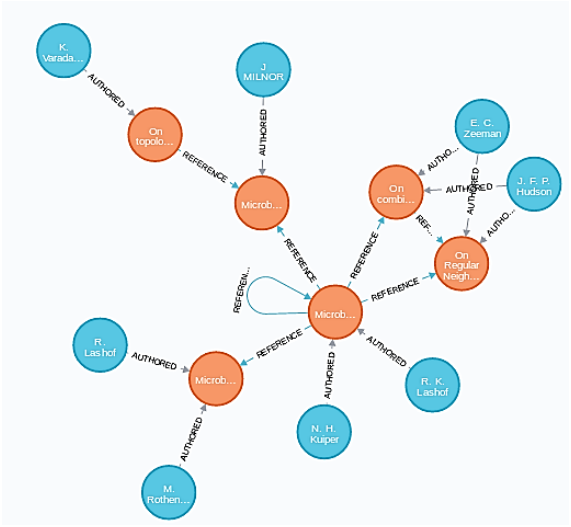


From the diagram above, we could visualize the dataset contained various simple and complex relationships. For example, the diagram below shows a simple hierarchical author-publication-reference relationship: the "NEO Impact Projections" paper authored by J.L Remo cited/referenced the "Overview of Orbits" which was authored by Brian G. Marsden.

Blue color nodes are the Authors Nodes

Orange color nodes are Publication | Paper Nodes

Below shows a more complex hierarchical relationship (3rd degree reference) showing referencing network among papers and the corresponding authors for each of the papers.



*b. **Most cited/reference publications***

Another key scenario is to find the most cited/referenced publication. The query is as follows:

```
MATCH p=(Paper)-[r:REFERENCE]->(m) RETURN
m,COUNT(m) ORDER BY COUNT(m) DESC LIMIT 25
```

Here we projected back the results in a tabular table as shown below.

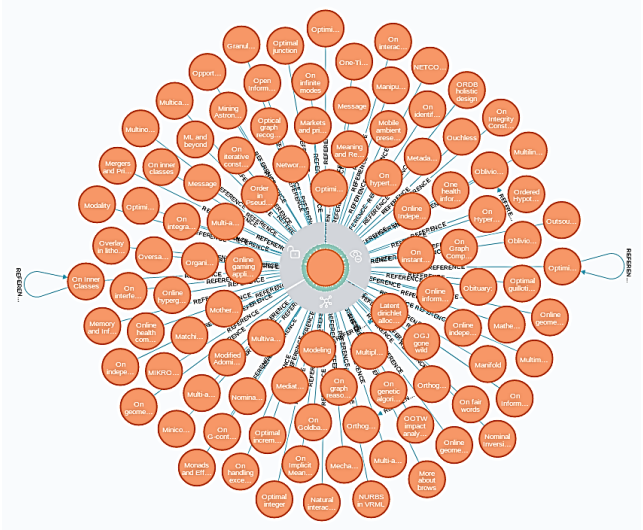
```
$ MATCH p=(Paper)-[r:REFERENCE]->(m) RETURN m,COUNT(m) ORDER BY COUNT(m) DESC LIMIT 25
```

Graph | Table | Text | Code

"m"	"COUNT(m)"
{"id":"55323bdc45cec66b6f9dabfc"}	1539
{"id":"53e99a1fb7602d9702272fbc"}	80
{"id":"53e99800b7602d970200f148"}	54
{"id":"53e9b84ab7602d970440dcf4"}	46
{"id":"53e99a1fb7602d9702273a32"}	42

We cloud then project back a graph of showing all papers referencing this publication with the query below:

```
MATCH p=(Paper) WHERE Paper.id =
"55323bdc45cec66b6f9dabfc" return p
```



c. *Publication with most authors*

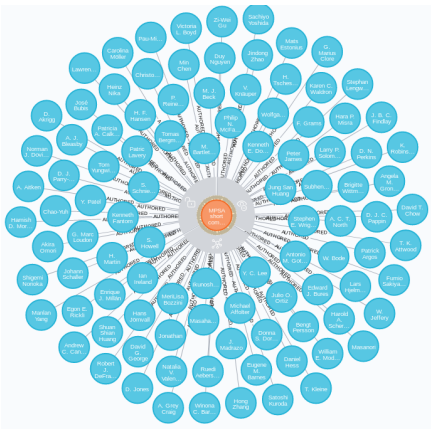
Like the previous scenario, here we find the publications with the most authors.

```
MATCH p=(Paper)-[r:AUTHORED]->(m) RETURN
m,COUNT(m) ORDER BY COUNT(m) DESC LIMIT 25
```

```
$ MATCH p=(Paper)-[r:AUTHORED]->(m) RETURN m,COUNT(m) ORDER BY COUNT(m) DESC LIMIT 25
```

"m"	"COUNT(m)"
{"name":"MPSA short communications","id":"53e99842b7602d97020714b1","lang":"en","year":1994,"citation":0,"url":""}	199
{"name":"Observation of B0→π0π0","id":"53e9983db7602d9702063392","lang":"en","citation":0,"year":2005,"url":"['http://dx.doi.org/10.1103/PhysRevLett.94.181803']"}	190
{"name":"Observation of D0→K+π-","id":"53e9983db7602d9702063302","lang":"en","citation":4,"year":1994,"url":""}	185
{"name":"List of Contributors","id":"53e9983db7602d9702067700","lang":"en","year":1980,"citation":0,"url":"['http://dx.doi.org/10.1109/TPAMI.1980.4766962']"}	177
{"name":"Observation of B+→pΛ¯γ","id":"53e9983db7602d9702063393","lang":"en","citation":0,"year":2005,"url":"['http://dx.doi.org/10.1103/PhysRevLett.95.061802']"}	160
{"name":"Observation of B+→ΛΛ¯K+","id":"53e9983db7602d9702063391","lang":"en","citation":0,"year":2004,"url":"['http://dx.doi.org/10.1103/PhysRevLett.93.211801']"}	150

Below shows the graph network for the paper "MPSA short communications" which has 199 authors.



d. *Finding publications using keyword*

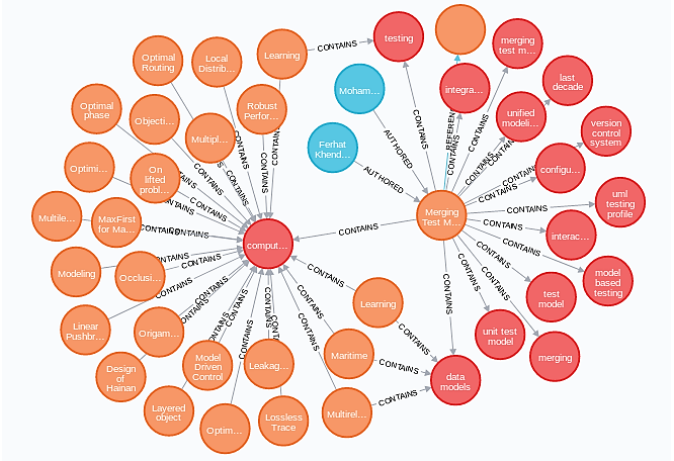
Keyword search is one of most useful and important scenarios for publication search. Below shows a simple query finding all publications containing the keyword "computational modeling".

```
MATCH p=()-[r:CONTAINS]->(k) WHERE
k.name="computational modeling" RETURN
p,r,k LIMIT 25
```

The graph network result is as shown below.

Red color node is the Keywords Node.

Here we could then leverage the interactive drill-down of the Neo4j query tool to drill down to the details for a node. For example, the diagram below the drill-down detail (authors, keywords) of the "Merging Test Models" paper.



VII. CONCLUSIONS

The overall solution leveraging containers with Kafka and Spark to build a distributed graph processing pipeline for massive dataset such as AMiner seems viable even though there were performance setbacks with Spark Streaming processing and loading data using py2neo driver. Using the container virtualization, spin up docker container for each of the processing units like Kafka broker, Spark master and worker nodes and neo4j node. This architecture helps to scale as data to be processed gets huge. As noted in code, we can use producer to publish messages to different topics and spin-up as many spark workers as needed to process and send data to neo4j graph database. Graph databases like Neo4j offers interactive query capability allowing us to retrieve the graphs depicting meaningful relationships among entities/nodes based on the search criteria provided.

As shown in the experimentation results, the ingestion rate on Kafka to Spark side was performant. Spark could ingest 1,000,000 rows of records (3 GB) from Kafka in about 25 minutes. Given that Spark Streaming is fundamentally batch processing in series, we need to overcome the processing bottleneck by leveraging as much of parallelism and distributed mechanics within Spark as possible. A warranted option is to eliminate the usage py2neo drive and fully leverage Spark GraphFrame to process and build the vertexes and edges. In addition, neo4j-spark-connector should be used to push data to Neo4j. This will ensure our solution maximize Spark distributed and parallel processing mechanics and thus the overall reducing data processing time.

VIII. Learnings

The following are the key learnings from our technology review and development phase:

1. GraphX vs Neo4j: Investigated both technologies and choose Neo4j as it solves the purpose of persistence and Graph representation. Neo4j also exposes API's for writing faster queries and gives browser capability to view the results with ease.
2. Docker-compose is a light weight instantiation of containers where we can stich all the Container images from both local and remote images. It provides an effective mean to allow containers or services to interact among themselves via a YAML file. This enables us to reduce setup and development time by not having to have external programs to stitch all the technologies together.
3. We also learned that Cloud PaaS solutions, from AWS and Microsoft Azure, would allow us to extend our work to the cloud. Microsoft Azure particularly offers a rich and mature set of analytics solutions such as Kafka on HDInsight and Azure Databricks. We will plan to leverage these fully-managed solutions for future extension of our work.
4. Spark Streaming is fundamentally batch processing in series. We need to leverage as much distributed and parallel processing mechanics as possible to increase processing performance.
5. Having multiple Spark workers performing atomic jobs in parallel should offset batch processing deficiency in Spark.
6. Leveraging Spark GraphFrame to process and build graph data is a more viable approach in terms of processing performance.

IX. Progress-Timetables

We utilize a weekly research and development cadence with 2 'standups' per week to address and mitigate blockers. Major deliverables are docked to the 4 set Milestones. The following shows the major milestones of our project. For details tasks and deliverables, please refer to Project-LeCloud @ trello.

Milestone	Date	Status
M1-Form Team	01/25	Done
Team Intro – Set Standup and Dev Cadence	01/26	Done
Finalize Project Scope and Proposal	02/17	Done
M2-Submit Project Proposal	02/22	Done
R&D on Cloud and Container	03/06	Done
POC on Kafka, Spark, Neo4j	03/26	Done
Containerize Kafka, Spark, Neo4J	03/31	Done
M3-Submit Progress Report	03/31	Done
Kafka producer ingestion code	4/13	Done
Spark consumer processing code	4/13	Done
Neo4j Graph queries	4/13	Done
Integration and End2End Run, Containerize solution	4/20	Done
Final draft review	4/28	Done
M4-Submit Final Report	05/1	Done

Source Repository:
Github: https://github.com/h0n2/teamCCA

X. Related Works

a. AMiner:

AMiner gathered all the Publication and Microsoft collaboration creating the repository of all Papers under one umbrella. Our current solution nearly matches with AMiner in context of Batch Processing, but we are incorporating Realtime streaming compared to Batch processing. This approach helps to Ingest each found document rather than uploading or processing all bunch of docs at once. For more information, please refer to https://aminer.org/open-academic-graph.

b. Bibliometrics

Bibliometrics offers a data management and consultation tool for enabling research libraries offer service to researchers and universities for better assessment. This is based out of MySQL and typical RDBMS which lacks the current technological advancement tooling for expanding with fast pace and still maintaining accuracy without landing on resources bottleneck. For more information, please refer to http://ceur-ws.org/Vol-1567/paper5.pdf.

Above are two examples for related existing works. These body of works are of similar objective. The main disadvantage of these systems is that the data in the system are static and reprocessing of the data would are not as effective.

Furthermore, the user experience to query the data is at a minimal. With further investment extending our solution to leverage Spark GraphFrame and a scalable graph store like Azure Cosmos DB, our solution would be able to facilitate on the fly data ingestion and graph processing. In addition, we would be able to facilitate greater graph queries for publication searches. One opportunity is to have our solution as the publication data-exchange facilitator that allows libraries, book publishers and various content publishers to push publication meta-data into our system and to allow end users like students and researchers to search of publications and citations.

XI. References

[1] *Connecting Kafka on HDInsight to Azure Databricks*, https://docs.azuredatabricks.net/spark/latest/structured-streaming/kafka.html

[2] *Get Started with Apache Kafka*, https://docs.microsoft.com/en-us/azure/hdinsight/kafka/apache-kafka-get-started

[3] *Get Started with AWS MSK,* https://aws.amazon.com/msk/getting-started/

[4] *Get Started with Docker,* https://docs.docker.com/get-started/

[5] *Kafka on Container,* https://www.kaaproject.org/kafka-docker/

[6] *Kafka listeners,* https://rmoff.net/2018/08/02/kafka-listeners-explained/

[7] *Spark Standalone Cluster on Docker,* *https*://medium.com/@marcovillarreal_40011/creating-a-spark-standalone-cluster-with-docker-and-docker-compose-ba9d743a157f

[8] *Spark, Neo4j on Docker*
https://www.kennybastani.com/2015/03/spark-neo4j-tutorial-docker.html
Kafka setup : https://dzone.com/articles/kafka-setup

[9] Neo4j Cypher Query Language, https://neo4j.com/developer/cypher-query-language/

[10] Custom guide for Neo4j https://neo4j.com/developer/guide-create-neo4j-browser-guide/