2022

DATAKEEPERS

FLIGHT NAVIGATOR

Alhareth Aboud

Logan Kloft

Nam Jun Lee

Madee Barnwell

Stephany Lamas

Introduction

The Data Keepers set out to create a flight navigator program that would be useful for more than just passenger air travel navigation purposes. Airport and airline searches, as well as airline aggregation, provide the user with information outside of what a typical travel search engine offers. These queries are more research-focused and can be used by journalists looking for aviation facts to include in news articles. They are beneficial for fiction writers looking for accurate and realistic information to include in their writing, and, additionally, they can be used by academics researching papers and reports, or simply by anyone with an interest in aviation.

Initially, Neo4J was used to write queries for the ten questions proposed by the Data Keepers in their Milestone 1 report. After these queries were implemented successfully, custom algorithms were created using Hadoop's MapReduce framework to compare against the Neo4J implementation with the intent of improving query speed.

The amount of aviation-related data is only going to increase in the future and having quick and reliable access to this information is vital. For the stranded traveler, having an accurate and speedy response to a query such as *"Given a country X, provide a list of airports operating in Country X"* could provide reassurance that a way home is within reach. On the other hand, a journalist writing about a travel itinerary for a trip around the world could benefit from information provided by a query such as *"Given two cities X and y, provide a list of routes connecting cities X and Y".* This report details the custom implementation of those two questions. A description of each custom algorithm is followed by its results and analysis. This report concludes with the team's lessons learned and ways to improve upon the algorithms' designs by using **CustomWritable** objects described by M. Ryczkowska and M. Nowicki in their report "Performance comparison of graph BFS implemented in MapReduce and PGAS programming models".

Related Work

M. Ryczkowska and M. Nowicki, "Performance comparison of graph BFS implemented in MapReduce and PGAS programming models," *Parallel Processing and Applied Mathematics*, pp. 328–337, Mar. 2018. https://link.springer.com/chapter/10.1007/978-3-319-78054-2_31

Ryczkowska and Nowicki compare Graph BFS implementations in Hadoop MapReduce and PCJ (Parallel Computations in Java) which allow for parallel and distributed computation in Java. The authors walk through their MapReduce implementation and provide code snippets. The implementation uses a **CustomWritable** object that represents a node containing an adjacency list in addition to parameters for distance and color which defines whether a node has been visited or not. The program starts at a random vertex and visits all nodes, updating distance and color until all nodes are visited. The PCJ implementation is based on other work. PCJ uses tasks to complete work. Each task uses two queues to keep track of vertices. The first queue tracks vertices currently being processed, while the second keeps track of vertices that are one hop away from the vertex currently being processed. A task only knows information about the vertices in its queues and is partitioned in such a way that if an expansion occurs that is outside of the bounds of the task, it will communicate with the other tasks to send that vertex to the relevant task. The results of the paper found that running MapReduce with a custom class was significantly faster than using text-based values. It also reported that the PCJ implementation was about 100 times faster than the MapReduce version for the datasets tested.

The Data Keepers' Flight Navigator project relates to this prior work since a variation of a BFS algorithm is performed using MapReduce. A distinction between the two is that Ryczkowska and Nowicki's work used a **CustomWritable** class to represent their graph, and then compared it to a text implementation, while the Data Keepers are using our own text implementation. Their report concluded that a **CustomWritable** class was more effective for graph traversal in Hadoop than using **Text** as values for the mapper and reducer. This conclusion points our team in the right direction towards improving the efficiency of our MapReduce program.

Ballas, V. Tsakanikas, E. Pefanis, and V. Tampakas, "Assessing the computational limits of graphdbs' engines - A comparison study between Neo4j and Apache Spark," *24th Pan-Hellenic Conference on Informatics*, 2020. https://dl.acm.org/doi/pdf/10.1145/3437120.3437356

The authors compare Neo4J and Apache Spark for big data processing, attempting to reach the limits of the two. They provide a historical background on how big data applications were originally tailored to individual tasks but have become generalized like MapReduce. However, another problem arose where these big data tools needed to be used together to perform meaningful data analysis. Thus, the introduction of Apache Spark has tried to combine some of the requirements for big data processing. The article describes the background of both Apache Spark and Neo4J Project, then moves onto the experiment where they use PageRank as the benchmarking algorithm on datasets from SNAP (Stanford Large Network Dataset Collection). The results show that Neo4J is better for smaller datasets but struggles for larger datasets that either reach the storage bounds of the computing device or require distributed storage. In this case, the three different Spark environments outperformed Neo4J. Additionally, for one dataset, Neo4J was not able to successfully perform the PageRank algorithm.

This report is useful for our application because it tells us that Neo4J is not feasible for a considerably large dataset. Instead, we must consider other alternatives such as Apache Spark. The information presented also seems to match some of our own findings. The Neo4J driver is at times unreliable. There have been multiple occasions where the team has come across instances where the python driver for Neo4J stops working because the AuraDB instance has gone offline due to an error. Other big data applications have more reliable fault tolerance than Neo4J does. This makes them a more appealing solution than Neo4J even if querying using other tools is more difficult than using Neo4J.

Custom Algorithms

The following sections describe the two custom algorithms that were implemented using MapReduce.

Q1

Given a country X, provide a list of airports operating in country X.

The first algorithm receives a country name as an input and outputs the airports in that country (if any). The algorithm uses the Apache Hadoop MapReduce framework. Data is stored in comma-separated lists. Attributes for the csv format are shown below, with the indices artificially inserted:

airports.csv:

[0]	AirportID	[1]	Name	[2]	City	[3]	Country	[4]	IATA	[5]	ICAO	[6]	Latitude	[7]	Longitude

The main program takes three arguments: Input source, output destination, and target country. The target country is added to the Configuration details of the job and given a key value of "Country". This allows map and reduce tasks to access user-defined values.

The mapper receives a record from a split. The job of the mapper is to extract the country field from the record so that the reducer can aggregate on the country. The mapper Ignores the key parameter, which is not applicable in the map function. Instead, the mapper focuses on the value parameter, which is a record. The map function converts the value to a String object, line, then splits up line based on the "," delimiter and stores the result in records, before finally extracting the country value and emitting the country as the key, and value as a value.

In the reducer, values are collected by their country. The reducer only emits values whose key matches the original input country. Notice the setup function; its job is to retrieve the target country value stored in the job's Configuration and store it in private member **targetCountry**. Inside the reducer function, the key is converted to a String and compared to the **targetCountry** String. If they match, the reducer will iterate through values and emit each value separately. Only one key will match the **targetCountry** in the reducer, which means that only one reduce instance will iterate through its collection of values.

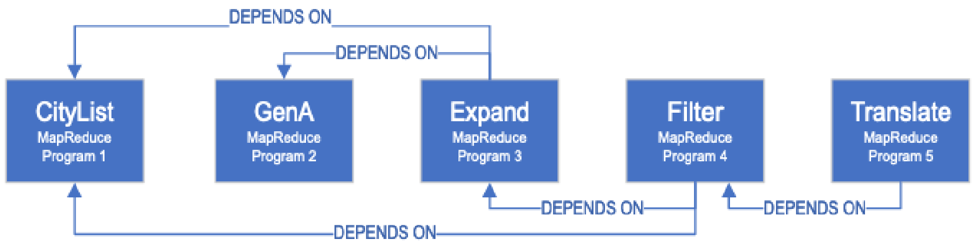
The result is the records whose country match the target country. Due to outputting the **targetCountry** as the key in the reducer, records are prepended with the target country followed by a space.

Q7

Given two cities X and Y, provide a list of routes connecting cities X and Y.

The next custom algorithm returns paths between two cities. These paths can be a minimum length of two, which is interpreted as a route from one airport (source) to another (destination). In a path, airports are separated by the airline that flew the route from the source airport to the destination airport. Additionally, this algorithm allows the user to specify how many iterations to run, with each iteration representing an increase in the maximum possible length of paths by one. The smallest number of meaningful iterations is one, which generates paths of length two and three.

The algorithm has five different sets of MapReduce programs. The fifth set depends on the outputs of the fourth, the fourth on the outputs of the first and third sets, and the third on the outputs of the first and second sets. All five MapReduce programs use the same values for the input and output of map and reduce. Map expects **LongWritable (key)** and **Text (value)**. Map writes **Text (key)** and **Text (value)**. Reduce expects **Text (key)** and **Iterable<Text> (value)**. Reduce writes **Text (key)** and **Text (value)**.



The solution uses airport, airline, and route information which are stored in comma separated value formats. Their attributes are listed below along with artificially inserted indices.

airports.csv:

[0]	AirportID	**[1]**	Name	**[2]**	City	**[3]**	Country	**[4]**	IATA	**[5]**	ICAO	**[6]**	Latitude	**[7]**	Longitude

airlines.csv:

[0]	AirlineID	**[1]**	Name	**[2]**	IATA	**[3]**	ICAO	**[4]**	Callsign	**[5]**	Country	**[6]**	Active

routes.csv:

[0]	Airline	**[1]**	Name	**[2]**	Source Airport	**[3]**	Source AirportID	**[4]**	Destination Airport	**[5]**	Destination AirportID	**[6]**	Codeshare	**[7]**	Stops

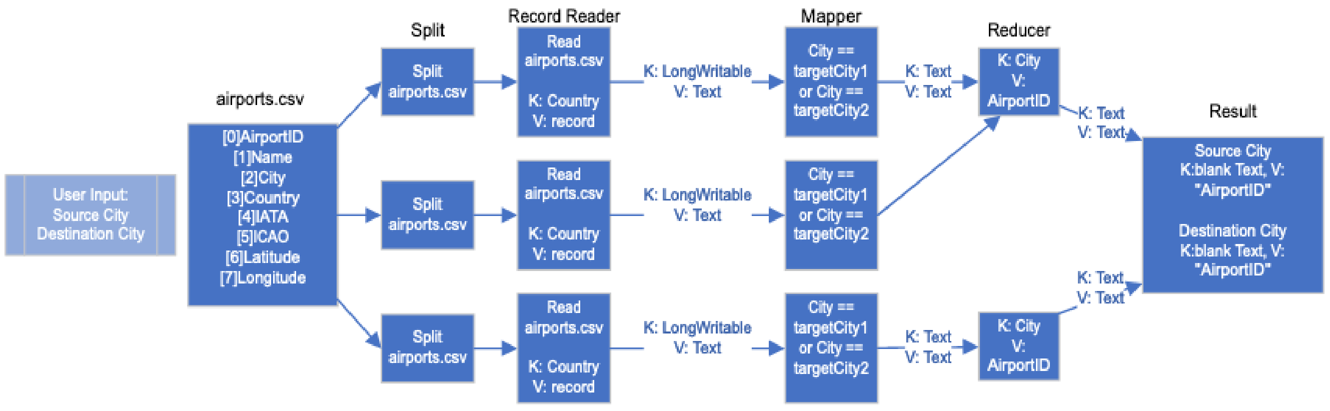
First MapReduce Program: CityList

The first MapReduce program, **CityList**, operates on the **airports** dataset. **CityList** is given two cities as its input and returns two separate lists: one for each city. Each list contains all **AirportIDs** for airports that operate in that city. Later, the first list will be used to start and build paths from **airports** with matching IDs. Meanwhile, the second list will be used to select paths that end with a matching ID from its list, effectively filtering paths that do not start and end from the two input cities.

The main program uses two command line arguments: the source and destination cities. It then adds the cities to the MapReduce job’s configuration with keys “TargetCity1” and TargetCity2” respectively.

The mapper class uses a setup function to load the values associated with TargetCity1 and TargetCity2 into similarly named variables. The first step in the map function is to convert value into a string and parse it on delimiter “,”, and then store it in a list of Strings called record. Next, the map function checks the city of the airport record to see whether it is equal to either TargetCity1 or TargetCity2. If there is a match, the map function writes the city as the key and ID as value for the reducer. Otherwise, no write occurs.

Only records whose city matches the target cities will be included in the reduce step since they were filtered out in the map step. Thus, the reducer only needs to combine the **AirportIDs** into a list of IDs for use in future steps of the algorithm. The reducer accomplishes this by iterating through its values, converting the **Text** to a string before appending it to a growing String of prior IDs, each of which is separated by a space. The final step involves adding ‘”’ to the beginning and end of the String of IDs so that the output is formatted appropriately to be passed as a command line argument. The IDs String is written as the value and an empty **Text** object is used as the key.



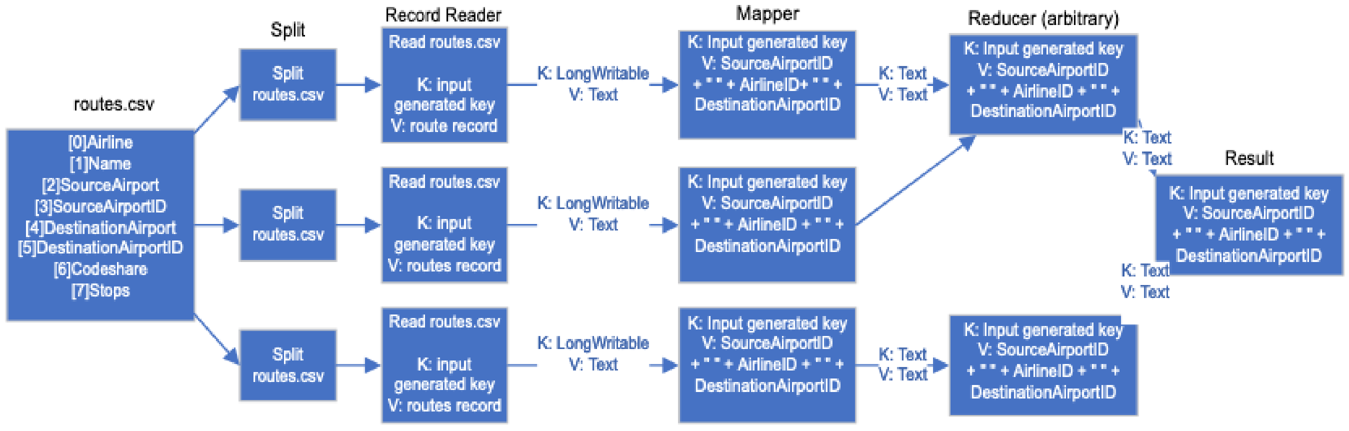
Second MapReduce Program: GenA

The second step in the algorithm is to generate adjacency triples. **GenA** takes route records as an input and combines the **SourceAirportID**, **AirlineID**, and **DestinationAirportID** of each route into a space separated list, making up a single adjacency triple. Future programs will be able to aggregate on an **AirportID** which, if a triple is included, will allow access to a sort of adjacency list.

The main class only sets up the input and output directories and then runs the job; meanwhile, the map function receives individual routes as **Text**. The first step is to convert the **Text** to a String and then split the String into a list of Strings using delimiter ",". That list is then checked to make sure that relevant fields are not empty (contain "N/A"). If they are empty, then the map function terminates. Otherwise, the value for the reducer is created in the following format to make the triple. The key remains the same as the input but must first be converted from **LongWritable** to **Text**:

$$SourceAirportID + ' \ ' + AirlineID + ' \ ' + DestinationAirportID$$

The reducer collects values on the original key of the mapper which has no significant use in this MapReduce program. The reducer's job is arbitrary- it writes all values it receives to output. By the time the reducer has finished, all adjacency triples have been created. These will be used in the next step to increase the length of a path in addition to being the starting point of paths.



routes.csv
[0]Airline
[1]Name
[2]SourceAirport
[3]SourceAirportID
[4]DestinationAirport
[5]DestinationAirportID
[6]Codeshare
[7]Stops
Split
Split routes.csv
Record Reader
Read routes.csv
K: input generated key V: route record
K: input generated key V: routes record
K: LongWritable V: Text
Mapper
K: Input generated key V: SourceAirportID + " " + AirlineID + " " + DestinationAirportID
K: Text V: Text
Reducer (arbitrary)
K: Input generated key V: SourceAirportID + " " + AirlineID + " " + DestinationAirportID
K: Text V: Text
Result
K: Input generated key V: SourceAirportID + " " + AirlineID + " " + DestinationAirportID

Third MapReduce Program: Expand

Now that the list of airports we want to start and end from has been generated, along with an adjacency list, we can create paths from an airport in the starting list to an airport in the ending list.

The main program takes three command line arguments. The first is the number of iterations to run **Expand**, where each iteration increases the possible maximum length of paths by one. The other two arguments are the starting list of **AirportID**s and ending list of **AirportID**s. These arguments are immediately added to a configuration variable that will be reused in successive jobs. This allows the mapper and reducer class to access the starting and ending lists. Next, the program starts iterating the specified number of times. Each iteration creates a new job where only the input and output locations are changed. The input of the first iteration is the output of **GenA.** The output of an iteration is in a folder that ends with the current iteration number. Then in future iterations, the input becomes the output of the previous iteration, while the output follows the same rules as described above.

The mapper considers two cases for its value. The value could either be an adjacency triple, which will be the case for all values in the first iteration of the mapper; however, a value could also be a path. In the case of an adjacency triple, the mapper will emit the beginning ID of the triple as the key and the entire triple as the value. In the case of paths, since expansion occurs at the end of a path, the mapper emits the ending ID of the path as the key and the entire path as the value. It should be noted that *a triple that begins with an ID from the starting destination will be emitted as if it were a path*. This prevents paths that reach a source airport from being expanded anymore, since the adjacency triple(s) will be passed as a path.

The reducer's job is to first check whether a value is a good path. A good path is defined as starting from an **AirportID** in the start list and ending in an **AirportID** from the end list. In this case, the good path is written to output, but prevented from being expanded since going beyond the destination is not useful for the scenario. Regardless of whether the path is good or not, it is checked to see whether it is an adjacency triple. If it is, it's added to a list that will expand paths. If it's not, then it's added to a list of paths which will be expanded (but only if it's not good). The next step after writing good paths to output and separating adjacency triples from paths is expanding the paths. This is done by a nested loop whose outer loop iterates through paths to be expanded and whose inner loop iterates through adjacency triples. The algorithm then checks whether the ending ID of the path matches the beginning ID of the triple. If they match it means the path can be expanded. This is done using two for-loops to format the new path with spaces between each ID. The new path is then written to output as the value. The key is insignificant to the output of the reducer and input of the mapper.

Future iterations of the MapReduce job will generate larger paths as each path is expanded using its related adjacency triples.

Fourth MapReduce Program: Filter

The output of **Expand** creates paths that are good and paths that are not good. The output also includes adjacency triples. To get rid of these values, the final output of **Expand** is filtered for values that start with an ID from the start list and end in a value from the end list.

Similarly to the main class of **Expand**, **Filter**'s main takes the start ID list and end ID list as arguments. It then makes these available in a configuration variable which is used by the **Filter** job. The input location for filter is the last output location of expand. The output location is a new folder, called untranslated, which will be used for the final step.

The mapper splits the string representation of value on delimiter " " and stores it in a variable, record. The start ID of record is then checked to determine whether it belongs to the start list, and the end ID of record is checked to determine whether it belongs to the end list (retrieved from the job's configuration). If both belong, then the mapper uses that path as the value for the reducer and the starting ID of the path as the key for the reducer. The key is not significant in the **Filter** program.

The reducer's job is arbitrary- it uses an empty **Text** object as the key for output, iterates through each value, and writes to output that value as the output value. Output now only contains good paths.

Fifth MapReduce Program: Translate

The current state of the good paths is not useful for humans since it consists of airport / airline IDs. The next step of the program is to translate the airport and airline IDs into the airport and airline names they refer to. The translate program accomplishes this through multiple iterations where each iteration translates from left to right, one ID in a path. In a big picture view, one can think of the translation as column by column.

The main class of **Translate** specifies new input and output locations in addition to a configuration variable for each iteration. The input is always the previous result of the program with the exception of the first iteration, which uses the output of the **Filter** program. The output is computed to be at the file system location that starts with "untranslated-" and ends with the current iteration. Each iteration sets the "round" variable of the configuration to the current iteration. Then, an additional input path is added which depends on the iteration. For odd iterations, the **airports** location is included, otherwise, the **airlines** location is included. The main class receives the number of times to iterate via command line and should be one more than the number of iterations supplied for expand.

The mapper's job is to select the ID to aggregate on in the reducer from the value parameter. There are two cases to consider: whether the value is a record or path. A value is a record if it contains a ",". For records, the ID is the first field of the record. This gets emitted to the reducer as the key, and the value is emitted as the value to the reducer. In the case of a path, we must consider which ID we are translating. This is decided by the field at index round – 1. If the round – 1 is equal to or greater than the length of the path, then the path has been fully translated. To denote that a path has been fully translated, the key is set to "translated." Otherwise, like the record, the algorithm extracts the ID from the path and uses it as the key for the reducer and passes the entire path for the value.

A note about the reducer- *it can receive the following key types: "translated" or an ID*. In the case of translated, the reducer writes every value to the output. Otherwise, the reducer searches for the record (there should only be one) in the list of values and stores paths during the search process in a separate collection so they can be iterated over later. Once the record is found, the reducer extracts the name from the name field. Now, the reducer will replace the ID that is at position round – 1 of each path with name. This is done by iterating over the separate list where paths were set aside and then finishing iterating over the original values list. After each replacement of ID at index round – 1, the path is written to output as the value and an empty key is used (empty **Text** object).

When **Translate** is finished running, the output now contains paths that start in a city from the first city list and end in a city from the second city list.

Further iterations increase the round value, which the input and output locations and “round” configuration variable depend on.

Results and Analysis

Q1 Analysis

The map function requires **O(m)** time where **m** is the longest length of a record. This is decided by the split() function. For the reducer instances, only one will iterate through its list of values. The others will all return in **O(1)** time. If we let **n** be the number of records whose country matches the target country, then the time complexity for reduce is **O(n)**, where **n** could be the entire length of the dataset in the case that it contains only records of the target country. However, our dataset does not have that characteristic.

The number of final records written will never be more than the amount of records input. The mapper will write just as many records as it receives, while the reducer will write that many or less. Therefore, to scale the solution, one should ensure they have equal space of output to the input size to accommodate for the worst-case scenario.

A note about the design- *checking the target country in the reduce function rather than the map function is more efficient than vice versa*. If we consider that the map function processes all records, then checking a matching country will occur with every record. Meanwhile, in the reducer, the check will occur one less time for every duplicate country in the records dataset. Therefore, it is more efficient to place the check in the reducer.

Q1 Results

For Q1 we decided to compare the implementation of filtering the country in the mapper versus the reducer. We used United States as the input country and ran the MapReduce program 10 times, recording the time it took for the job to complete using real time to measure the time. The average time to complete the job when filtering in the reducer was 1.5382 seconds. The average time to complete the job when filtering in the mapper was 1.5560 seconds. The reducer version was 1.14% more efficient. This is not as big of an improvement in efficiency as we had thought would occur. This might be due to the reducer needing to sort inputs based on their key values, and in the case of the mapper filtering, only two key values are passed to the reducer.

Another experiment for measuring the efficiency of Q1 was comparing it to our Neo4J implementation. We used United States as input for both the MapReduce and Neo4J program. The programs were run 10 times, and their runtimes were recorded and summed before being averaged. The average completion time for the Neo4J version was .8730 seconds. All trials were under one second except one, which was over two seconds. We consider this an outlier which brings the average time down to .7184 seconds. If we compare this to the above time for filtering in the reducer, the Neo4J solution is 53.29% more efficient.

For our experimental results, we also considered the output sizes between the MapReduce and Neo4J implementation. There is a large discrepancy between the two. The MapReduce program outputs 1510 airports while the Neo4J implementation outputs 3024 airports. That is approximately double the MapReduce program. If we compare the time per record output, then the efficiency of the Neo4J implementation increases even further at 76.68% more efficient per output record compared to the MapReduce program. One shortcoming of the Neo4J program is the quality of its results. When using a find all in the **airports** file for "United States" there comes back 1512 matches. Since Neo4J has exactly double this amount, it suggests that the Neo4J implementation duplicates results. Interestingly, the MapReduce program is two airports short of what the find all suggests there should be. This might be due to a mix-up of "clean" csv files or records are being dropped in the MapReduce program. We could not find a logical error in the MapReduce program that might have caused this discrepancy.

Q7 Analysis

Let **n** be the number of records in the input of a program. The solution for Q7 uses five different MapReduce programs. The first program, **CityList,** takes as input all records from the dataset. The mapper then parses the dataset in **O(m)** time where **m** is the maximum length of a record. The mapper then checks the record's city field to see whether it is one of the two target cities. This takes at most linear time because equality checking for strings can be done in one pass. On a single processor, this would take **O(nm)** time, but for multiple processors, **n** is reduced by a factor equal to the minimum of the number of processors and splits. For records with matching city fields, they are passed to the reducer and aggregated on the city. The reducer will run at most twice since there are at most two keys from the mapper (the two target cities). In the worst case, all records' cities belong to one of the target cities. Then the reducer will create **n** strings. The result is an **O(n)** operation.

In the **GenA** program, the mapper receives every record from its dataset and processes each one in a similar **O(mn)** time. The reducer simply writes its values to output which is an **O(n)** operation. The **Expand** program is significantly more complex. It uses iterations to expand the length of all paths in its input. We define **x** as the number of paths in an iteration's input and **y** the number of adjacency triples from **GenA** that aren't also a path. Then the mapper runs in **O(mn)** time due to its similar processing as in the first two programs. The reducer, however, expands at most **x** paths **y** times. This means that the reducer's time complexity is bounded by **O(xy)**; however, since **x** increases by each iteration, the output increases exponentially. For example, consider starting with one path and five adjacency triples for that path. Then five new paths will be created. Then, in the next iteration, say each of those new paths also have five adjacency triples. Then the new output will be 25, and the next 125, and so one which is exponential growth. The triples are not spread in such a way in practice, but still lead to a similarly fast-growing output that is upper bounded by an exponential function.

The mapper function of the **Filter** program checks the beginning and ending of all **n** records it receives which is an **O(mn)** operation. The reducer simply outputs what it receives, like **GenA**, which is an **O(n)** operation. The final program, **Translate,** runs in iterations similar to **Expand**. However, it produces output the same size (regarding number of records) as its input. This means that iterations don't increase the number of records in the original dataset. Instead, the lengths of records are typically increased due to the length of airport and airline names being longer than their IDs. The mapper of translate considers the current iteration to be an index value and groups records by the ID at the current iterations index value for the reducer. This is an **O(mn)** operation. The reducer iterates through its value list at most once and performs a string operation on each record that replaces part of it with a string. The most time the string operation takes is the length of the string. Meanwhile, cumulatively, the reducers will iterate over all **n** values, hence, the operation is **O(mnp)** where **p** is the number of iterations.

Ultimately, time is dominated by the **O(xy**) operation of the reducer in **Expand** because the **O(mn)** operations are essentially linear. This is because string manipulation is relatively cost efficient compared to the other operations. Finally, the number of records of the dataset is most influenced by the reduce operator of the **Expand** program.

Q7 Results

The experiment for Q7 involved comparing the MapReduce program to both Q7 and Q8 implementations in Neo4J. We have decided to include Q8 since it allows a user to specify a number of stops to search within, which is analogous to the iterations in the MapReduce solution. Q7 from Neo4J uses their allShortestPaths function while Q8 from Neo4J uses an expandConfig function that grows paths using BFS. The essential difference between Q7 and Q8 is that Q7 only searches for the shortest paths it finds, while Q8 expands its search level by level and includes all paths that start and end from the desired nodes.

For the Q7 comparison, we used San Francisco as the source location and New York as the destination location. The Neo4J implementation took 1.8215 seconds to run and returned 271 paths. Meanwhile, the MapReduce side took 10.527 seconds and returned 1078 routes. Note- *the MapReduce program returned paths of length two and three (both non-stop and one-stop flights)*. Once again, the MapReduce program is slower than its Neo4J counterpart. This time, the Neo4J implementation is 82.7% more efficient timewise. But if we consider time per path produced, then it's only 31.17% more efficient than the MapReduce solution.

The Q8 comparison was used to test the efficiency and limits of both programs. We used Seattle as the source city and New York as the destination city. Iterations / stops were increased until the programs produced undesirable results. In the MapReduce program, iterations could only be increased to two. The third iteration was never finished since it lasted over an hour and took up over 50GB of space, at which point we terminated the test since it long passed the feasible point for the application. Meanwhile, the Neo4J implementation performed well for 0 and 1 stops and lost signal to our AuraDB instance for anything over 1 stop due to no data left. We surmise that the BFS had visited all nodes which is why more than 1 stop caused the Python driver to lose connection to the database instance. For 0 stops, Q8 in Neo4J produced 5 routes in 1.4191 seconds. For at most 1 stop, Q8 in Neo4J produced 766 routes in 3.7416 seconds. This is comparable to MapReduce in its first iteration which covers direct and indirect flights of at most one stop. The MapReduce program produced an equal number of paths (including 5 direct flight paths) in 10.409 seconds. For the same volume of output paths, the Neo4J implementation was 64.05% more efficient. When increasing stops to two for Neo4J, it lost connection to the database and increasing the stops anymore led to the same result. In the case of MapReduce, a program that went through its first and second iteration took 41.932 seconds and created 136,639 paths. It is likely that many of the paths are duplicates, but we are unsure as to how many. Any further iterations beyond 2 caused the MapReduce program to produce unmanageable results.

Conclusion

Inspired by Ryczkowska's and Nowicki's demonstration of MapReduce and PCJ performance, the Data Keepers have been able to complete our analysis and distinguish the capacities and differences of

MapReduce and Neo4J. We originally set out to create an application that provides useful travel information and statistical information for users. Our Neo4J application has been able to answer the ten questions we proposed to answer. In addition to using Neo4J, we sought to create custom MapReduce algorithms to compete with our Neo4J implementation to improve query speed performance. These custom implementations were for questions 1 and 7.

Our experiments demonstrate that our custom algorithm for question 1 *did not out-perform* our Neo4J implementation in speed. In fact, the efficiency of the MapReduce algorithm was a little over 50% less efficient than the Neo4J implementation. There are options to increase the run-time, such as adding more nodes to Hadoop instead of running on a local device. Further testing needs to be done to decide whether we can count the MapReduce program out or not. Neo4J supports a limited amount of parallel computation. Therefore, it is not built for the same scalability that MapReduce programs are.

Question 7 in the MapReduce program is limited to up to two iterations. Attempts at additional iterations cause the program time to go beyond an hour which is infeasible. Additionally, the output grows beyond 50 GB. The first iteration of the MapReduce program produces both direct and indirect paths compared to Q8 of Neo4J which only produces direct paths for 0 stops. In this regard, the MapReduce program is less targeted. Add that to the fact that it takes longer than the Neo4J implementation, then it becomes clear that the MapReduce program needs some improvement.

To combat this, we have come up with a few improvements for Q7 in MapReduce. Firstly, in **CityList** there is an opportunity to combine a conditional such that it is a compound conditional. This will reduce code size and, in this case, increase readability. Then, in the reducer of **Expand**, we already know that the adjacency triples are all valid for the paths in the path **ArrayList**, thus the conditional checking for validity can be removed. *This will not have a significant impact on the runtime*, but still needs to be addressed since it serves no purpose. In the Reducer of both **Translate** and **Expand,** a string is created via the concatenation of a series of strings. Strings are immutable, so using a StringBuilder instead will increase the efficiency of creating these new strings. Using a StringBuilder is an especially important improvement since strings only grow per iteration. The largest improvement that can be made is using a **CustomWritable** as used in the related works report, “Performance comparison of graph BFS implemented in MapReduce and PGAS programming models”. The authors reported that using a class to represent a vertex instead of **Text** was the faster option. Additionally, using a node data structure (**CustomWritable**) means that our application would gain the ability to traverse unrepeated paths. By implementing a better BFS, we would have a better comparison to Neo4J’s BFS solution. Additionally, further iterations of MapReduce would have diminishing returns after a certain point since a path could only visit a node once. Given these context, constraints and recommended solutions, Data Keepers can confidentially hold a neutral recommendation for the usage of Hadoop MapReduce or Neo4J for future students and scientists to complete their big data analysis, solutions, and applications.