Oracle Benchmarks for Graph Computing

1. Overview

Graph computing is an important aspect in our current world. Many companies and work environment use graph computing to understand the relationship between many things, such as social media, ecommerce, etc. To simply put, graph computing is useful in many areas, and some companies have created their own graph computing service that they also offer to other companies or individuals. For this project, when it comes to evaluating how useful these services are, we mainly measure their performance. Specifically, as the number of computing resources increases, then the time it takes to perform graph algorithms/queries should decrease.

At the University of Washington Bothell, the Distributed Systems Lab (DSL) has created a system called Multi-Agent Spatial Simulation (MASS). MASS is UWB's distributed memory system library, and it uses agents to perform its' computations. MASS also includes a graph computing system that is used with graph database. For this project, we want to compare MASS' performance with the performance of the commercial graph computing systems.

For my role in this project, I currently am working with Oracle Cloud Infrastructure (OCI) to use their database and graph computing system together to measure the performance. OCI gives access to a database where data can be stored, and it also has access to creating a virtual machine instance where our graph server will be held. This virtual machine instance needs a "Flex" shape so that we will be able to change the amount of OCPUs allocated for our graph server. The Oracle PGX Graph Server is the graph server that oracle has created and utilizes when it comes to graph computing. My current goal was to install the Oracle PGX Graph Server on to a virtual machine and test the performance of that graph server using graphs ranging from 1K vertices to 40K vertices. For my current progress, I implemented and benchmarked some simple queries, I implemented and benchmarked some of the given PGX algorithms, and I have worked towards fully implementing Articulation Points.

2. Background

We need to utilize both an Oracle Database and the Oracle PGX Graph Server to test our graphs. The Oracle Database is simply where one can store the data of the graph, such as the vertices, edges, properties, etc. When using a CREATE PROPERTY GRAPH query on this data, the Oracle Database does not actually create a property graph, instead it will create the metadata that is required to understand and reconstruct that graph. Oracle's Graph Server will then use that metadata to understand, interpret, and interact with the graph.

2.1. Understanding PGX

The Oracle PGX Graph Server is a server that when installed on to a machine/system, it will utilize the computing resources of that machine/system to aid in performing parallelism over the graph. Oracle PGX is designed to automatically perform parallelism while using given PGX algorithms. The number of threads available for the server will depend on the number of CPUs on the machine that the server is on. If your PGX server is installed on a machine with 4 CPUs, then PGX can only use 4 CPUs. This also means that PGX will primarily scale vertically. While other systems we are testing, like MASS, utilize multiple machines on a cluster to increase the computing power, Oracle PGX will require allocating more computing on a single machine that the PGX is on. Currently I am utilizing a virtual machine instance on OCI as that allows me to allocate more or less computing power when I need to for testing. The main way to utilize PGX is as a remote server, which means that the client application needs to travel through the network to connect to the PGX Server, and same when the PGX Server needs to connect to the client application.

2.2. Loading Graphs

Oracle PGX usually loads graphs using the metadata of that graph, and Oracle PGX can load that metadata using different data sources. These data sources include JSON files, a Graph Configuration Object, or using the readGraphByName API to read the graph straight from the database. All three of these methods have one thing in common, which is that the data for this graph is stored on an Oracle Database. The data in the Oracle Database is stored in relational tables. In the Oracle Database there must be at least 1 vertex table and at least 1 edge table. The vertex table will hold the ID of each vertex and the properties of each vertex. The edge table will hold the ID of each edge, the source vertex ID and destination vertex ID for each edge, and the properties for each edge. As mentioned earlier, executing a CREATE PROPERTY GRAPH Pgql Statement with the vertex table and edge table will create the metadata for that graph. Once the metadata is created, you can read the graph into the PGX using the readGraphByName API.

Oracle PGX uses this metadata and loads the graph in-memory. The readGraphByName API essentially reads the metadata from the Oracle Database, internally creates a Graph Configuration Object, and then internally creates that graph for analysis. Many of the graph's topology, data, and structure are stored off-heap, however any properties that have the data type of String are stored on-heap. Oracle PGX partitions a graph by using the labels that the vertex or edge has. If two vertices or edges have the same label, then they are grouped together. Oracle PGX also utilizes different partitioning strategies: KEYS_AS_IDS, PARTITIONED_IDS, and UNSTABLE_GENERATED_IDS. The default strategy is PARTITIONED_IDS which creates IDs for vertices and edges by combining the label name with the provided key (user provided or auto-generated). Results for how long it takes to load the graphs can be seen in the Appendix

2.3. Simple Queries

For my current progress I tested some simple queries using the PGX Graph Server. These simple queries are getVertex(), getNeighbors(), and getParents(). getVertex involves seeing how the PGX Server gets a vertex from the graph. getNeighbors is testing to see the performance when we want the neighbors of a vertex. Neighbors are categorized as the vertices that a vertex has an edge with. getParents is similar to getNeighbors, however we instead of getting all vertices, we specifically want the vertices that are seen as the "parents' of the given a specific vertex. Graphs in the Oracle PGX are primarily directed, so this means we need to account for the direction of the edges. These simple queries should not be that intensive of computing resources compared to more complex graph algorithms, and their performance should be relatively quick.

2.4 PGX Algorithms

Maven Central includes the Oracle PGX API which allows one to connect with their PGX Server using Java code. The PGX API also gives access to the Analyst class. The Analyst class allows someone to use PGX Algorithms with their graph. These algorithms are built-in with the PGX Server and include algorithms for counting triangles, local clustering coefficient, weakly connected components, etc. I wanted to test these algorithms to see how PGXs' implementation of these algorithms compared to other services and MASS.

2.5. Articulation Points

Articulation Points (or sometimes described as cut vertices) are vertices where if removed, increases the number of components within a graph. Articulation Points are helpful to find as they can show possible vulnerabilities within the current graph, as if an articulation point gets removed, then the graph becomes disconnected. There are two ways to find articulation points. The first way is to check if the number of total components increased when you remove a vertex. You do this for each vertex, and any vertex that when removed increases the number of components is an articulation point. However, this method is simply too taxing to perform. For smaller graphs it might be fine, but as the graph gets larger it will get slower. The other way to find articulation points is to use Tarjan's algorithm. When using Tarjan's algorithm, you have to take account of two things, discovery time and lowest possible vertex reachable. Discovery time is simply when the vertex was discovered, and lowest possible vertex reachable is the farthest back possible vertex that the vertex can reach based on the DFS tree.

Using Tarjan's algorithm, there are two cases that determine when a vertex is an articulation point. The first case is related to the root vertex. The root vertex is the vertex that the articulation points algorithm started with. If the root vertex has at least 2 subgraphs connected to it, then that is an articulation point. The next case is related to a vertex that isn't the root. For a vertex that isn't the root, if the subtree of a child vertex has no connection (back edge) to an ancestor of the parent vertex, then the parent vertex is an articulation. In Figure 1 you can see how articulation points using those cases create more components.

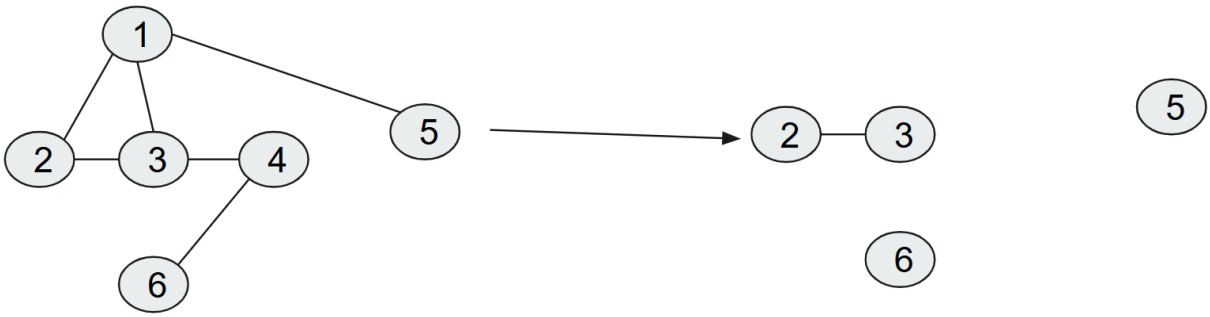

Figure 1: How Articulation Points Are Found

3. Implementation

3.1. Simple Queries

Implementing the simple queries utilizes the API given from the PGX-API Maven dependency. For simple tests like getVertex, geNeighbors, and getParents required using the api functions to interact with the graph and get the vertices. For getVertex(), all I did was use the getVertex function to get the vertex from the graph. getNeighbors and getParents are a bit different. Oracle Graphs are directed, so getNeighbors and getParents do not function the same. For getParents, we need to use the function getInNeighbors() to get the incoming vertices, which would indicate that they are parents of that vertex. For getNeighbors, we need both the incoming and outgoing neighbors, so I use getNeighbors(Direction.BOTH) to get every neighbor vertex that is attached to that vertex. When it comes to testing the simple queries, I ran them 1000 times and calculated the average.

3.2. PGX Algorithms

Implementing the PGX given algorithms is also simple. The Analyst class is the object that executes these algorithms, so I have to create the analyst object using the session from my PGX connection. Using the analyst class, I call the functions for the specified algorithm that I want to test. One thing to note is that the implementation of these algorithms is based off how PGX has implemented them. For example, the triangle counting algorithm implemented in PGX is for the entire graph, which means that it will count the triangles for the entire graph. If you want to count triangles for a specific vertex, then that needs to be implemented on its own. For the PGX algorithms, my main method of getting the benchmarks was running them five times and then calculating the average.

3.3. Articulation Points

When it comes to implementing articulation points, I followed the Tarjan's algorithm approach. Like mentioned earlier, using Tarjan's approach is usually faster than the simple approach. First, I have to implement setting up the articulation points algorithm by gathering all the neighbors for each vertex. I am collecting the neighbors in an ArrayList so that every neighbor of the vertex is contained. The getNeighbors function returns a Collection<PgxVertex<Object>> structure, so I take that collection and turn it into an ArrayList so that I can have access to each neighbor. A Collection only acts as a container that holds the data, so you can't get individual elements, which is why I am using an ArrayList so that I can have access to the individual neighbors in the collection of neighbors. I have to go through each vertex in the graph and collect their neighbors, and all of these neighbors are in one big ArrayList which holds ArrayLists.

Next, I initialize all the necessary data that I need to run the articulation points algorithm. This includes creating the arrays for low and disc, lowest reachable vertex and discovery time respectively. I also create arrays to hold the number of subgraphs for each vertex, the parents of each vertex, and whether or not a vertex has been initially visited. Each array has the length of the number of vertices in the graph. This makes it that the ID of the vertex can act as the index in the array. When it comes to actually running the articulation points algorithm, I created a Stack. Usually when it comes to creating an articulation points algorithm, it involves using recursion. However, when I initially used the recursive approach, I was running into a StackOverflowError. The reason the StackOverflowError was triggering was because too many recursive calls were being pushed into the call stack, which means at some point the call stack had reached its limit. Usually for a smaller graph, like a graph with 1K vertices, this doesn't occur. However, when running the articulation points algorithm for larger graphs, ones with 10K+ vertices, I was running into the StackOverflowError.

In order to prevent this error from occurring, I used a Stack data structure. Using a Stack data structure means that in order to implement the articulation points algorithm, I can't use the recursive approach and instead can only use the iterative approach. When running the articulation points algorithm, I utilize a Stack to emulate the recursive process. When a vertex id is in the stack, we only want to pop it from the stack once we have checked every neighbor. That means initially when you want to get the index of a vertex, you need to peek instead of pop so that the structure of the DFS tree isn't broken.

When a vertex is first initially visited, we need to set the low and disc for that vertex as the current number of hops/moves, and we also want to set the visited for that vertex to be true. Increment the moves by one afterwards. It's important to remember that because we are using the Stack approach, we want to set the low and disc only if the vertex is not visited yet, as if we don't check for that, the low and disc for a vertex originally can be overwritten.

```
int index = stack.peek();
// System.out.println("This
if (!visited[index]) {
    visited[index] = true;
    disc[index] = moves;
    low[index] = moves;
    moves++;
}
```

Figure 2: Setting low and disc

After setting low and disc, we want to go through the neighbors of that vertex. In the iterative stack approach, we have the neighbors for each vertex in an ArrayList so that we can get the neighbors by simply using an index. I created an array in the setup phase called edges. This array simply holds the neighbor index for each ArrayList related to a vertex. Once I get the current neighbor index, we get that neighbor from the current vertex's ArrayList of neighbors. Afterwards we increment that neighbor index, so that next time we can get the next neighbor. Using the current neighbor index, we determine if it's been visited or not. If it hasn't been visited, then that means you add it to the stack so that you can go through the neighbors of that vertex the next loop. I also increment the number of subgraphs for the parent vertex, and you set the neighbors' parent. If the neighbor has been visited and it is not the parent of the current vertex, then you update the low of the current vertex with the minimum between the low of the current vertex and the discovery time of the neighbor vertex.

```
int edge_index = edges[index]; // current neighbor index
if (edge_index < totalEdges[index]) {
    edges[index]++; // Go to next neighbor index
    String sub = StringUtils.substringAfterLast(neighbors.get(index).get(edge_index).getId().toString(), separator:"("); // get edge
    int e_index = Integer.parseInt(StringUtils.substringBeforeLast(sub, separator:")"));
    // System.out.printf("Current edge is %d\n", edge_index);
    if (!visited[e_index]) { // check if visited
        subgraphs[index]++;
        parent[e_index] = index;
        stack.add(e_index);
    }
    else if (e_index != parent[index] && visited[e_index]) { // if already visited
        low[index] = getMin(low[index], disc[e_index]);
    }
}
```

Figure 3: Checking the neighbors

Once we have gone through every neighbor for a vertex, for each neighbor we update the low of the current vertex with the minimum between the low of the current vertex and the low of the neighbor vertex. Afterwards we check the two cases for articulation points I described earlier,

and then we pop from the stack since we're done with that vertex. Once this algorithm is fully completed, then we should have all the articulation points possible in our graph.

```
else { // Post recursion
    for (PgxVertex<Object> v : neighbors.get(index)) {  // Go through neighbors
        String sub = StringUtils.substringAfterLast(v.getId().toString(), separator:"(");
        int e_index = Integer.parseInt(StringUtils.substringBeforeLast(sub, separator:")"));
        low[index] = getMin(low[index], low[e_index]);
        // Articulation Points Cases
        if (subgraphs[index] > 1 && parent[index] == -1 && !points.contains(index)) {
            points.add(index);
        }
        if (parent[index] != -1 && low[edge_index] >= disc[index]) {
            points.add(index);
        }
    }
    stack.pop(); // Remove vertex that's been fully checked
}
```

Figure 4: Getting articulation points

4. Results

One important note to remember is that the benchmark results can vary based on the way you are running the tests. This is also related to how PGX works, specifically connecting to it. There are two primary ways to run the tests: Running on your client machine (laptop, cssmpi machine, etc.) or running on VM on the same subnet as the PGX Server. Every PGX API function/method is a remote procedure invocation. This means that when you call a PGX API function/method, it needs to travel through the network to connect to the machine the PGX server is on. This is also the same if the PGX server needs to send something back. When running from you client machine, the results can vary as communication between the client and the server needs to be through the network, however when you run these tests on the same subnet, that connection time is somewhat nonconsequential. Another thing that's important to remember is that when you first connect to the PGX Server, it needs to establish a connection with the database. This can make loading a graph and some queries seem longer than they are, which can make the results less accurate. To account for this, just have the PGX Server read a graph once so that it establishes a connection to the database and then run the tests again to start properly getting the results. Also, as a note for benchmarking, because PGX uses OCPUs, the equivalent is 2 OCPUs = 1 machine (cssmpi/Hermes).

4.1. Simple Queries

One thing that we can notice with these simple queries is that the number of computing power (in this case OCPUs) does not necessarily impact on the performance of the simple queries. This implies that the PGX Server does not utilize parallelism to perform these queries. We also notice that getNeighbors consistently takes longer than getParents. Once again, this is because

getNeighbors has to get both incoming and outgoing neighbors, while getParents just needs to get the incoming neighbors. This means that getNeighbors does more work than getParents.

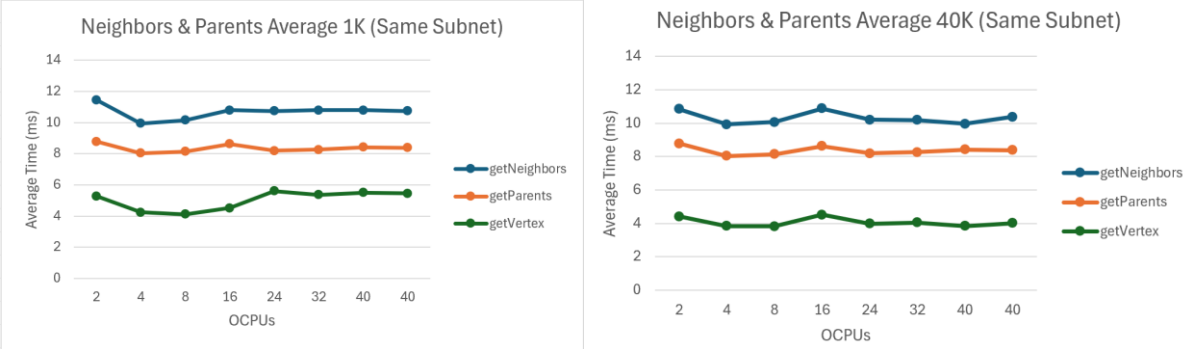

Figure 5 – 6: Averages for Simple Queries for 1K Graph and 40K Graph over 2 – 48 OCPUs

4.2. PGX Algorithms

The PGX algorithms provided by the PGX API and Analyst class are algorithms that are designed to utilize the parallelism of the PGX Server. In the graphs we can see that parallelism in the PGX Server is taking effect. When we add more OCPUs (more computing power), the amount of time it takes to complete these algorithms is shortened. Some of the algorithms that are affected by parallelism are local clustering coefficient, triangle counting, weakly connected components, and article rank. Figure 7 showcases this trend in local clustering coefficient and triangle counting as the average time decreases when the number of OCPUs increases. This trend also exists for the graphs of various sizes, such as 1K vertices, 3K vertices, 10K vertices, etc.

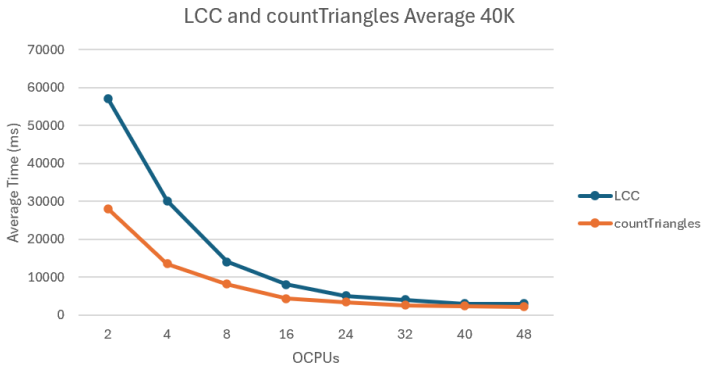

Figure 7: Averages for Local Clustering Coefficient and Triangle Counting for 40K Vertices over 2 – 48 OCPUs

However, there are some algorithms where the amount of OCPUs seems to have no effect or little effect on how it performs. This can give insight into the inner working of these algorithms, as it could show that they have a more iterative approach. If they have a more iterative approach, then that might mean parallelism doesn't really influence how fast it finishes. You can see in

Figures 8 and 9 that the number of OCPUs does not seem to affect an effect on the algorithms compared to local clustering coefficient, triangle count, weakly connected components, and article rank.

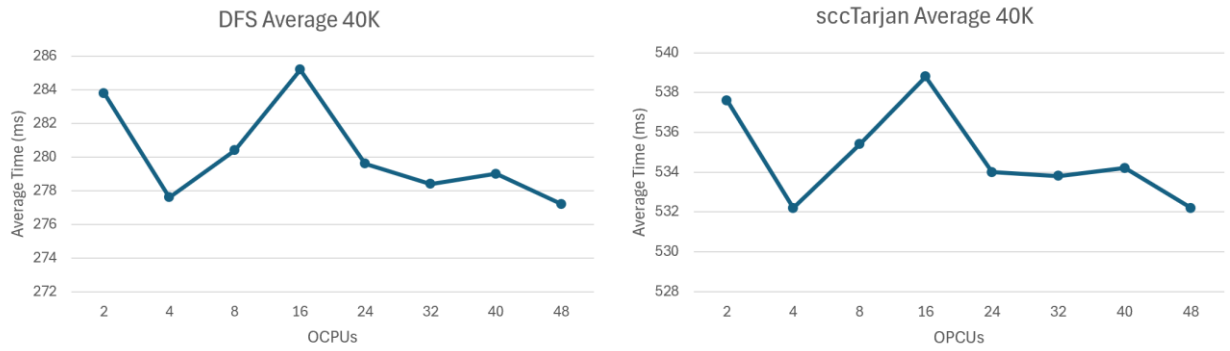

Figure 8 – 9: Strongly Connected Components Tarjan's Approach and DFS Average for 40K Vertices over 2 – 48 OCPUs

4.3. Articulation Points

I have not gotten the chance to collect benchmark results for articulation points yet. This is because there are still some errors in my setup and algorithm that I need to fix. I aim to collect benchmark results by the end of the Summer Quarter.

5. Conclusion

Going into this project, I had very little information on distributed systems, graph computing, etc. However, as I was working on the project, I got to learn more about these topics in general. I especially learned more about distributed systems and graph computing. I was also the first person on this project to set up and learn about Oracle and how they do graphs. Using Oracle and creating benchmarks really helped me understand the idea of why using distributing systems is important, especially when it comes to understanding large datasets.

For Spring Quarter 2025, I mainly aimed to do two things: Set up using Oracle's Graph Server and Database, and to create some benchmarks for Oracle. For Summer Quarter 2025, my aim is to finish writing and getting results for the Oracle benchmarks. Another goal I have for Summer Quarter 2025 is to get into MASS so that I can get results to compare with Oracle's results.

Appendix A: Simple Queries Implementation

```java
public static void getVertex(PgxGraph g, Random rand, String vtable) {
    long elaspedTime = 0;
    for (int i = 0; i < 1000; i++) {
        String vertex = vtable + "(" + rand.nextInt(g.getVertices().size() - 1) + ")";
        long startTime = System.currentTimeMillis();
        PgxVertex<Object> v = g.getVertex(vertex);
        elaspedTime += System.currentTimeMillis() - startTime;
    }
    System.out.printf(format:"It took %.2f milliseconds to get the vertex\n", (float)elaspedTime / 1000);
}

public static void getVertexNeighbors(PgxGraph g, Random rand, String vtable) {
    long elaspedTime = 0;
    for (int i = 0; i < 1000; i++) {
        String vertex = vtable + "(" + rand.nextInt(g.getVertices().size() - 1) + ")";
        long startTime = System.currentTimeMillis();
        Collection<PgxVertex<String>> neighbors = g.getVertex(vertex).getNeighbors(Direction.BOTH);
        elaspedTime += System.currentTimeMillis() - startTime;
    }
    System.out.printf(format:"It took %.2f milliseconds to get the neighbors\n", (float)elaspedTime / 1000);
}

public static void getVertexParents(PgxGraph g, Random rand, String vtable) {
    long elaspedTime = 0;
    for (int i = 0; i < 1000; i++) {
        String vertex = vtable + "(" + rand.nextInt(g.getVertices().size() - 1) + ")";
        long startTime = System.currentTimeMillis();
        Collection<PgxVertex<String>> parents = g.getVertex(vertex).getInNeighbors();
        elaspedTime += System.currentTimeMillis() - startTime;
    }
    System.out.printf(format:"It took %.2f milliseconds to get the parents\n", (float)elaspedTime / 1000);
}
```

Figure 10: Implementing Simple Queries Benchmarks

Appendix B: PGX Algorithms Implementation

```java
public static void startTests(PgxGraph g, Analyst analyst, String vtable) {
    long elaspedTime = 0;
    try {
        for (int i = 0; i < 5; i++) {
            long startTime4 = System.currentTimeMillis();
            long t = analyst.countTriangles(g, true);
            long endTime4 = System.currentTimeMillis();
            elaspedTime += endTime4 - startTime4;
        }
        System.out.printf(format:"It took %.2f milliseconds to get the triangles\n", (float)elaspedTime / 5);
        elaspedTime = 0;
        for (int i = 0; i < 5; i++) {
            String vertex = vtable + "(0)";
            long startTime = System.currentTimeMillis();
            Pair<VertexProperty<String, Integer>, VertexProperty<String, PgxVertex<String>>> res = analyst.filteredBfs(g, g.getVertex(vertex));
            long endTime = System.currentTimeMillis();
            elaspedTime += endTime - startTime;
        }
        System.out.printf(format:"It took %.2f milliseconds to get the bfs\n", (float)elaspedTime / 5);
        elaspedTime = 0;
        for (int i = 0; i < 5; i++) {
            String vertex = vtable + "(0)";
            long startTime = System.currentTimeMillis();
            Pair<VertexProperty<String, Integer>, VertexProperty<String, PgxVertex<String>>> res = analyst.filteredDfs(g, g.getVertex(vertex));
            long endTime = System.currentTimeMillis();
            elaspedTime += endTime - startTime;
        }
        System.out.printf(format:"It took %.2f milliseconds to get the dfs\n", (float)elaspedTime / 5);
        elaspedTime = 0;
        for (int i = 0; i < 5; i++) {
            long startTime4 = System.currentTimeMillis();
            VertexProperty<String, Double> res = analyst.articleRank(g);
            long endTime4 = System.currentTimeMillis();
            elaspedTime += endTime4 - startTime4;
        }
        System.out.printf(format:"It took %.2f milliseconds to get the articleRank\n", (float)elaspedTime / 5);
        elaspedTime = 0;
        for (int i = 0; i < 5; i++) {
            long startTime4 = System.currentTimeMillis();
            VertexProperty<String, Double> res = analyst.localClusteringCoefficient(g);
            long endTime4 = System.currentTimeMillis();
            elaspedTime += endTime4 - startTime4;
        }
        System.out.printf(format:"It took %.2f milliseconds to get the localClusteringCoefficient\n", (float)elaspedTime / 5);
        elaspedTime = 0;
        for (int i = 0; i < 5; i++) {
            long startTime2 = System.currentTimeMillis();
            Partition<String> res = analyst.sccTarjan(g);
            long endTime2 = System.currentTimeMillis();
            elaspedTime += endTime2 - startTime2;
        }
        System.out.printf(format:"It took %.2f milliseconds to get the scc\n", (float)elaspedTime / 5);
        elaspedTime = 0;
        for (int i = 0; i < 5; i++) {
            long startTime3 = System.currentTimeMillis();
            Partition<String> res = analyst.wcc(g);
            long endTime3 = System.currentTimeMillis();
            elaspedTime += endTime3 - startTime3;
        }
        System.out.printf(format:"It took %.2f milliseconds to get the wcc\n", (float)elaspedTime / 5);
        elaspedTime = 0;
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

Figure 11: Implementing PGX Algorithms Benchmarks

Appendix C: Setting Up Articulation Points

```
public static void startAP(PgxGraph g) {
    ArrayList<Collection<PgxVertex<Object>>> neighbors = new ArrayList<Collection<PgxVertex<Object>>>();
    ArrayList<ArrayList<PgxVertex<Object>>> listNeighbors = new ArrayList<ArrayList<PgxVertex<Object>>>();
    int[] size = new int[g.getVertices().size()];
    g.getVertices().forEach(v -> {
        // System.out.printf("The id is %s\n", v.getId().toString());
        String id = StringUtils.substringAfterLast(v.getId().toString(), separator:"(");
        int id2 = Integer.parseInt(StringUtils.substringBeforeLast(id, separator:")"));
        neighbors.add(v.getNeighbors(Direction.BOTH));
        ArrayList<PgxVertex<Object>> temp = new ArrayList<PgxVertex<Object>>(neighbors.get(id2));
        listNeighbors.add(temp);
        size[id2] += neighbors.get(id2).size();
    });

    boolean[] visited = new boolean[g.getVertices().size()];
    int[] low = new int[g.getVertices().size()];
    int[] disc = new int[g.getVertices().size()];
    int[] parents = new int[g.getVertices().size()];
    ArrayList<Integer> APpoints = new ArrayList<Integer>();
    int[] edge_index = new int[g.getVertices().size()];
    int[] subgraphs = new int[g.getVertices().size()];
    Stack<Integer> stack = new Stack<Integer>();
    // long APStart = System.currentTimeMillis();
    g.getVertices().forEach(v -> {
        String firstSub = StringUtils.substringAfterLast(v.getId().toString(), separator:"(");
        int index = Integer.parseInt(StringUtils.substringBeforeLast(firstSub, separator:")"));
        if (visited[index] == false) {
            parents[index] = -1;
            stack.push(index);
            int moves = 0;
            AP.articulationPoints(low, disc, visited, parents, APpoints, moves, size, listNeighbors, edge_index, subgraphs, stack);
        }
    });
}
```

Figure 12: Setting Up Everything Needed to Run Articulation Points

Appendix D: Benchmark Results

Table 1: Simple Queries Benchmarks Client Connection

Average over 1000 runs

OCPUs	Vertices	Edges	get_vertices_avg	get_vertex_avg	get_neighbors_avg	get_parents_avg
2	1000	93480	0	102	220.26	208.98
2	3000	293804	0	103.15	215.62	207.85
2	5000	492890	0	101.53	218.24	245.03
2	10000	989990	0	104.7	219.54	214.53
2	20000	1986380	0	103.06	217.19	216.48
2	40000	3994424	0	103.26	216.63	211.05
4	1000	93480	0	105.92	215.73	211.56
4	3000	293804	0	102.18	216.32	209.59
4	5000	492890	0	100.18	212.3	209.57
4	10000	989990	0.02	102.34	212.73	203.93
4	20000	1986380	0	99.31	211.07	207.02
4	40000	3994424	0	100.29	208.63	205.84

8	1000	93480	0	103.02	216.22	211.1
8	3000	293804	0	103.81	213.49	205.83
8	5000	492890	0	101.77	210.51	206.14
8	10000	989990	0	100.4	212.32	206.28
8	20000	1986380	0	99.92	213.6	207.31
8	40000	3994424	0	100.67	211.18	209.8
16	1000	93480	0	103.33	217.42	218.45
16	3000	293804	0	101.31	213.6	207.07
16	5000	492890	0	100.88	209.91	205.38
16	10000	989990	0	99.91	211.7	205.62
16	20000	1986380	0	100.91	213.96	206.91
16	40000	3994424	0	100.72	216.23	206.53
24	1000	93480	0	103.8	210.27	204.28
24	3000	293804	0	101.58	209.15	207.21
24	5000	492890	0	101.59	212.35	213.79
24	10000	989990	0	102.38	211.81	205.57
24	20000	1986380	0	102.96	211.55	206.46
24	40000	3994424	0	100.09	210.4	205.88
32	1000	93480	0	103.49	218.62	209.87
32	3000	293804	0	102.39	215.39	210.83
32	5000	492890	0	101.64	213.18	209.63
32	10000	989990	0	101.82	213.52	212.96
32	20000	1986380	0	101.53	213.47	208.81
32	40000	3994424	0	102.28	215.55	208.5
40	1000	93480	0	105.22	217.79	220.06
40	3000	293804	0	102.25	216.63	213.01
40	5000	492890	0	103.26	217.46	213.27
40	10000	989990	0	103.81	219.65	217.82
40	20000	1986380	0	103.89	219.42	214.25
40	40000	3994424	0	102.57	220.87	209.41
48	1000	93480	0	103.84	216.28	208.33
48	3000	293804	0	103.11	215.68	209.44
48	5000	492890	0	102.52	216.1	209.85
48	10000	989990	0	102.2	219.68	207.5
48	20000	1986380	0	100.04	210.83	206.11
48	40000	3994424	0	100.4	209.61	207.01

Table 2: Simple Queries Benchmarks Same Subnet

OCPUs	vertexcount	edgcount	get_vertices	getvertex	getneighbors	getparents
2	1000	93480	0	5.28	11.46	8.79
2	3000	293804	0	4.59	11.21	8.78
2	5000	492890	0	4.35	11.26	8.59

2	10000	989990	0	4.54	11.26	8.77
2	20000	1986380	0	4.45	11.07	8.36
2	40000	3994424	0	4.41	10.84	8.51
4	1000	93480	0	4.24	9.94	8.04
4	3000	293804	0	4.03	10.19	8.05
4	5000	492890	0	4.07	10.29	8.1
4	10000	989990	0	3.86	10.18	8
4	20000	1986380	0	3.82	9.91	7.91
4	40000	3994424	0	3.85	9.92	7.8
8	1000	93480	0	4.11	10.15	8.14
8	3000	293804	0	3.83	9.98	8.05
8	5000	492890	0	3.93	10.14	8.14
8	10000	989990	0	4.03	10.13	8.06
8	20000	1986380	0	3.83	9.94	7.8
8	40000	3994424	0	3.83	10.06	8.02
16	1000	93480	0	4.51	10.8	8.63
16	3000	293804	0	4.61	10.6	8.52
16	5000	492890	0	3.94	10.18	8.41
16	10000	989990	0	3.9	10.28	8.22
16	20000	1986380	0	4.5	11.59	9.78
16	40000	3994424	0	4.52	10.88	9.06
24	1000	93480	0	5.61	10.74	8.19
24	3000	293804	0	4.2	10.39	8.41
24	5000	492890	0	3.85	9.8	8.04
24	10000	989990	0	3.99	10.09	8.21
24	20000	1986380	0	3.81	9.81	8.14
24	40000	3994424	0	3.98	10.21	8.39
32	1000	93480	0	5.36	10.8	8.27
32	3000	293804	0	4.45	10.59	8.7
32	5000	492890	0	4.01	10.37	8.64
32	10000	989990	0	4.07	10.32	8.52
32	20000	1986380	0	3.88	10.04	8.35
32	40000	3994424	0	4.05	10.19	8.48
40	1000	93480	0	5.5	10.79	8.41
40	3000	293804	0	4.2	10.3	8.34
40	5000	492890	0	4.06	10.05	8.34
40	10000	989990	0	3.94	10.07	8.42
40	20000	1986380	0	3.95	10.23	8.46
40	40000	3994424	0	3.85	9.96	8.19
48	1000	93480	0	5.45	10.74	8.39
48	3000	293804	0	4.41	10.58	8.5
48	5000	492890	0	4.19	10.6	8.65
48	10000	989990	0	4.04	10.29	8.49

48	20000	1986380	0	4.07	10.4	8.54
48	40000	3994424	0	4.02	10.37	8.59

Table 3: PGX Benchmarks Client Connection

PGX Benchmarks were the average of 5 runs

OCPUs	Vertices	Edges	countTriangles	bfs	dfs	articleRank	LCC	sccTarjan's	wcc
2	1000	93480	1250.6	539.6	446.8	235.4	1161.8	354	336
2	3000	293804	3879.8	442.4	428.4	253.2	3672	323	320.6
2	5000	492890	5127	430.4	447.4	269.4	5756.8	345	359
2	10000	989990	6464.2	426.2	447.4	343.4	12001	377.8	364.4
2	20000	1986380	26331.2	425.2	555.4	493.8	28946.6	461.6	438.8
2	40000	3994424	34216.2	415	657.4	702.8	71055	612.4	595
4	1000	93480	845	480.2	433.4	228.6	736.4	325.8	324.4
4	3000	293804	2149.4	430.2	438.4	210.6	2619.8	315	306.4
4	5000	492890	2803.4	416.2	459.2	231.2	3671.8	335.4	304
4	10000	989990	3956.4	444.6	442.2	246.8	6769.4	375.4	343.8
4	20000	1986380	10377.8	422.8	525.6	299.6	11998	446.2	369
4	40000	3994424	16810	410.8	621.6	405.4	32094.2	649	433.2
8	1000	93480	870.4	496	440.8	251.2	443.8	321	362.2
8	3000	293804	1313.6	439.8	437.6	214.8	1031.2	314.4	303.2
8	5000	492890	1732	412.8	441.2	203.8	1581.6	347.6	308.8
8	10000	989990	2817.6	429.2	442.2	210.8	3659.2	389.4	304.8
8	20000	1986380	4921.6	419.6	522.2	253.6	5945.2	438.6	348
8	40000	3994424	12110.8	426.6	638.2	315	14082.6	652.2	378.2
16	1000	93480	793.8	508.8	432.2	219.4	379.2	330	364.8
16	3000	293804	969	451.4	435.8	209.4	752	314	317
16	5000	492890	1035.8	431.8	442.6	206.2	1036.2	354.8	306.6
16	10000	989990	1637.8	420.8	461	207.6	1787	386.2	302.2
16	20000	1986380	2822.8	423	512.4	225.8	3645.8	446.2	318.6
16	40000	3994424	4127.4	418.8	625.6	238.4	7856.8	624.6	351.6
24	1000	93480	753.2	487.8	442.6	236.8	324.2	313.6	353.6
24	3000	293804	733.4	423.8	432.4	215.8	476.8	305.8	312.4
24	5000	492890	798	411.6	428.6	200.2	729.4	332.4	308.2
24	10000	989990	1176.8	426	442.6	200.8	1564	352.6	309.2
24	20000	1986380	2830.4	422.6	512.4	208.8	2588	430	307
24	40000	3994424	3880.8	419	598.8	249.2	5863.2	599.8	335
32	1000	93480	762.4	493.4	436.2	235.4	304.4	333.4	354.4
32	3000	293804	997.4	433.8	434	200.8	424.4	305.6	317.8
32	5000	492890	703.6	420.2	428.2	207.8	584	346	312.6
32	10000	989990	893	416.4	440.8	208	1046.2	370.8	296
32	20000	1986380	1755.8	441	504.8	217	2582.8	434.8	299.8

32	40000	3994424	3042.8	410	593.2	245.2	4709.4	620.8	303
40	1000	93480	529.4	407.6	431.2	219.4	238.8	218.6	302.6
40	3000	293804	550.2	425	429.4	203	384.6	304.4	303.8
40	5000	492890	625.2	401	423.2	204.8	569.8	334.8	304.4
40	10000	989990	731.4	430.8	447.2	199.6	978.2	380.2	295.6
40	20000	1986380	1768.6	406.6	529.6	213.8	1563.8	439.2	302.6
40	40000	3994424	2237	422.8	595.2	212.6	3652.4	604	312.2
48	1000	93480	396	402	425.8	212.4	237.2	297.6	346.8
48	3000	293804	570.6	412.6	419.4	203.8	363.8	305.6	302.4
48	5000	492890	585.8	417.2	434.8	204.2	457.6	337.6	315.8
48	10000	989990	729.4	416	453.6	201.4	736.8	368.8	314.4
48	20000	1986380	1211.8	406.4	513.4	197.6	1555.2	441.4	296.4
48	40000	3994424	2842.4	414.2	616.6	208.8	2612	635.2	311.4

Table 4: PGX Benchmarks Same Subnet

OCPUs	vertexcount	edgcount	countTriangles	bfs	dfs	articleRank	LCC	sccTarjan	wcc
2	1000	93480	690.6	37.6	32.8	36	1049.6	27.2	30.8
2	3000	293804	2147.4	47.6	55.8	47.2	3049.2	51.2	51.2
2	5000	492890	3149.2	39	91.4	83	5059	79.4	77.2
2	10000	989990	6078.8	43.4	100.4	145.8	10064	138.6	99.6
2	20000	1986380	10959.2	49.8	152.2	276.6	19081.8	279.4	255.4
2	40000	3994424	28051.2	52.4	283.8	533.4	57148.6	537.6	537
4	1000	93480	583	21.8	27	19	535.4	23.8	17.6
4	3000	293804	2100.4	33.8	51.6	31.2	2043.8	48.4	30.6
4	5000	492890	3101.8	27.6	59.6	44.8	3046.4	55.6	33.2
4	10000	989990	3939.8	36.6	84.2	77.4	6050.4	81	69.2
4	20000	1986380	9616.8	35.8	148.6	144.8	11868.2	275	121.4
4	40000	3994424	13558.4	35.4	277.6	272.6	30089	532.2	249
8	1000	93480	360.2	26	27.2	19.6	325.8	22.4	19
8	3000	293804	1082.6	44.6	51.8	28.8	1045.2	49.2	21.4
8	5000	492890	2106.4	31.4	60.2	28.4	2046	76	28.2
8	10000	989990	2122.8	39.2	85.6	45	3048.6	121.6	38.4
8	20000	1986380	5191.8	26.4	150.2	78.8	6055.8	277.2	82
8	40000	3994424	8130.8	33.4	280.4	146	14068.6	535.4	136.4
16	1000	93480	564	31.4	29.2	20	147.2	24.4	17.8
16	3000	293804	577	40.2	52.8	19.8	532.4	49.4	17.6
16	5000	492890	1085.8	27.2	66.8	20.2	1046	83	23.4
16	10000	989990	1107.2	34	87.2	30.6	2048.8	97.6	32.4
16	20000	1986380	3143.4	28.2	152	47.2	3052.6	280.2	41.4
16	40000	3994424	4414.6	30.2	285.2	83.4	8065.6	538.8	81.2
24	1000	93480	388.8	158.6	38.2	47.8	172.4	45.8	113.2
24	3000	293804	676.8	41.8	58.4	18.8	530.4	48.6	20.8

24	5000	492890	568.4	49.4	63.6	18.4	529	61.2	19.2
24	10000	989990	793.4	38.6	84.2	27.4	1042.4	133.2	23.6
24	20000	1986380	2138.4	26.6	150	45	2047.4	275.8	41
24	40000	3994424	3397.8	27.6	279.6	79	5054.8	534	74.8
32	1000	93480	611	94	36.6	46.8	120	48	85
32	3000	293804	684	42.2	58.2	19.2	271.8	48.4	20
32	5000	492890	572.8	42.2	57.8	18.6	531.4	61.8	16.2
32	10000	989990	592.6	73.8	85	27.4	1042.8	94.8	26.6
32	20000	1986380	2138.6	37.6	149	27	2044.8	275.4	30.6
32	40000	3994424	2564.8	26.6	278.4	45.6	4052.6	533.8	50.6
40	1000	93480	588	98.6	46	49.8	121	46.8	71.2
40	3000	293804	680.4	65	64.8	18.8	274.8	48.2	19.4
40	5000	492890	426	40.8	65.2	19	530	62.2	16.6
40	10000	989990	595.2	51.4	83.8	25.8	1045.6	82	21.8
40	20000	1986380	1129.8	29.4	149	27	2047	198.2	26.6
40	40000	3994424	2409.8	28.2	279	45.4	3048	534.2	45.2
48	1000	93480	606.4	107	40.4	48	172.2	49.6	65.2
48	3000	293804	323.4	54.6	50.6	27.6	276	48.8	19.6
48	5000	492890	318	55.4	65	18.8	273.6	55.4	18.2
48	10000	989990	594.4	40.8	84.6	25.8	534	96.2	21.6
48	20000	1986380	1111.8	27.2	149.4	27.4	1046.8	250.2	23.4
48	40000	3994424	2206.4	39	277.2	44.8	3052.6	532.2	47.8

Table 5: Load Times Same Subnet

OCPUs	Vertices	Edges	load_time_ms
2	1000	93480	1313
2	3000	293804	2333
2	5000	492890	2575
2	10000	989990	3460
2	20000	1986380	5435
2	40000	3994424	7334
4	1000	93480	1277
4	3000	293804	2350
4	5000	492890	2331
4	10000	989990	2301
4	20000	1986380	4259
4	40000	3994424	6375
8	1000	93480	1249
8	3000	293804	2339
8	5000	492890	2320

8	10000	989990	2314
8	20000	1986380	4285
8	40000	3994424	6334
16	1000	93480	1259
16	3000	293804	2248
16	5000	492890	2234
16	10000	989990	2244
16	20000	1986380	3246
16	40000	3994424	5262
24	1000	93480	1262
24	3000	293804	2239
24	5000	492890	2239
24	10000	989990	2230
24	20000	1986380	3232
24	40000	3994424	5239
32	1000	93480	1240
32	3000	293804	2242
32	5000	492890	2250
32	10000	989990	2238
32	20000	1986380	3250
32	40000	3994424	6252
40	1000	93480	1260
40	3000	293804	2249
40	5000	492890	2235
40	10000	989990	2229
40	20000	1986380	4242
40	40000	3994424	6243
48	1000	93480	1233
48	3000	293804	2284
48	5000	492890	2234
48	10000	989990	2228
48	20000	1986380	4243
48	40000	3994424	6250

Appendix E: Running the Benchmarks

1. **Clone the repo (oracle-benchmarks branch)**
   git clone -b shahzham/oracle-benchmarks --single-branch
   https://bitbucket.org/mass_application_developers/mass_java_appl.git

2. **Go to the directory**
   cd Graphs/OracleBenchmarks/SimpleAndPgxBenchmarks

3. **Setup Database and PGX Graph Server**
   Go read the README in the directory

4. **Package the jar**
   mvn clean package

5. **Running code**
   Note: Only graph_name and vertex_table_name are required, if you already created the graph, then there is no need for the other arguments. However, if you wish to create the graph, then the other arguments are required. create_graph can only be true or false. I already took the data from Michaels' branch and turned them into txt files for the code; you just need to give the file name. I also suggest using easy name conventions for graphs and tables to make them easier to remember. For example, a graph name can be GRAPH1 with the vertex table VTABLE1 and edges table ETABLE1.

   a. **On client**
      java -jar target/SimpleAndPgxBenchmarks.jar <graph_name> <vertex_table_name> <edge_table_name> <create_graph> <file_location>

   b. **On same subnet as Server**
      scp -i /location/of/ssh.key /location/of/jar.jar username@publicIP:~/

      ssh -i /location/of/ssh.key username@publicIP

      java -jar SimpleAndPgxBenchmarks.jar <graph_name> <vertex_table_name> <edge_table_name> <create_graph> <file_location>

      Note: Using the server VM is fine, the results should be the same/similar when just executing jar on another VM on the same subnet or just on the VM the server is on.