Graph Data Processing and Analytics Using MongoDB

Aditya Vardhan
IMT2019003
aditya.vardhan@iiitb.ac.in

Archit Sangal
IMT2019012
archit.sangal@iiitb.ac.in

Gagan Agarwal
IMT2019031
gagan.agarwal@iiitb.ac.in

Abstract

The main contribution of this project is to offer a simple and fast method for loading, processing, and analysing graphs, using aggregation pipeline, provided by MongoDB. With the aggregation pipeline, we were able to use a series of stages to filter, group, and transform the given input graphs in a way that is tailored to the user's specific needs. Each aggregation pipeline that we've written, could be thought of as a component, that can be reused for various instances of a series of graph transformations. If these components are stored in a folder or are stored in a reachable reference repository, they can be used like a library to build up user defined pipelines.

Keywords: MongoDB, Graphs, Algorithms, Map Reduce, Aggregation Pipeline, Data Processing, Data Analytics

1 Problem Definition

1.1 What is the problem you are trying to solve? This should include the background description of the problem, and how is it related to the overall goal/theme and sub-categories.

The project's goal is to provide an easy and quick way to load, process, and analyse graphs utilising the aggregation pipeline made available by MongoDB. In many different domains, such as social networks, recommendation engines, and biology, graphs are a prevalent data structure. Large graph processing and analysis, however, can be difficult and time-consuming. The project seeks to address this issue by utilising the capabilities of MongoDB's aggregation pipeline. The project can adjust the analysis to the demands of the user by filtering, grouping, and transforming the input graphs through a series of stages. The overall objective of the project is to create a reusable set of components that may be used for various graph transformation scenarios, facilitating and accelerating the analysis of big graphs.

1.2 Why is this problem/task/application interesting from a NoSQL systems perspective?

From the standpoint of NoSQL systems, this problem is interesting since it entails processing and analysing enormous graphs, which can be difficult and time-consuming using conventional relational databases. Graph data is a good fit for NoSQL databases because they are made to handle significant amounts of unstructured data, like MongoDB. MongoDB's aggregation pipeline offers a robust toolkit for managing graph data, making it possible to carry out complicated analysis and transformations quickly. This project showcases the advantages of using MongoDB's aggregation pipeline for graph analysis and illustrates the possibilities of NoSQL systems for managing graph data.

1.3 What are the main results/insights you intend to obtain?

The main results are the components and their re-usability. The insights we get is how the library is coded and how it can be extended further. We tried to find some similar project to our but found nothing online. So this projects looks like one of a kind that uses MongoDB. I would be using the word "library" for this project from now on.

2 Approach

We can have a database which represents a collection of graphs. Each collection of the graph is either a graph or it represents a property of a graph. We use two forms of graphs the graphs that are store in adjacency list and the graph that are stored in edge list. Also, we deal with both directed and undirected graphs.

2.1 The algorithms/techniques/models that you have used in this study.

Algorithms: One of the libraries 'find the component's vertices of the graphs' uses DFS and/or BFS.

Models: List the components that we have made using the aggregation pipeline are listed below. Look in the README.md for the details of how to use them.

- Making a database
- Making a Petersen's Graph
- Making cyclic graphs of length 5
- Making two cyclic graphs of length 5 (2 components in the graph)
- Copying the current graph to a new graph
- Removing self loops
- Removing multi-edges
- Making a Induced subgraphs after deleting one vertex
- Making a sub-graph after deleting one edge
- Making a Induced subgraphs after deleting multiple vertices
- Making a sub-graph after deleting multiple edge
- Making a Topological Minors of the graph
- Making a Minors of the graph
- Finding and storing Union of Graph
- Converting Edge List Form to Adjacency List Form

- Converting Adjacency List Form to Edge List Form
- Returning a collection of documents that contains the vertices of all the components.
- Converting a Directed graph to Undirected by adding edges
- Given a directed graph return Weakly connected components

2.2 Justifications for using a specific tool or a specific combination of tools.

We are using MongoDB. Our project was based on graphs and properties and operations performed on graphs because these might result in a lot of semi-structured form of data, which can be very well managed by databases like MongoDB. MongoDB's aggregation pipeline offers a robust toolkit for managing graph data, making it possible to carry out complicated analysis and transformations quickly. This project showcases the advantages of using MongoDB's aggregation pipeline for graph analysis. MongoDB also have dedicated tool and stages for graphs like graphLookup for implementing algorithms like DFS and/or BFS.

2.3 How to setup the environment for running the project?

- npm and node version 18 (Assuming that is already installed or you can refer to this link and this link)
- Install mongoDB. You can follow the following link for the same: link. Once the monghsh is accessible using your terminal you are ready to execute the script files.
- For debugging you can also install mongoDB compass but this is optional. You can follow the following link for the same: link
- http-server (Assuming that is already installed or you can refer to this link).

2.4 How does your approach make it simple and easy to build reusable pipelines that are user-friendly?

If you had a chance to look at the README.md of this project, it would be quite clear of how to use the above listed components. Here is a small description of what makes them usable.

Each component is like a function, that takes some input like the collection or graphs to be used, in what collection to store the output graph, what are the collections that can be used to store intermediate results.

For using any components we define variable in a .js file. For example:

- nameOfDB.js

```
dbName = "graphs"
```

- inputVariable.js

```
inputCollectionName = "petersen"
outputCollectionName = "petersen"
intermediateCollectionName="intermediatepetersen"
vertices = [6,7,8,9,10]
```

- Then you may have your component in a javascript file.

We will be using a more advanced mongoDB shell called mongosh for the execution of the scripts of the components. Assuming all the scripts of the components are in a folder called scripts we may use some thing like:

```
$ mongosh ./scripts/nameOfDB.js ./scripts/
    subgraphInducedMultipleVertexVariable.js ./
    scripts/subgraphInducedMultipleVertex.js
```

Now user can get the graphs from the dataset and applu these scripts to them and create a pipeline. This pipeline will have components (these scripts). Suppose we need a subgraph of the original graph and then after taking the subgraph, we need to take a minor of that graph, we can do that simply by using the output of the first component as an input to the second component. Such a pipeline can be executed in the following manner:

```
$ mongosh ./scripts/nameOfDB.js ./scripts/
    subgraphMultipleEdgeVariable.js ./scripts/
    subgraphMultipleEdge.js
$ mongosh ./scripts/nameOfDB.js ./scripts/
    topologicalMinorOneEdgeVariable.js ./scripts/
    topologicalMinorOneEdge.js
```

Sequence is of running the commands is very important like placing of the component of the pipeline.

2.5 List one or two specific examples.

Suppose if we have the graphs in the collection that are being used as an input. Such a graph can be taken from a dataset or can be a synthetic graph (refer to section 2.5 for more details).

Now suppose, we want to find the Weakly connected component vertices in the graph we can run the following command directly to get our result:

```
$ mongosh ./nameOfDB.js ./wccVariable.js ./wcc.js
```

nameOfDB.js is the same as mentioned before. It just contains a variable that has the name of the database. wccVariable.js also contains variable that are inputs to the script. For example:

- wccVariable.js

```
inputCollection = "c5"
intermediateCollection = "c5_intermediate_temp"
outputCollection = "c5_WCC_components"
```

The above file means that the input graph is taken from the collection with name "c5" from the database with name "graphs". Any collection who's name starts with the substring "c5_intermediate_temp" can be used by the script, keep this in mind.

- wccVariable.js

```
db=connect("mongodb://localhost:27017/" + dbName);

inputGraphName = inputCollection
outputGraphName = intermediateCollection
load("./directedToUndirected.js")

inputCollection = outputGraphName
intermediateCollection=intermediateCollection+"2"
load("./componentsOfGraph.js")
```

The above code is an example of how to make a pipeline script. First, we connect to the data base. Then we set the input variable to the script then we call the script. We have done this above at 2 instances, when calling the script directed to undirected graphs and when calling components of the graph script.

To find the Weakly connected component vertices, we need to first convert the directed graph to an undirected graph and then find the connected components. As then every vertex is reachable from every vertex in that component. So we use the script stored in "directedToUndirected.js". Now, I will explain the "componentsOfGraph.js" script, as it is quite rich in the functionalities we have used. Once the variable are set, from the files like "componentsOfGraphVariable.js", we call the "componentsOfGraph.js" script.

```
db=connect("mongodb://localhost:27017/"+dbName);
```

The above connects to the database. Now we begin our aggregation pipeline using the following:

```
db.getCollection(inputCollection).aggregate([
```

We mention the sequence of stage, through which the documents go through. The stages are as follows:

- The first stage below converts the edge list format to a basic adjacency list format by the grouping of vertices:

```
  {
    $group: {
      _id: "$u",
      v: { $push: "$v" },
      w: { $push: "$w" },
    },
  },
```

- To perform the DFS/BFS better, we add a field 'visited' in the document which represents that a particular vertex was visited or not. It's initial value is set to *false*:

```
  {
    $addFields: { "visited": false }
  },
```

- We save this result in an intermediate collection taken as input. This is done so that we can increase the productivity and decrease the complexity of the script, it also help in debugging:

```
  {
    $out: intermediateCollection,
  },
])
```

Now we start a new aggregation pipeline, but now the input is the intermediateCollection:

```
db.getCollection(intermediateCollection).aggregate
    ([
```

- We filter out the unvisited vertices, here this stage is used just for representational purposes, as right now all the vertices are not visited.

```
    {
      $match:
        {
          visited: false,
        },
    },
```

- graphLookUp is one of the most powerful features to iterate on the graphs. It is because of features like these, we used MongoDB. The graph stored or iteration is take from the "from" field. We start the iteration from the correct node identified by the "_id". This is taken as vertex. The "connectFromField" and "connectToField" both are considered as vertex, while if both exist in the document, then it considers that there is an edge. This returns a list of vertices that are reachable from the starting vertex.

```
    {
      $graphLookup:
        {
          from: intermediateCollection,
          startWith: "$_id",
          connectFromField: "_id",
          connectToField: "v",
          as: "vertices",
        },
    },
```

- Update the values of starting vertex and the vertices that are reachable from that verted as visited.

```
    {
      $addFields:
        {
          visited: true,
          "vertices.visited": true,
        },
    },
```

- The above document as a lot of extra information, we remove the extra information, and just store the important information which is the vertices that are reachable from that component, rest of the things are filtered out.

```
    {
      $project: {
        v: "$vertices._id",
      },
    },
```

- We save this result in another intermediate collection taken. This is done so that we can increase the productivity and decrease the complexity of the script, it also help in debugging:

```
    {
        $out: intermediateCollection+"_2",
    },
]);
```

Now we start a new aggregation pipeline, but now the input is the intermediateCollection_2:

```
db.getCollection(intermediateCollection+"_2").
    aggregate([
```

- Though we used used the visited field, but we still have other permutations of the same components. This is done keeping in mind, if this code is extended to strongly connected component. But here we don't require that we unwind the array by v, later to sort it:

```
    {
      $unwind: "$v"
    },
```

- The unwinded documents are sorted in the increasing order. We don't have sort function for sorting the array, we only have sorting function for sorting the documents:

```
    {
      $sort: {
        "v": 1
      }
    },
```

- Now we need to group the documents back together to get the original array. But the arrays are sorted. Hence, we group them with "_id":

```
    {
      $group: {
        _id: "$_id",
        v: { $push: "$v" }
      }
    },
```

- All the sorted array above represents a component of the graph. Now, we need to group the documents based on the array. But grouping by arrays is not allowed in MongoDB. So we introduce another field and convert the array to string and group by that field.

```
    {
        $project:
        {
            _id: 1,
            v: 1,
            component: "$v"
        }
    },
    {
        $out: intermediateCollection+"_3",
    },
])
```

- Starting another aggregation pipeline: Stage below is converting the array to string:

```
db.getCollection(intermediateCollection+"_3").
    aggregate([
    {
        $project: {
            component: 1,
            string: {
            $reduce: {
                input: {
                $map: {
                    input: "$v",
                    in: { $toString: "$$this" }
                }
                },
                initialValue: "",
                in: { $concat: [ "$$value", ",", "
    $$this" ] }
            }
            }
        }
    },
```

- Groping by the string

```
    {
        $group:
        {
            _id: "$string",
            component: { $first: "$component" }
        }
    },
```

- Dropping that string and id as they are not required now. Only the field component is required now.

```
    {
        $project:
        {
            _id: 0,
            component:1,
        }
    },
```

- Storing the output collection in the variable provided. Note that this collection doesn't represent a graph.

It represents documents that contains the vertices of components.

```
    {
        $out: outputCollection,
    }
])
```

You may have noticed that Weakly connected component is also a pipeline, which uses 2 scripts one after the another in a specific order, and not this pipeline is also available as a component. You can find a complete list of hierarchy in the Figure 1.

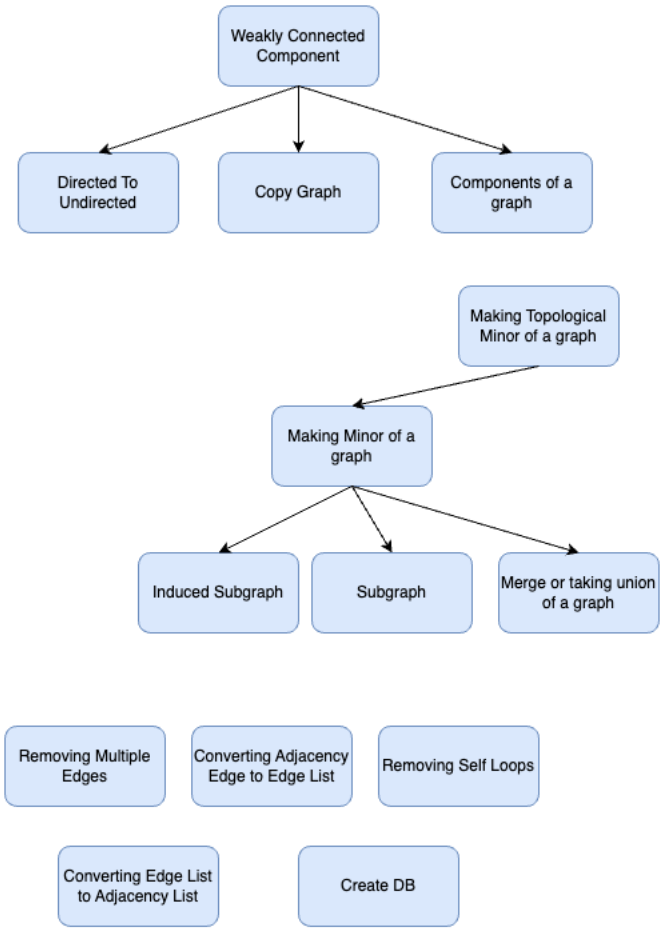

Figure 1. Hierachy Of Scripts

2.6 The dataset(s) that you have used in this project.

For this project, we needed directed and undirected graphs of varied node counts, and since we couldn't find a good fit for our use case online, we made a script file that generates random directed and undirected graphs with a specified user inputted number of nodes. The script can be used to generate graphs of small to large size, and also assign weight to edges randomly or with a specific pattern. The script also can generate a required specific graph from a edge list or adjacency list format. The graph generated using this script can directly be exported to a json file, which can then be imported into our database as a collection using mongoimport. We have used some features of a javascript library called jsnetworkx[10] for implementing these.

3 Evaluation and Results

3.1 The experimental results (in terms of both result quality and efficiency/runtimes) should be briefly evaluated on different parameter settings. The students are expected to use large datasets.

Because we are dealing with graphs and we have taken 3 datasets with variation on number of vertices in graph, i.e. 10 vertices, 100 vertices, and 1000 vertices. The large datasets that we found online, many of them were of size nearly 1000 vertices. We will evaluate on undirected graphs as they have more edges:

	Time taken for 10 vertices	Time taken for 100 vertices	Time taken for 1000 vertices
Importing Graph	0.1	1.9	11.0
Copying Graph	1.7	1.75	3.2
Conversion to Adj. List	1.7	1.77	3.58
Conversion to Edge List	1.71	1.773	4.42
Induced Subgraph	1.83	2.284	1800+
Subgraph	2.154	2.644	1800+
Topological Minor	1.889	2.073	1800+
Minor	2.02	2.517	1800+
Remove Self Loop	1.789	2.073	4.32
Remove Mutli-edges	1.717	2.2	4.57
Components Of graph	1.815	2.605	4.57
WCC Of graph	10.635	1206.4	1800+

System I used for getting these result had the following configuration and brand:

- Brand: Apple
- Model Name: MacBook Pro
- Screen Size: 41.05 Centimetres
- Colour: Silver
- Hard Disk Size: 512 GB
- CPU Model Apple: M1
- RAM Memory Installed Size: 16 GB
- Operating System macOS: 13 Ventura
- Core: 10 Core
- Graphics Card Description: 12 Core GPU
- CPU Speed: 3.1 GHz

3.2 Data Analytics

Once we obtained our final output graph as a collection, we export it to a json file, using **mongoexport**. We then use this json file in our data analytics script. In the data analytics script we use JSNetworkx to create a graph from the json file, and then once the graph is created, we can perform functions on it like, getting the number of nodes, number of edges, finding the cliques, the maximum sized clique, We can also calculate shortest paths between all pairs of nodes.

To visualize any graphs we have also used d3.js, this script takes as input the json file of the graph , and generates a 3d model of the graph that can be viewed on the browser. By default we have provided basic analytics as well as can be seen in Figure 2.

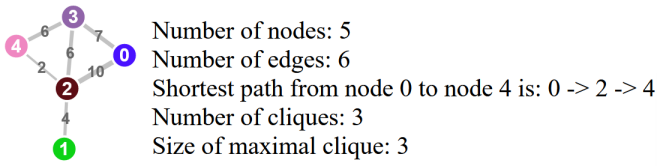

Figure 2. Graph Visualization

References

[1] JSNetworkX, A JavaScript port of the NetworkX graph library, https://felix-kling.de/jsnetworkx
[2] Data-Driven Documents, JavaScript library for manipulating documents based on data, https://d3js.org/
[3] MongoDB Documentation For Installation , https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-os-x/
[4] MongoDB Documentation For Installation , https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-windows/
[5] MongoDB Documentation On aggregation , https://www.mongodb.com/docs/manual/aggregation/
[6] MongoDB Documentation On aggregation , https://www.mongodb.com/docs/manual/aggregation/
[7] MongoDB Documentation On Aggregation Pipeline , https://www.mongodb.com/docs/manual/core/aggregation-pipeline/
[8] MongoDB Documentation On Map Reduce , https://www.mongodb.com/docs/manual/core/map-reduce/
[9] MongoDB Documentation Aggregation Pipeline Operator , https://www.mongodb.com/docs/v6.0/reference/operator/aggregation-pipeline/
[10] MongoDB Documentation Aggregation Operators , https://www.mongodb.com/docs/v6.0/reference/operator/aggregation/