Vrije Universiteit Amsterdam

Faculty of Sciences,
Department of Computer Science

vrije Universiteit amsterdam

Per Olav Høydahl Ohme, student no. 2530033

Reducing Memory Requirements for Distributed Graph Query Executions in Lighthouse

Master's Thesis in
Parallel and Distributed Computer Systems

Supervisor:
Prof. Dr. Peter Boncz, Vrije Universiteit Amsterdam, Centrum Wiskunde & Informatica

Second supervisor:
Dr. Spyros Voulgaris, Vrije Universiteit Amsterdam

Second reader:
Claudio Martella, Vrije Universiteit Amsterdam

Amsterdam, July 2016

Abstract

Lighthouse is a graph compute engine that has been developed to execute high-level Cypher queries on very large datasets. By utilizing the Apache Giraph framework, the graph compute engine is able to run in a distributed and parallel manner. The core functionality of Lighthouse enables subgraph pattern matching, based on a user provided query, with any given input graph. During a query execution, Lighthouse has the opportunity to apply various optimizations depending on the properties of the given query and input graph, and the available hardware. This project has focused on how to reduce the required memory for executing query plans, allowing smaller Hadoop clusters with limited memory resources to run more communication-heavy subgraph pattern matching jobs.

Significant reductions of memory requirements have been achieved by reimplementing Lighthouse as a pipelined compute engine. In fact, any left-deep query plan can now be executed without Out-of-Core disk access, provided that the input graph fits into the available memory. Prediction of accurate numbers for message production in bushy query plans has shown to be difficult, often resulting in multiple execution restarts before a query execution is successfully finished. In addition to pipelining, several other memory optimizations have been evaluated to give beneficial outcomes. These optimizations span from changes for when to start computations on paths in bushy query plans to improved data structures and serialization formats for input graph data and messages.

Contents

List of Figures

Listings

Chapter 1

Introduction

This chapter presents a context for this thesis project. First, in section 1.1, the appeal of graph data structures is discussed and examples of how graphs can be used in multiple fields of industry and research are given. An overview of advantages gained by utilizing graph processing frameworks is then presented in section 1.2. Next, in section 1.3, both the design and performance of Lighthouse are explained together with memory consumption problems which may arise for certain types of queries and input graphs. A solution using Out-of-Core Giraph for memory-limited executions is presented in section 1.4, followed by an introduction to an alternative solution utilizing pipelining in section 1.5. The research questions which constitute the basis for this project are in section 1.6.

1.1 Graph-Structured Big Data

The fundamental flexibility of graphs enables them to represent a large variety of structures. Several ubiquitous structures can be modeled as graphs, such as transportation systems, social relations, the Web, disease outbreaks and DNA molecules. When a structure is modeled as a graph, the related analysis can benefit from previous development of efficient algorithms for general computing problems. Among others, for finding shortest paths, community detection and quality rating.

Large-scale processing of graphs is already common in multiple fields of industry and research. Leading IT companies are relying on analysis of large graphs to provide advanced services for their users. Social networks, for example Facebook and LinkedIn, require a proper understanding of present social relations to propose creation of new ones to their users. Web search engines, such as Google Search and Bing, depend on ranking of Web sites to provide relevant results to handled user queries. DNA sequencing is an example of graph processing for research purposes. It involves shearing of DNA molecules into fragments and assembling of fragments into genomic sequences.

1.2 Graph Processing Frameworks

The development of an application that needs to perform large-scale graph analysis, from scratch, can quickly turn into a time consuming effort. To be able to process huge graphs gracefully, many potential pitfalls need to be considered and some countermeasures must likely also be implemented. To achieve better computation times, multiple CPU cores can be utilized with parallel graph algorithms. These are usually a lot more complex than their sequential counterparts, often involving work distribution and synchronization. Larger graphs might not fit into the available memory of a single machine. This can be handled through use of attached disks and support for serialization and de-serialization (occurring when offloading or fetching graph partitions to or from the disks). The application should restrain its number of disk accesses to prevent long computation times. Problematically, some graphs are too large to be practically stored and processed on one machine. This creates the need for a distributed computing environment, which introduces problems like when and how to perform network communication and more machines that can possibly fail.

By utilizing a graph processing framework and its corresponding computing model, a programmer can reduce own development effort for a graph processing application to just involve the implementation of graph processing logic. Different types of existing frameworks offer various functionality. Graph databases, a common type of graph processing frameworks, provide create, read, update and delete methods for working with stored graphs[13]. Using transactions, they protect the integrity of stored data while allowing multiple queries to be executed simultaneously. Neo4j[10] is a popular graph database that supports high-level graph queries written in Cypher (described in appendix C). This language is declarative and simplifies the specification of graph processing jobs. Apache Giraph (described in appendix B) is another popular framework which enables distributed graph processing. Differently, it requires users to write applications in Java and to consider the Pregel computing model (described in appendix A). A Giraph application that can run on a single machine can also run on a large cluster without any changes to its related code. Before a Giraph job is started, the input graph is loaded from attached disks into the memory of the utilized workers.

1.3 Lighthouse

Lighthouse is built on top of Apache Giraph to enable graph processing in a parallel and distributed environment. This type of environment is needed to complete analysis of enormous graphs within a reasonable time. The fundamental purpose of Lighthouse is to perform large-scale graph pattern matching. More specifically, to find subgraphs of the input graph which are isomorphic to a graph representing the provided query. Instead of requiring users to implement graph processing behavior in Java, as Giraph, Lighthouse allows users to specify graph queries with the Cypher language. This simplifies users' work of expressing large-scale jobs to be executed. Lighthouse automatically starts a Giraph job based on a given query, an input graph and some configurations. The engine can utilize numerous optimizations to improve the performance of all carried out pattern matching. Which optimizations are to be applied for an execution, depends on the properties of the job and the available hardware. The choice of optimizations is for simplicity kept hidden from the user.

A provided Cypher query is used by Lighthouse to generate an in-memory query plan. When the original version of Lighthouse[6] runs, this plan is executed without taking any job or hardware properties into consideration. A query plan has the structure of a tree, with each node representing an operator from the Lighthouse algebra. The solutions from an operator is the input to its parent. Executions start on the leaf operators which filter initial messages on all the input graph vertices.

Listing 1.1: Example Cypher Query

```
MATCH (p1:Person{firstName:"Antonio"})-[:WORKAT]->(company)-[:IS_LOCATED_IN]->(country),
      (p2:Person{firstName:"John"})-[:WORKAT]->(company)
WHERE p1.browser = {"Chrome"} AND p2.browser = {"Chrome"}
RETURN p1.id, p1.firstName, p2.id, p2.firstName, company.id, country.id
```

Listing 1.2: Query Plan for Example Cypher Query

```
StepJoin(PathJoin(StepJoin(Project(Select(Scan(firstName:"Antonio"),
                                          =({browser}, Chrome)),
                                   [$1, {firstName}]),
                           WORKAT),
                  StepJoin(Project(Select(Scan(firstName:"John"),
                                          =({browser}, Chrome)),
                                   [$1, {firstName}]),
                           WORKAT)),
         IS_LOCATED_IN);
```

The Cypher example query in Listing 1.1 finds pairs of Chrome users, named Antonio and John, working for the same company. A relationship for the location of a company must exist to return matches with its corresponding Chrome user pairs. In addition to returning IDs for vertices in matched subgraphs, the first name of each Person in a match is also returned. Lighthouse creates the query plan in Listing 1.2 based on the example query. This plan is visualized in Figure 1.1, which also shows when the different operators are computed. A more detailed explanation of the query plan and the related computations is given after a presentation of the Lighthouse algebra.

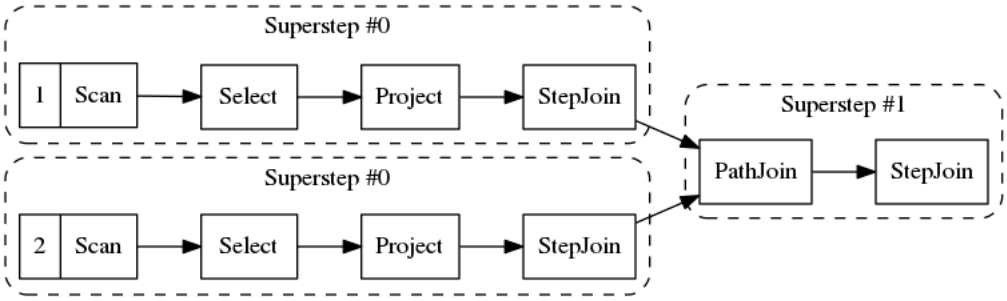


Figure 1.1: Visualized query plan for the example Cypher query

1.3.1 Lighthouse Algebra

There are two types of Lighthouse operators which have different effects on Giraph job executions. Global operators often lead to communication between the workers in an execution, while local operators do not. Since the Pregel computing model forces workers to process transferred messages in the superstep after they are sent, global operators introduce additional supersteps to executions.

A local operator causes messages to be passed to an operator that is computed on the same vertex.

Scan (first operator of every query path)
Filters solutions based on labels and properties of the current vertex.

Select
Filters away solutions that do not meet a condition.

Project
Retrieves information stored by the current vertex (other than vertex ID) or trims messages.

PathJoin
Joins pattern graphs of solutions which are received from different query paths.

A global operator causes messages to be passed to an operator that is computed on other vertices.

Move
Moves the computation from the current vertex to a previously visited vertex.

StepJoin
Filters solutions based on labels and properties of edges with the current vertex as source.

With this information about the Lighthouse operators, it is simpler to understand how the plan in Listing 1.2 (visualized in Figure 1.1) represents the Cypher query in Listing 1.1. The execution of the query plan starts on the leaf operators. The query paths are computed in sequence and are similar except for the first name they use in their filtering on Scan. In query path 1, Scan passes through vertices with the *firstName* property value "Antonio". In query path 2, Scan passes through vertices with the *firstName* property value "John". When a vertex is "passed through" by an initial Scan operator, a solution message is passed to the subsequent operator (by calling the compute method of its corresponding QueryItem object). The Select operators filter away solutions on vertices that are not representing Chrome users. All solutions meeting the Select condition are passed to a following operator. When a message reaches any of the Project operators, the *firstName* value of the current vertex is added to its solution. The message is then passed to the next operator. As a non-last global operator, each StepJoin passes the computation to the vertex representing the company which the Chrome user works for. The computation is passed by transferring a created solution message to the worker that is responsible for the relevant company vertex. After adding a message to its network send buffer, a worker starts to compute the other Scan operator of the query plan for either the same or another vertex. These computations continue until all vertices have been processed in the first superstep. In the next superstep, on vertices for companies with matched users, pairs of Johns and Antonios are created with the PathJoin operator. For each message with a Chrome user pair passed from the PathJoin operator, a complete solution is written to the HDFS if the last StepJoin finds a relationship to the country in which the related company is located.

In the first superstep of a query plan execution, the Lighthouse compute method in Listing 1.3 is called once per vertex stored on a worker. At this stage, no messages are passed via the *messages* argument. For each query path, the compute method of the path's first QueryItem is called with a created initial message as an argument. In all non-initial supersteps, the Lighthouse compute method is called once per active vertex stored on a worker. For each of these calls, at least one message is passed via the *messages* argument. The messages trigger further compute method calls.

Listing 1.3: Lighthouse compute()

```
public void compute(Vertex vertex, Iterable<Message> messages) {
    <Get worker context used to check query plan and write complete solutions>
    if (getSuperstep() == 0) {
      <Create initial messages for all paths of the query plan>
      <Pass computation for each created initial message to first query item of path>
    } else {
      <Pass computation for each received message to next query item of path>
    }

    <Vote to halt>
}
```

The compute method in Listing 1.4 is executed each time a message is passed to a Select operator. Specific for Select compute is that it checks whether a Select condition is valid for the handled solution. This is done by considering data of the received message and the current vertex. As seen in the last part of the outer if-body, a valid solution which is not yet complete is directly passed to the parent operator. This behavior is similar for every local operator.

Listing 1.4: Select compute()

```
public void compute(BasicComputation computationClass, Vertex vertex, Message message) {
   <Get worker context used to check query plan and write complete solutions>

   if (<Select expression is valid for this solution>) {
       <Increment path step for message>

       // Check whether the solution is complete
       if (message.getStep() == workerContext.getNumberOfSteps(message.getPath())) {
           <Write complete solution to HDFS>
       } else {
           <Pass computation to next query item of path>
}}}
```

The compute method in Listing 1.5 is executed each time a message is passed to a StepJoin operator. Specific for StepJoin compute is that it checks whether a StepJoin condition is valid for any of the current vertex' outgoing edges. As seen in the last part of the outer if-body, a valid solution which is not yet complete is sent to the target vertex of the related edge. This involves a message to potentially be transferred to another worker and to be processed by the parent operator. All global operators pass non-complete solutions to other vertices in the same manner.

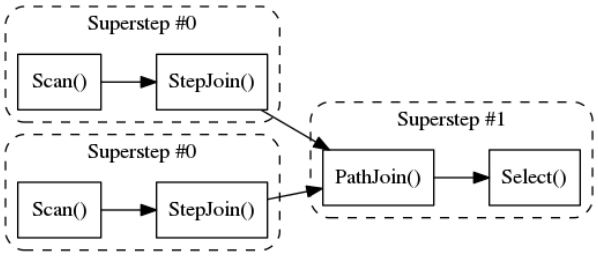
Listing 1.5: StepJoin compute()

```
public void compute(BasicComputation computationClass, Vertex vertex, Message message) {
   <Get worker context used to check query plan and write complete solutions>

   // Find edges satisfying the StepJoin condition>
   for (Edge edge : vertex.getEdges()) {
       if (<Edge satisfies the StepJoin condition>) {
           <Create binding for new message>
           <Prepare message with new binding>

           // Check whether the solution is complete
           if (newMessage.getStep()
               == workerContext.getNumberOfSteps(newMessage.getPath())) {
               <Write complete solution to HDFS>
           } else {
               <Send message to target vertex of edge>
}}}}
```

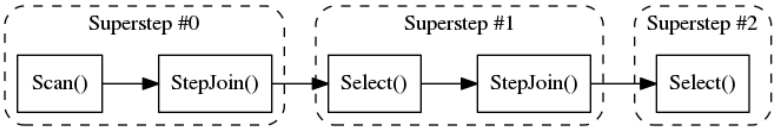
Since the various vertices of an input graph can be stored on different workers (distributed based on hash partitioning), network communication must often be performed to build solutions which include several vertices. The Pregel computing model requires that the production of these solutions involves multiple supersteps (see appendix A for a more detailed overview of how the Pregel computing model works). In executions of the query plan in Listing 1.2, the StepJoin operators of the two query paths transfer solution messages with information about Chrome users to vertices which represent their related companies. This means that even though vertices for Chrome users in the same company are stored on different workers, the PathJoin operator can create solution messages with pairs of these users on the worker which stores the company vertex.

1.3.2 Left-Deep and Bushy Query Plans

Query plans which do not have any operator with more than one child are called left-deep, while the others are called bushy. For left-deep query plans, one can interpret the input of each operator to be solution messages passed from its left incoming path. Since PathJoin is the only available binary operator in Lighthouse, all bushy query plans must contain at least one PathJoin operator.



(a) Bushy query plan



(b) Left-deep query plan

Figure 1.2: Bushy query plan requiring a smaller number of supersteps

For executions with undirected graphs, PathJoin operators can be utilized to increase concurrency and reduce the number of required supersteps. A bushy query plan visualized with Figure 1.2a can match the exact same patterns as a left-deep plan visualized with Figure 1.2b. As an example, the StepJoin operators can send messages between vertices representing persons having friend relationships. Since the friend relationships are mutual, it does not matter in which direction the solution messages are sent. Either of the showed query plans can therefore be used. Other left-deep plans have corresponding bushy plans that reduce the number of needed supersteps even more. PathJoins store all received messages in join tables. They should therefore be used with caution, as available memory after loading the local partition of the graph may be filled up with join tables' content. In worst case, no memory will be available for storing messages to be sent or received.

With directed graphs, many patterns can only be found using bushy query plans. As an example, one is looking for pairs of persons named Antonio and John knowing a specific celebrity in a graph with the following relationships: Antonio and John know the celebrity, but the celebrity does not have a relationship to any of them. A bushy query plan also visualized with Figure 1.2a can be used to successfully find this pair. One of the Scan operators passes through vertices with the *firstName* property value "Antonio", the other passes through vertices with the *firstName* property value "John". The StepJoin operators send messages to vertices representing known persons, while the Select filters away solution messages which are not processed on the specific celebrity's vertex. For a left-deep query plan to be able to match the described pattern, the celebrity would need to have an "is known by" relationship to Antonio or John, forming a chain of relationships with the same direction. A query plan visualized with Figure 1.2b could in such case be used.

1.3.3 Messages in Lighthouse

Messages are in this thesis often referred to as solution messages due to their content. Every message in Lighthouse stores the current state of a single attempt to perform a pattern match. Listing 1.6 shows the Java data structure that is used to hold message data. It implements the interface Writable to enable corresponding objects to be serialized and de-serialized. The *path* and *step* variables of a message specify which operator in the query plan should next continue the pattern matching attempt. A message is passed to the QueryItem object for the next operator via Lighthouse compute or another QueryItem object's compute method. The *binding* of a message contains all gathered information for the related solution. This data may be needed during later computations or for being written as part of the complete solution output for a successful pattern match. The binding structure is inspired by tables from the database world. Its content can be interpreted as a tuple of a database table, with values on indices corresponding to column numbers. The values themselves depend on the computed parts of the query plan and the visited vertices.

Listing 1.6: Message Data Structure

```
public class Message implements Writable {
    private byte path;
    private byte step;
    private MessageBinding binding;
    ...
}

public class MessageBinding
    extends ArrayList<Writable> implements Writable, Configurable {
    ...
}
```

An encountered operator passes messages based on the data of the current vertex, the binding in the received message and its own parameters. The binding of a message to be passed is built by altering the binding of the message which was received. Here is a list of the Lighthouse operators and how bindings in their messages passed are different from the binding in a received message:

Scan
Adds a column with the current vertex' ID to the message binding.

Select
Does not alter the message binding at all, but performs filtering.

Project
Adds columns with values for specified properties or removes columns.

PathJoin
Creates joined bindings with bindings from the opposite path.

StepJoin
Adds a column with the target vertex' ID for an edge which meets a condition.

Move
Moves a specified column to the last position in the message binding.

Following is an explanation of the message bindings constructed during an example execution of the query plan from Listing 1.2. The tables 1.1, 1.2 and 1.3 show the bindings created by the last computed operators in the two supersteps of the execution. The visualization of the query plan in Figure 1.1 gives an overview of which parts of the plan's paths are computed in superstep 0. Here are the listings for the corresponding query plan components:

Listing 1.7: Query Plan for Example Cypher Query: Part of Path 1 Processed in Superstep 0

```
StepJoin(Project(Select(Scan(firstName:"Antonio"),
                        =({browser}, Chrome)),
                 [$1, {firstName}]),
         WORKAT)
```

Listing 1.8: Query Plan for Example Cypher Query: Part of Path 2 Processed in Superstep 0

```
StepJoin(Project(Select(Scan(firstName:"John"),
                        =({browser}, Chrome)),
                 [$1, {firstName}]),
         WORKAT)
```

The tables 1.1 and 1.2 contain the bindings of messages which are produced by respectively computing the query plan components of listings 1.7 and 1.8. The values in the first column of each of the tables are set by the related Scan. On path 1, the ID of every vertex having the *firstName* property value "Antonio" is added. On path 2, the ID of every vertex with the *firstName* property value "John" is added. The Select operators filter away the bindings for vertices which do not represent people using Chrome. The Project operators create a new column by adding the *firstName* property value stored by the vertex of each binding. Next, the StepJoin operators create another new column by adding values of vertex IDs for companies that encountered people work for.

Table 1.1: Binding Table for Messages Produced by First StepJoin of Path 1

$1	$2	$3
136	"Antonio"	1236
8	"Antonio"	1700
758	"Antonio"	1587

Table 1.2: Binding Table for Messages Produced by StepJoin of Path 2

$1	$2	$3
522	"John"	1587
312	"John"	1300
18	"John"	1587

Table 1.3: Binding Table for Messages with Complete Solutions

$1	$2	$3	$4	$5	$6
758	"Antonio"	522	"John"	1587	10031
758	"Antonio"	18	"John"	1587	10031

Table 1.3 contains the bindings of messages which are produced by the last StepJoin operator, each representing a complete solution. The table also shows how bindings from the tables 1.1 and 1.2 are joined by the PathJoin operator. On the vertex with ID 1587, representing a company, one binding holding the name "Antonio" is received via path 1, while two bindings with the name "John" are received via path 2. This leads to the creation of two messages after joining all bindings from each path with all from the other. The last StepJoin operator creates a new column by adding values of vertex IDs for countries where encountered companies are located. In this case, the vertex ID for the same country is added for both the bindings, since they involve the same company.

1.3.4 Performance and Scalability

The performance and scalability of the original version of Lighthouse are here considered with regards to the needed time and memory consumption for executions.

Execution Times

Various types of operations are performed during a Lighthouse execution, each with an impact on the total execution time. Before any computations are started, disk overhead is introduced when the input graph is loaded from disks into the memory of the workers. The execution can become more time consuming if large amounts of output are written to disks. When enabled, Out-of-Core Giraph (presented in section 1.4) and checkpoints cause additional time consuming disk operations.

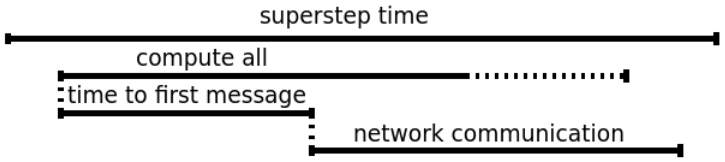


Figure 1.3: Execution of a Giraph superstep

The amount of time needed to finish a superstep in Lighthouse depends on the carried out executions of its three contained components: computation, communication and barrier synchronization. As visualized in Figure 1.3, the executions of these components overlap during a Giraph superstep. The computation component is executed in the period *compute all*. It includes the time needed to finish the compute calls for all the currently active vertices and to write solution output to the available HDFS. The communication component is executed in the time span *network communication*. It lasts between when the first message and the last message are flushed to the network. The time it takes to compute the whole superstep is represented with *superstep time*. It also includes the time needed for barrier synchronization. The *time to first message* period lasts between when the first computation in the superstep starts and the first message is flushed. Giraph can measure and provide the length of the mentioned time spans if the `giraph.metrics.enable` option is set. These numbers can be used for an in-depth analysis of Lighthouse's performance and scalability.

Listing 1.9: Query Plan for Measurement of Lighthouse Execution Times

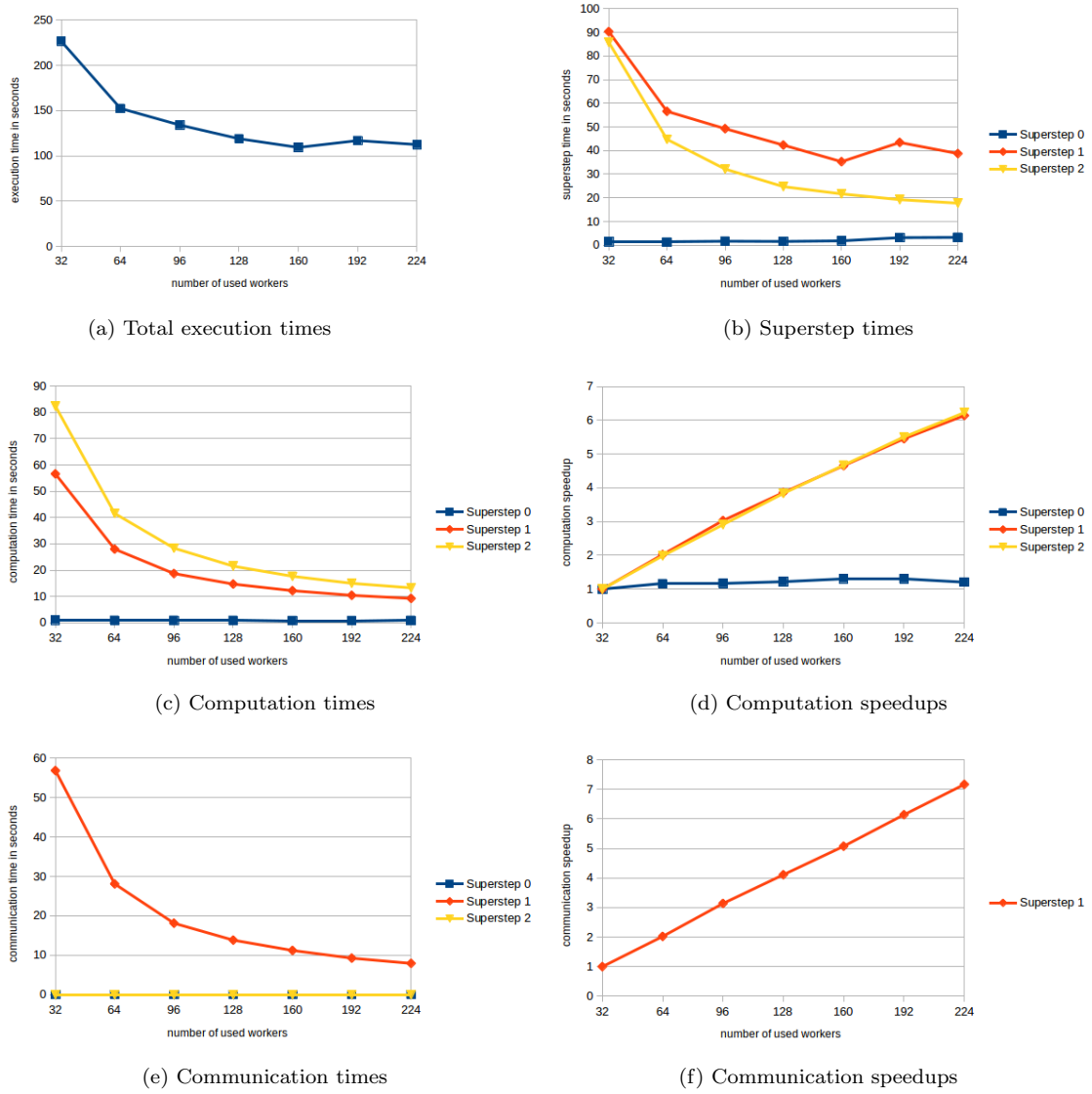
```
Project(Select(StepJoin(Project(Select(StepJoin(Scan(Comment),
                                                HAS_CREATOR),
                                       =({gender}, male)),
                                [$1, $2, {firstName}, {creationDate}, {gender}]),
                        likes),
                =({browser}, Chrome)),
        [$1, $2, $3, $4, $5, $6, {content}, {creationDate}]);
```

Figure 1.4 includes multiple graphs showing the measured duration times and derived speedups for executions of Query Plan 1.9. The measurements were performed on the SURFsara Hadoop Cluster, using Giraph release 1.1.0 and the `hadoop_2` profile, with the LDBC 10K dataset as input. This experimental setup is more extensively explained in section 3.1 of the evaluation chapter.

The time needed to finish an execution of the query plan in Listing 1.9, with different numbers of workers, is presented in Figure 1.4a. Clearly, the total execution time does not scale very well. When using 160 workers instead of 32, the execution time is just halved despite having 5 times more workers. The time spent per superstep, with different numbers of workers, is shown in Figure 1.4b. The duration of superstep 2 scales close to linearly with the number of used workers. This pattern is not apparent for the duration of superstep 1. There are notably small duration times for superstep 0, despite that this superstep involves the processing of many more vertices than the others (since all the vertices of the input graph are initially in an active state). However, differently from the other supersteps, it does not involve any de-serialization of received messages.

Duration times for previously mentioned superstep components are analyzed to get an understanding of which operations reduce the scalability of the system. All presented speedups are calculated with durations relative to the ones for the execution with 32 workers. The computation time spent per superstep, with different numbers of workers, is presented in Figure 1.4c. The derived speedups are shown in Figure 1.4d. The communication time spent per superstep, with different numbers of workers, is presented in Figure 1.4e. The communication time speedups are shown in Figure 1.4f. Undoubtedly, the numbers reported by Giraph for communication time in superstep 0 are wrong. It must take more than 0 milliseconds to transfer the messages of superstep 0 among the workers.

The speedup graphs show that Lighthouse is scalable with regards to computation and communication time. Both of these metrics scale almost linearly with the number of used workers. Slow loading of graph partitions from the HDFS may be the bottleneck causing the bad scalability for the total execution time. Additionally, the long duration times for superstep 1 with many workers include unknown overhead, potentially caused by some synchronization issues. There are no disks being touched at this stage, and the corresponding computation and communication times are low.



(a) Total execution times

(b) Superstep times

(c) Computation times

(d) Computation speedups

(e) Communication times

(f) Communication speedups

Figure 1.4: Measured duration times and derived speedups for executions of Query Plan 1.9

Memory Consumption

The original version of Lighthouse does not take any precautions to ensure that query plan executions can be performed with limited amounts of available memory. This puts restraints on what query plans can be executed, and on size and properties of input graphs which can be processed. Lighthouse is currently only supporting read queries, causing an input graph to stay unchanged during an execution and to consume a constant amount of memory. Differently, the memory space that is used to store messages during an execution can vary in size. Query plans matching interesting patterns may lead to exponential growth in numbers of present messages or cause message bindings to vastly increase in size. Distributed subgraph pattern matching is inherently space complex, potentially causing workers to run out of available memory. Figure 3.1 in the evaluation chapter shows how messages in an execution may require more space than the handled input graph.

Several of the previously presented Lighthouse operators can contribute to an increase of memory required to store messages during an execution. When a Scan operator passes near the same number of messages as it receives, more space is needed to store the messages it passes than the messages it receives. This is caused by Scan adding a column with vertex ID to the binding of each message passed. A Project operator can similarly add multiple columns to message bindings, but without filtering away any solutions based on a condition. The added columns can store any types

of data, potentially of huge sizes. The number of messages in the system can increase exponentially through computation of StepJoin operators. For every received message on a StepJoin, one message may be passed per outgoing edge of the processed vertex. A PathJoin operator might also lead to the need for more memory space. In addition to storing all received messages in join tables, it can pass a number of messages equal to the product of the numbers of messages received from its left and right incoming paths. The bindings of the passed messages are large, containing columns with data from bindings of both the PathJoin's incoming paths.

The number and length of paths in a generated query plan also play a role on the numbers and sizes of messages in its related executions. In the beginning of an execution, the number of initial messages created per vertex is similar to the number of paths in the generated query plan. This means that bushy plans with many paths lead to the creation of more initial messages. Long query paths will often cause the creation of bindings with many columns and a large number of messages.

Following is an example showing how memory consumption can become a problem with StepJoin:

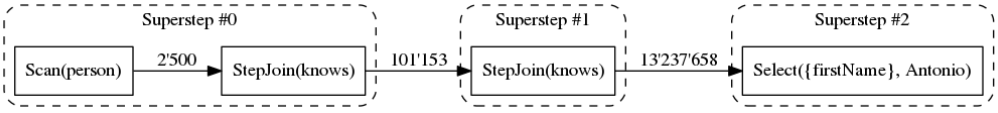


Figure 1.5: Query plan execution in which increasingly many messages are produced

The query plan execution visualized in Figure 1.5 finds every person who knows someone knowing a person with the first name Antonio. All the numbers for messages passed by the operators are taken from a worker after an execution with 4 workers using the LDBC 10K dataset as input. On both the computed StepJoin operators, each active vertex passes one message per outgoing 'KNOWS' relationship for every received solution message, giving a significant increase in number of present messages. The first StepJoin in superstep 0 receives 2'500 messages and leads to 101'153 messages being sent. Subsequently, the StepJoin in superstep 1 causes more than 13 million messages to be transferred. The sizes of the messages in the system increase for each computed non-last operator.

Here is an abstract overview of the memory consumption during the execution. The input graph is evenly distributed among the used workers, filling A percent of memory on each worker. There is a remaining B percent of memory available to store produced messages during the execution. When the Scan operator is computed, V number of vertices satisfy the Scan condition, each causing the first StepJoin to be computed. The StepJoin condition is satisfied for an average E outgoing edges per processed vertex, causing a total of $V*E$ number of messages to be sent. If E and the sizes of the sent messages are too large, then B percent of memory will not be sufficient for storage. The last StepJoin creates more messages, increasing the possibility of running out of memory.

Following is an example showing how memory consumption can become a problem with PathJoin:

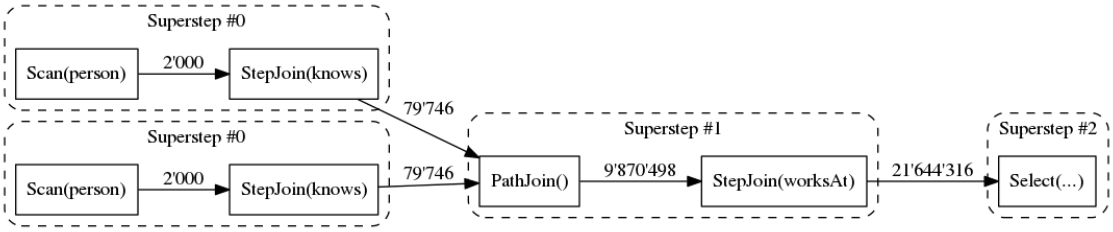


Figure 1.6: Query plan execution in which messages are stored in PathJoin tables

The query plan execution visualized in Figure 1.6 finds pairs of people knowing the same person who works for a specific company. All the numbers for messages passed by the operators are taken from a worker after an execution with 5 workers using the LDBC 10K dataset as input. Each of the first StepJoin operators in superstep 0 leads to a total of 79'746 messages being sent. In the next superstep, messages are joined on the PathJoin and a total of 9'870'498 messages are passed to the following StepJoin on the worker. This operator passes 21'644'316 messages, all which

are transferred via the used network. Again, in addition to the increasing number of messages produced, the message sizes increase for every computed non-last operator.

Once more, an abstract overview of the memory consumption during an execution is presented. The input graph is evenly distributed among the used workers, filling A percent of memory on each worker. The remaining B percent of memory is available to store produced messages. After the first StepJoin operators are computed, some vertices receive messages and compute the PathJoin. For each of these vertices, *m1* messages received from the left path are joined with each of *m2* messages from the right, causing *m1*m2* messages to be passed. The total number of messages which are passed by a PathJoin is in the most extreme case equal to the product of the total numbers of messages received from the paths on the worker, *M1* and *M2*. The B percent of available memory for messages can quickly be overflowed with the *M1+M2* messages stored in join tables and with the messages to be passed or received on the following non-last global operators.

1.4 Running Giraph with Support of Disks

Apache Giraph is first and foremost designed to execute graph processing jobs in-memory. With default configurations, the framework will before starting any computations split the provided input graph and load its partitions into the memory of the utilized workers. Disks are only touched for writing of output during an execution. With limited memory resources, Giraph may run into two different problems for jobs that require large amounts of available memory:

1. the input graph is too large to fit into the available memory
2. the input graph fits into the available memory, but the produced messages do not

It is difficult to in advance of any computations predict whether all created messages will fit into the available memory. This is highly dependent on the behavior of the algorithms to be executed and the input graph, which impact the number of messages to be stored and their sizes.

As a solution to the problems mentioned above, Giraph offers support for offloading data to the local disks of the used workers. This feature is called Out-of-Core Giraph[12]. Through splitting the input graph into more partitions, only having some partitions in memory at the same time, jobs with larger graphs can be executed. By using a limit for the number of messages each worker can keep in memory simultaneously, temporarily storing additional messages on disk, jobs involving more messages can be executed. The disadvantage of using disk operations is that they are slow.

This project explores alternative solutions reducing the memory requirements of Lighthouse query plan executions. The time-consuming disk operations in Out-of-Core Giraph for messages should be avoided. With use of pipelining (extensively explained in the next section), eliminating the need for storing messages on disks, only messages to immediately be processed are created. Unfortunately, pipelining introduces more supersteps and synchronization. Other memory optimizations are also explored, reducing the need for both Out-of-Core Giraph and Lighthouse with pipelining.

1.5 Pipelining

To perform a pipelined computation, the required processing must be split into a set of connected processing elements where the output of an element can be the input of another. It is essential that the computation produces correct output when the processing elements are executed in parallel. For Lighthouse, it is convenient to consider the operators of a generated query plan as processing elements. Pipelining is with dedicated hardware for the different processing elements often used to achieve a higher throughput when the input is a stream of data. Pipelining is in this aspect not relevant for Lighthouse, as every available CPU core can be used to compute all the present operators. Stream input is also not supported by Giraph, in which computations can only start after the input graph is completely loaded into the memory of the used workers. However, by enabling pipelining for Lighthouse, the memory consumption for many query plan executions can be reduced. This through limiting the number of vertices which can start processing query paths per superstep, at the expense of increasing the number of supersteps which are needed to finish.

In the original version of Lighthouse, only one global operator on each path of the generated query plan can be computed per superstep. With pipelining, multiple global operators on the same path can be computed in the same superstep, increasing the likelihood of all workers being busy.

Pipelining has already been used in database management systems for many years[7]. By passing tuples directly via an in-memory buffer, on requests of a subsequent operator, there is never a need to store intermediate tuples on disks. The alternative strategy, called materialization, involves storing all the output of an operator on disks before it is needed by another. This is generally in-efficient due to the large amount of required disk writes and reads. Unary operators, such as Select and Project, have simple pipelined implementations. Input tuples are one at a time fetched from an input buffer, then processed with results written to an output buffer. Binary operators, such as NaturalJoin and Division, have more variable and complex pipelined implementations. Each needing a single output buffer, but varying numbers of input and processing buffers.

An approach for pipelining with operators which pass one tuple at a time, does not lend itself well to distributed executions with Lighthouse. If a global operator only passes a single solution message per superstep, the related query plan execution may involve an extreme amount of supersteps to finish. This is a consequence of the potentially huge number of messages which are to be passed by a global operator, in addition to Giraph demanding a message transferred over the network to be processed by an operator in the subsequent superstep. The total network latency experienced for sending messages will also be high, since messages are not accumulated and sent over the network together. Lighthouse with pipelining should instead support to pass multiple messages from global operators per superstep, but in a controlled manner to prevent workers from running out of memory. The goal should be to keep memory consumption within the limits of the used hardware platform, while holding the number of needed supersteps to a minimum, avoiding synchronization overhead.

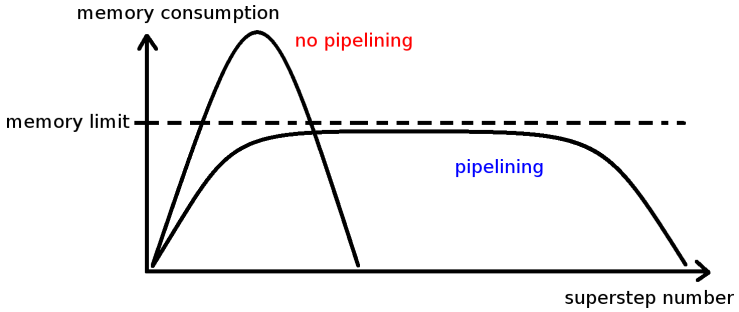


Figure 1.7: Possible memory consumption for query plan executions with and without pipelining

As visualized in Figure 1.7, pipelining is supposed to decrease the maximum memory consumption for Lighthouse query plan executions which require large amounts of memory. By increasing the number of used supersteps, the memory consumption should be controlled to stay well within the memory limit on the available hardware. Pipelining will be particularly useful for path queries (described in appendix D), which often result in both communication-heavy and memory-hungry query plans. For pipelined Lighthouse to be preferred over Lighthouse with Out-of-Core Giraph, the additional synchronization must be less time-consuming than the alternative disk operations.

1.5.1 Lighthouse Query Plan Executions Benefiting from Pipelining

Not all query plan executions can benefit from pipelining. Executions of query plans which contain only local operators can not. Ignoring join tables, they store very small amounts of messages at a time, since vertices are sequentially processed and messages are passed directly between operators. After a message is passed by a local operator, it is immediately handled by the subsequent operator. Many query plans with non-last global operators can benefit from pipelining. Non-last global operators cause messages to pile up in network send and receive buffers, which in worst case leads to executions running out of memory. Various query plans are presented, each with an explanation of whether its related executions can reduce maximum memory consumption with pipelining. In the following figures, LM is used as an acronym for “local messages” passed between operators computed on the same vertex. Similarly, GM is used for “global messages” passed between workers.

Executions of Left-Deep Query Plans

Following is an overview of different types of query plans and corresponding benefits of pipelining:

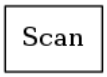


Figure 1.8: Query plan with a single operator

Executions of the query plan in Figure 1.8 can not be optimized to use less memory with pipelining. For each processed vertex, a final solution message may be created by the Scan operator. Instead of this message being passed to another operator, it is immediately written to the HDFS by the worker. The memory consumed by the output is after the write operation made available for reuse. Since multiple messages are never stored in memory at the same time, there is nothing to be gained by trying to control the amount of messages present during the execution.

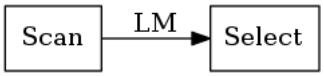


Figure 1.9: Left-deep query plan with multiple local operators

Executions of the query plan in Figure 1.9 can not be optimized to use less memory with pipelining. This is caused by the similar reasons as why the previously discussed executions can not be optimized. When a worker finishes a computation of the Scan operator with output, the solution message is immediately passed to the Select operator. Since Scan is a local operator, the Select is computed before the Scan is again computed on a next vertex. Messages created by the Select are right away written to the HDFS. Multiple messages are again never stored in memory at the same time, giving no opportunities to reduce the number of messages present at a time.

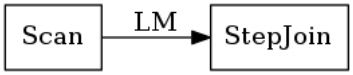


Figure 1.10: Left-deep query plan with a global last operator

Executions of the query plan in Figure 1.10 can not be optimized to use less memory with pipelining, despite the query plan containing a global Lighthouse operator. When the final StepJoin operator is computed, it creates a solution message per edge matching its condition. These messages are never further processed and each is immediately written to the HDFS after creation. Since multiple messages are never stored in memory at the same time, there are no benefits from pipelining.

Figure 1.11: Left-deep query plan with a single non-last global operator

Executions of the query plan in Figure 1.11 can be optimized to use less memory with pipelining. Since StepJoin is a global operator, the messages produced in a superstep by the non-last StepJoin can only be processed in the subsequent superstep. The messages are transferred to workers storing target vertices for matched edges, before they are provided as input for the Select operator. Until the messages are finally sent, they are stored in the network send buffers on each worker. While a worker sends solutions, messages from the StepJoin produced on other workers are simultaneously received and stored. Pipelining should enable control over the number of messages being processed in each superstep, allowing reduction of the maximum memory consumption.

Executions of the query plans in Figure 1.12 and Figure 1.13 can as well be optimized to use less memory with pipelining. Each StepJoin operator in the query plans may have the same effect on the number of messages in executions as the StepJoin of the query plan in Figure 1.11. With multiple StepJoin operators, the number of present messages in a system can grow exponentially.

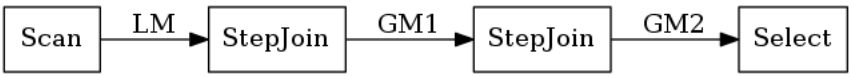


Figure 1.12: Left-deep query plan with two non-last global operators

Figure 1.13: Left-deep query plan with three non-last global operators

In all executions of the two above query plans, message sizes increase for every non-last operator which is computed. If the same input graph is provided for executions of both the query plans and the operators in the two query plans have similar parameters, the possibility of available memory being overflowed will be higher for the query plan in Figure 1.13. This is a consequence of it being the longest. Pipelining should again provide control over the number of messages to be sent and received per superstep, thereby also the maximum memory consumption.

Executions of Bushy Query Plans

All executions of bushy query plans generated by Lighthouse can benefit from pipelining. Executions of bushy query plans which exclusively contain local operators can not, but such query plans should not under any circumstance be generated. They are fundamentally uninteresting, since the output of their executions reflects how input graph vertices are distributed among utilized workers.

A PathJoin stores messages it receives during an execution in join tables of encountered vertices. All memory occupied by join tables is never made available again for storage of other messages. The amount of memory which can store messages from global operators decreases when join tables are filled, increasing the possibility for workers to run out of memory. Some query plans contain non-last global operators after a PathJoin, causing additional storage of messages to be sent and received. As previously explained, a PathJoin may pass many messages compared to the number of messages it receives. If a PathJoin is not followed by any non-last global operator, its output can still significantly affect memory consumption by related messages being stored in join tables. When an execution is expected to involve PathJoin operators with large input or which produce many messages, pipelining should be used to ensure completion with limited memory resources.

Following is an overview of different types of query plans and corresponding benefits of pipelining:

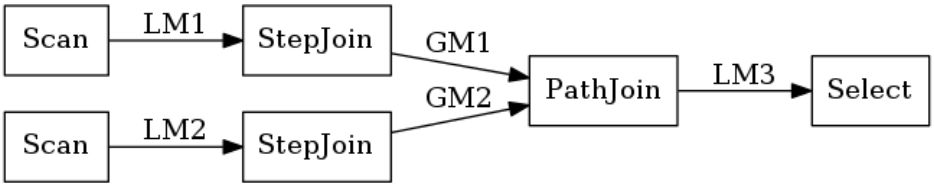


Figure 1.14: Bushy query plan with a single PathJoin not followed by non-last global operators

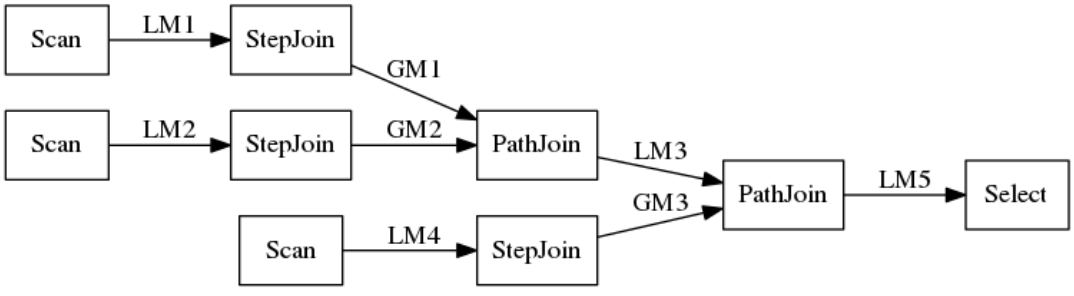


Figure 1.15: Bushy query plan with multiple PathJoin operators

Executions of the query plans in Figure 1.14 and Figure 1.15 can be optimized to use less memory with pipelining. The messages produced by each StepJoin in a superstep are passed among workers and processed in the subsequent superstep. This permits computed PathJoin operators to work on solutions with information from all vertices of the provided input graph. Since the query plans are not containing any non-last global operator following a PathJoin operator, final solutions can be produced in the same superstep as PathJoin output and be written to the HDFS. The messages passed by the first PathJoin in Figure 1.15 are stored in the join tables of the next PathJoin. Pipelining enables control over the numbers of messages that are handled by the StepJoin operators per superstep, allowing reduction of the maximum required memory for storage of messages.

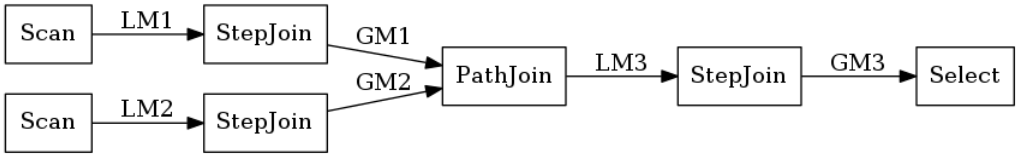


Figure 1.16: Bushy query plan with a single PathJoin followed by a non-last global operator

Executions of the query plan in Figure 1.16 can be optimized to use less memory with pipelining. Oppositely from the previously discussed query plans in Figure 1.14 and Figure 1.15, the query plan contains a non-last global operator which follows a PathJoin. The number of messages produced by the PathJoin has a significant impact on the memory consumption of related executions. The messages from the PathJoin are not written to the HDFS, but instead processed by the subsequent StepJoin, causing multiple messages to be stored in memory at the same time. As explained before, this gives opportunities to reduce the memory consumption with pipelining. Bushy query plans with late global operators are likely to give very memory-hungry executions with many messages.

1.6 Research Questions

1. How can pipelining be utilized to reduce memory consumption for query executions?
   - For which queries and input graphs can executions benefit from pipelining?
   - How should workers calculate initial message limits?
   - When should new initial message limits be calculated and used?
   - What measures can be taken to decrease damages caused by skew in given input data?
   - How can a pipelined execution which fails be restarted or continued?
   - What problems can occur for pipelined Lighthouse when binary operators are present?
   - How can the start superstep for each path of a query plan be set to improve performance?
2. How can used data structures and formats be changed to reduce required amounts of memory?
   - In what ways can data structures for input graphs and messages be improved?
   - Which data serialization formats should be used for input graphs and messages?
   - How can binary join operators of Lighthouse store messages more efficiently?
3. What other optimizations can be used to increase available memory for messages?
   - How can selective loading of vertices and edges from the input graph be utilized?
   - How does Lighthouse with pipelining perform compared to with Out-of-Core Giraph?
4. Which operators can with implemented optimizations still be considered bottlenecks?
   - For which queries and input graphs should these operators be avoided?
5. What memory optimizations should be enabled by default in Lighthouse?

Chapter 2

Lighthouse Changes

This chapter explains how Lighthouse is improved to require less memory for some executions. For each presented optimization, a conceptual description and details for how it is implemented are provided. In section 2.1, changes which enable Lighthouse with pipelining are in-depth explored, including how and when to calculate initial message limits. Different ways of reattempting failed executions are also considered. Next, in section 2.2, PathJoin related memory optimizations are discussed. They involve different approaches for when to start executions of paths in bushy query plans and various methods for storing messages in join tables. Finally, in section 2.3, improved data structures and serialization formats for vertices, edges and messages are presented.

2.1 Lighthouse with Pipelining

As made clear in the introduction chapter, pipelining can reduce Lighthouse's maximum memory consumption for executions of query plans containing non-last global operators. Lighthouse with pipelining is implemented by limiting the number of vertices on which query path computations are started per superstep. This enables fewer messages to be passed between the operators of a generated query plan per superstep, requiring less available memory to store messages to be sent or received on non-last global operators. Since all paths of a query plan must be computed for every input graph vertex before the related execution can finish, more supersteps may be needed when using pipelining. This introduces additional synchronization and communication overhead.

A query path computation is started on a vertex when the compute method of a path's initial Scan object is called with the vertex as an argument. The initiative to a path computation is taken after an initial message has been created on a vertex. It contains the path's ID and step number 0. As showed in Listing 2.1, the number of initial messages created per worker in each superstep is controlled by using the new variables *initialMessageCount* and *initialMessageLimit*. Both are available for all computation on vertices of a worker, via the worker's WorkerContext object. Each WorkerContext object is also used after a superstep is finished, when *initialMessageCount* is reset to 0, through an automatic call to its provided postSuperstep method.

Listing 2.1: Lighthouse compute() of Pipelined Implementation

```
public void compute(Vertex vertex, Iterable<Message> messages) {
    <Get worker context used to check initial message count and limit>
    <Get number of unstarted paths for vertex>
    if (workerContext.initialMessageCount < workerContext.initialMessageLimit
        && numUnstartedPaths > 0) {
        <Create initial messages for paths of the query plan>
        <Pass computation for each created initial message to first query item of path>
    }

    <Pass computation for each received message to next query item of path>

    <Vote to halt when all query paths have been started on the vertex>
}
```

The number of unstarted query path computations for a vertex is stored in the vertex itself. As seen in Listing 2.1, it enables a quick determination of whether new initial messages should be created in a superstep. The number of unstarted path computations for a vertex is also used to decide when it should be turned inactive, which is done by calling its provided voteToHalt method. If voteToHalt is called for a vertex before all its related path computations have started, the query plan execution might end without a complete set of final solutions. Since a vertex which has turned inactive may never be reset to active again, some path computations are possibly never started. If voteToHalt is not called for an active vertex in every superstep after all its related path computations have started, a larger number of supersteps may be executed. In this situation without receiving any messages, such active vertex can not be processed to create solutions, but force new supersteps to be started. It also introduces more computational overhead per subsequent superstep being active, even when other vertices are processed, constituting an additional unnecessary iteration step.

A 1 byte boolean *hasStartedAllPaths* can be used instead of the 4 byte integer *numUnstarted-Paths* in each vertex. However, this seldom reduces the memory consumption for storing vertices significantly, since all objects in Java are padded to have sizes in bytes being multiples of 8. An advantage of using the integer *numUnstartedPaths* is that it enables the creation of a number of initial messages per superstep exactly equal to the initial message limit. A boolean *hasStartedAll-Paths* can only restraint the creation of initial messages to multiples of the number of query paths. The addition of attributes to the vertex data structure has motivated changes to reduce the required memory for storing input graphs, which are presented in section 2.3. An approach providing control over the creation of initial messages per superstep, without adding more data to vertices, is to use a counter on every worker for the number of vertices on which path computation has started. The counter determines how many vertices must be iterated in a superstep before initial messages can be created. This approach utilizes that vertices are processed in the same order in every superstep, but denies vertices to turn inactive before all initial messages have been created, giving extra computation overhead. If an unknown number of vertices are inactive, it is impossible to determine for which vertices to produce initial messages. It is difficult to maintain a counter for the number of inactive vertices on each worker, since vertices are unpredictably reactivated when receiving messages between supersteps.

2.1.1 Calculations of Initial Message Limits

If the initial message limit on each utilized worker is set to a low value in the beginning of an execution, fewer messages are passed by non-last global operators of the query plan in every following superstep. This enables clusters with less available memory to be used, as smaller amounts of messages are stored the same time, but forces the execution to spend more supersteps to finish. The limits for the number of initial messages to be created per superstep should not be set too small, since this causes a need for unnecessarily many supersteps with associated overhead. If the initial message limits are set too large, workers in the execution may run out of available memory. The same limit values must not be used for all executions on a cluster, as various query plans and input graphs will cause different amounts of messages to be produced. Initial message limits should be intelligently calculated, considering properties of the handled query plan and the input graph, to enable any execution to succeed without requiring an excessive number of supersteps.

These numbers are used by a worker to calculate its optimal initial message limit in an execution:

- *available unused memory* after the local graph partition has been loaded
- *number of input graph vertices* stored in memory
- *ratio between the numbers of out and in messages* for each query plan operator
- *average size for serialized out messages* of each non-last global operator
- *number of in messages* from each path to every present PathJoin operator
- *average size for serialized in messages* from each path to every present PathJoin operator
- *number of currently active input graph vertices* for each present PathJoin operator

A worker in Lighthouse with pipelining is responsible for controlling its own initial message limit. This limit is never communicated, requiring the above numbers to have similar values on all sites. With significant differences, extra measures must be taken to avoid any worker being overloaded with messages passed. Since Pregel systems run on homogeneous clusters, all workers have physical memory of identical sizes. An input graph will before an execution be partitioned using a hash function, giving an even distribution of vertices, causing similar amounts of initial unused memory.

Each worker measures its *available unused memory* before superstep 0, from the preSuperstep method of its WorkerContext object, using Giraph's MemoryUtils class. Garbage collection must be requested to take place before all memory consumption API calls, to get accurate reports in return. Since garbage collection is a time consuming operation, it should be initiated as seldom as possible. An estimation for the *number of input graph vertices* stored on a worker is calculated through dividing its worker context's *getTotalNumVertices* value by the number of utilized workers. This takes place during the second superstep of a computation, after all workers have reported the number of vertices they iterated during the first superstep. Alternatively, the precise number of vertices stored on each worker can be found by including a counter in all WorkerContext objects. During the initial superstep, for every vertex processed locally, this counter must be incremented. The *ratio between the numbers of out and in messages* for each operator is calculated with statistics from already performed computation on a query plan. When an operator is used to process a message on a worker, the *inMessageCount* of its corresponding QueryItem object is incremented. Similarly, for each message produced and passed, the QueryItem object's *outMessageCount* is incremented. The QueryItem object of a global operator also stores the related *average size for serialized out messages*. This variable changes every time a different size is observed for a message passed. Figure 2.1 visualizes statistics gathered for operators of a query plan on a single worker during an execution. In this case, the average sizes for serialized in and out messages were stored for all encountered operators, not only for global ones, in addition to the out-in ratios for messages.

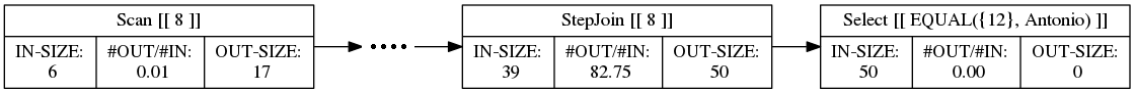
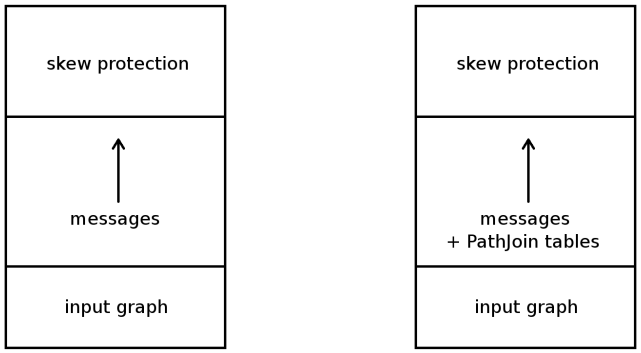


Figure 2.1: Statistics gathered on a worker during a query plan execution

All calculations of initial message limits during a query execution should be performed defensively, to decrease the risk of an imbalanced message distribution causing workers to run out of memory. In addition to the previously presented numbers, a calculation must consider a *skew protection fraction size*, ensuring that a part of the available memory is left empty when no skew is apparent. This memory is used as a buffer when a sudden large amount of messages is produced or received. Another advantage, of not utilizing all the available memory at once, is that less runtime overhead is introduced by the garbage collector. It is invoked more often on creation of new Message objects when the memory is nearly full. Potential message distribution imbalances caused by input skew in Giraph may be reduced through use of an alternative graph partitioning algorithm, for instance Spinner[9]. Figure 2.2 shows how memory of workers is consumed during executions. The amount of memory required to store an input graph's vertices and edges is constant, while the number and sizes of created and stored messages vary depending on the handled query plan and input graph.

In the beginning of a superstep, Giraph keeps received serialized messages in a message store on every worker. For each input graph vertex which is processed by a worker during the superstep, all related messages are sequentially de-serialized to a single reused Message object. This approach reduces the numbers of created Message objects and garbage collector invocations, in addition to the lifetime of received messages as Java objects. The messages to be passed by non-last global operators are oppositely serialized and added to the worker's local send buffers. They are eventually transferred via the used network. To accurately predict the memory consumption of a worker with an expected set of messages, all memory to be allocated by its data structures for serialized messages must be considered. This is implemented by multiplying the sum of anticipated sizes for serialized messages with 1.5. After several attempted test executions of query plans with little skew, this factor has been favored to ensure that all available memory except skew protection is consumed. Facebook has similarly implemented multiplication with 1.5 when estimating sizes of serialized input graphs, taking into account memory fragmentation and inexact byte array sizes[3].



(a) Left-deep query plans (b) Bushy query plans

Figure 2.2: Workers' memory during executions of left-deep and bushy query plans

General Calculation Details

Listing 2.2 contains the formulas used by an active worker to calculate its *initial_message_limit*. This limit represents the optimal number of initial messages which it should create per superstep. The formulas should be examined in the same order as they are listed in. Each shows how a new named value is calculated. Corresponding Java code is later explained, including a description of how QueryItem objects of a generated query plan are traversed while carrying out the calculation.

Listing 2.2: Overview of Initial Message Limit Calculation

```
skew_protection_mem = skew_protection_fraction_size * total_mem
predicted_max_mem_pathjoin_tables =
    SUM(predicted_total_num_in_left_path * in_left_avg_size * 1.5,
        predicted_total_num_in_right_path * in_right_avg_size * 1.5)
total_mem_messages =
    unused_mem - skew_protection_mem - SUM(predicted_max_mem_pathjoins_tables)

mem_messages_nonlast_global =
    predicted_superstep_num_out(initial_message_limit) * out_avg_size * 1.5
SUM(mem_messages_nonlast_globals) = total_mem_messages
divisor = total_mem_messages / initial_message_limit

initial_message_limit = total_mem_messages / divisor
```

skew_protection_mem represents the amount of memory which should stay unused on a worker when there is no input skew or message distribution imbalance in the computation. Its value depends on the *skew_protection_fraction_size*, a constant with default value 0.3. This value has proved itself to be appropriate for executions with smaller amounts of input skew, preventing the garbage collector to be invoked too often. If commonly executed queries and provided input graphs trigger greater imbalances, this value should be set larger. *total_mem* is the total amount of available memory to the worker, a value which is obtained via Giraph's MemoryUtils.maxMemoryMB method.

predicted_max_mem_pathjoin_table is the predicted amount of memory a specific PathJoin operator's tables will occupy at the end of the computation. PathJoin tables are gradually filled with messages and enable mergences of messages arriving in different supersteps. The predicted final memory consumption for the tables of a PathJoin is based on estimations of how many messages will in total arrive at the PathJoin during the computation, multiplied with average message sizes. If an incoming path does not contain other PathJoin operators, the number of messages to arrive from the path is estimated by multiplying the total number of initial messages to be created on the worker for the path, with the product of the out-in ratios for the path's operators before the PathJoin. If an incoming path contains other PathJoin operators, the number of messages to arrive from the path is estimated with calculations which are discussed in the next section.

total_mem_messages is the maximum amount of memory a worker should consume for messages on global operators when there is no input skew or message distribution imbalance in the computation. This amount does not include all the unused memory of the worker after loading its input graph

partition. One memory fraction is reserved for skew protection, while another is reserved for the messages to be stored in present PathJoin operators' tables. The amount of consumed memory after loading an input graph partition is obtained via Giraph's MemoryUtils.totalMemoryMB method.

mem_messages_nonlast_global is the predicted amount of memory required to store all the messages produced on a worker per superstep with a specific non-last global operator. This amount depends on the unknown value of *initial_message_limit*. For non-last global operators with an incoming path not containing any PathJoin operators, the *predicted_superstep_num_out* function can be expressed as *initial_message_limit * 1/num_paths * PRODUCT(ratios_from_path_start_to_with_nonlast_global)*. For non-last global operators with an incoming path containing PathJoin operators, the *predicted_superstep_num_out* function involves more complex calculations which are discussed in the next section. The predicted number of messages is multiplied with the stored average message size.

initial_message_limit is calculated by setting the sum of the predicted amount of memory needed to store the messages produced on a worker per superstep with each global operator, equal to *total_mem_messages*, the maximum amount of memory the worker should consume for messages on global operators. Since all the summands of *SUM(mem_messages_nonlast_globals)* contain the factor *initial_message_limit*, a division on both sides of the equation with this factor causes it to only be present on one side. The side without is in lack of a better name called *divisor*. Its value can be used to calculate *initial_message_limit* when already having the value for *total_mem_messages*.

Listing 2.3: Calculation of Initial Message Limit

```
private long calculateInitialMessageLimit() {
    // Calculate divisor for query plan
    double divisor = calculatePathDivisor(paths.get(0), true);

    // Avoid zero division in case non-last global operators are not present
    if (divisor == 0) {
        return Long.MAX_VALUE;
    }

    // Return the calculated initial message limit
    return (long) (TOTAL_MEM_MESSAGES / divisor);
}
```

Listing 2.3 shows the Java method calculateInitialMessageLimit which is utilized in Lighthouse with pipelining to calculate a worker's optimal initial message limit. The method calls calculatePathDivisor for the main path of the handled query plan. This path has the ID 0 and is different from other paths as it is guaranteed to end with the plan's last operator. To avoid consideration of memory consumption for produced messages on the last operator, the passed argument for the *isMain* parameter is `true`. When no non-last globals operators are present in the query plan, a large initial message limit is returned to minimize the execution's number of required supersteps.

Listing 2.4: Calculation of Path Divisor

```
private double calculatePathDivisor(QueryItemPath currentPath, boolean isMain) {
    double divisor = 0;

    <Set end to avoid visiting a last global operator when traversing the main path>
    for (int step = 0; step < end; step++) {
        if (currentPath.get(step) instanceof Move
             || currentPath.get(step) instanceof StepJoin) {
            // Increase divisor on reached global operator
            divisor +=
                currentPath.get(step).getAverageOutMessageSizeSerialized();
                * 1.5 * calculatePathFactor(currentPath, step);
        } else if (currentPath.get(step) instanceof PathJoin) {
            // Increase divisor on global operators in right query path
            int thisPathJoinId = ((PathJoin) currentPath.get(step)).getId();
            QueryItemPath right = getPathJoinRightPath(thisPathJoinId);
            divisor += calculatePathDivisor(right, false);
    }}

    return divisor;
}
```

Listing 2.4 shows the Java method calculatePathDivisor which is used in Lighthouse with pipelining to calculate the *divisor* on a worker. It is returned from a call with the main path as argument, after all paths of the handled query plan have been traversed and their divisors have been summed. The divisor of a path is increased for every encountered non-last global operator with a summand similar to *mem_messages_nonlast_global* of Listing 2.2. However, this summand does not include an *initial_message_limit* factor. The called method calculatePathFactor multiplies out-in ratios and the fraction of initial messages which may cause messages to be passed by the current operator. This method is later presented in Listing 2.6 and corresponds to the previously used *predicted_superstep_num_out* function. When a PathJoin is encountered in the calculatePathDivisor method, a recursive call with its right path as argument takes place, returning another divisor to be added. All local operators which are come across during the traversal of operators are ignored.

Calculation Details for Executions of Bushy Query Plans

It is more complex to accurately predict memory consumption in executions of bushy query plans. Even if executions of bushy query plans have initial message limits which never change, each worker is likely to increase its memory consumption for every computed superstep. This is a consequence of PathJoin operators commonly passing small numbers of messages in early supersteps, then large numbers of messages in late ones. Normally, all non-last global operators and join tables following a PathJoin are gradually requiring more memory, potentially causing workers to fail. A calculation of an optimal initial message limit must take into account the growing numbers of messages to be passed by PathJoin operators in the handled query plan. With continuously filled join tables, the observed out-in ratio for any PathJoin during the calculation is quickly outdated.

A proposed solution, to handle the changing out-in ratios for PathJoin operators, is to replace them with forged out-in ratios in all calculations of initial message limits. Each forged out-in ratio represents the predicted out-in ratio which can be observed for the related PathJoin in the end of the execution. Listing 2.5 presents how a *forged_out_in_ratio* is calculated for a PathJoin on a worker. The predicted total number of messages to be received from a path to the PathJoin on the worker, represented by *predicted_total_num_in_left_path* and *predicted_total_num_in_right_path*, is calculated with a multiplication of the path factor for the path's previous operators and the total number of initial messages to be created by the worker. The predicted number of vertices on the worker to store a join table for the PathJoin after the final superstep of the execution, represented by *predicted_total_num_active_vertices*, is calculated with the worker's current numbers of vertices storing a related join table and already created initial messages, and the total number of initial messages to be created by the worker. This prediction assumes that the number of active vertices for the PathJoin grows linearly with the number of created initial messages. The predicted total number of messages to be passed by the PathJoin on the worker is represented by *predicted_total_num_out*. The prediction assumes that on all vertices where the PathJoin is computed, equally many related messages are received. If the assumptions are violated, more messages than estimated might be passed by the PathJoin, causing workers to run out of memory.

Listing 2.5: Calculation of Out-In Ratio for PathJoin

```
predicted_total_num_in_left_path =
    calculate_path_factor(left_path, current_left_path_step - 1)
    * worker_total_num_initial_messages
predicted_total_num_in_right_path =
    calculate_path_factor(right_path, current_right_path_step - 1)
    * worker_total_num_initial_messages
predicted_total_sum_num_in = predicted_total_num_in_left_path
                           + predicted_total_num_in_right_path

predicted_total_num_active_vertices = current_num_active_vertices
                                    / worker_current_num_initial_messages
                                    * worker_total_num_initial_messages
predicted_total_num_out =
    predicted_total_num_in_left_path * predicted_total_num_in_right_path
    / predicted_total_num_active_vertices

forged_out_in_ratio = predicted_total_num_out / predicted_total_sum_num_in
```

Listing 2.6 shows the Java method calculatePathFactor which is used in Lighthouse with pipelining to calculate path factors on a worker. With the provided *lastStep* argument, a calculation can be limited to a smaller part of the *currentPath* passed. A returned *pathFactor* is calculated considering the fractions of initial messages to start paths leading towards the last operator, specified by the method's arguments, and the out-in ratios for the operators on these paths. The *pathFactor*'s initial value is the fraction of initial messages to start the path containing the last operator. While the path's operators are later traversed, the *pathFactor*'s value is multiplied with the out-in ratio for each. If a PathJoin is encountered, a path factor is recursively calculated for its right path. The path factors for the PathJoin's incoming paths are summed to *pathFactor*, since the expected number of messages a PathJoin passes is a product of its out-in ratio and sum of received messages.

Listing 2.6: Calculation of Path Factor

```
private double calculatePathFactor(QueryItemPath currentPath, int lastStep) {
    double pathFactor = (double) 1 / numPaths;

    for (int step = 0; step <= lastStep; step++) {
        if (currentPath.get(step) instanceof PathJoin) {
            QueryItemPath right =
                getPathJoinRightPath(((PathJoin) currentPath.get(step)).getId());
            pathFactor += calculatePathFactor(right, right.size() - 2);
        }

        pathFactor *= currentPath.get(step).getOutInRatio();
    }

    return pathFactor;
}
```

Path factor calculations are required in listings 2.2, 2.4 and 2.5. By multiplying the total number of initial messages to be created on a worker in an execution with a path factor, an expected total number of messages to be passed by the worker can be calculated for any operator in the plan. If the number of initial messages to be created on a worker in a superstep is used instead, an expected number of messages to be passed by the worker in a superstep for any operator can be calculated.

2.1.2 When to Calculate New Initial Message Limits

As a precaution, a low default value must be used by each worker for its initial message limit until all statistics needed from the ongoing execution to calculate an optimal limit is available. Unfortunately, a single default value does not suit all potential combinations of generated query plans, input graphs and utilized workers. An initial message limit appearing small in one execution, may in another immediately cause workers to run out of memory. When only having a few workers with a small memory capacity, a given input graph will occupy a larger fraction of the available memory, increasing the possibility for workers to fail. During a query plan execution, each worker should calculate and use an optimal initial message limit instead of the default one. This normally reduces the number of required supersteps, but still enables the execution to successfully finish.

Two different patterns for when to calculate new initial message limits have been implemented for Lighthouse with pipelining. With static initial message limits, a calculation of an optimal initial message limit is performed once by each worker in an execution. With dynamic initial message limits, eventually a calculation is performed by each worker before every started superstep.

Pipelining with Static Initial Message Limits

As implied by describing the used initial message limits with this calculation pattern as static, the limit on a worker remains unchanged in an execution after it once has been calculated. Prior to any calculations taking place, computation statistics must be gathered for every non-last global operator of the handled query plan (this requires all of them to potentially have been computed). In executions where some query plan operators are not computed, the calculations of limits should not be postponed forever. A problem with static initial message limits is that they are calculated with statistics gathered after a small number of query path computations. These have been started

on just a few vertices of the input graph, which might not constitute a representative sample for the whole input graph, possibly causing calculations of too large limits and workers to run out of memory. The calculation pattern for pipelining with static initial message limits is implemented in the preSuperstep method of Lighthouse's WorkerContext class, as showed in Listing 2.7.

Every worker measures its *available unused memory* before starting the first superstep in an execution. This previously discussed number is later used when a worker calculates its optimal initial message limit. Shortly after performing the measurement, a worker temporarily sets its *initialMessageLimit* to the small constant *DEFAULT_INITIAL_MESSAGE_LIMIT*. This assignment is only avoided if the handled query plan does not contain any non-last global operators. In such situation, the maximum memory consumption for Lighthouse can not be reduced with pipelining. The value of *minimumExecutionLength* depends on the processed query plan, as it represents the number of supersteps required to compute all the present operators for creation of a complete solution. When the preSuperstep method is called on a worker right before superstep *minimumExecutionLength* - 1, all the non-last global operators of the query plan have potentially been computed. The worker's *initialMessageLimit* is then calculated and set, changing the number of path computations it can start per superstep. A RuntimeException is thrown if a negative or zero initial message limit is calculated. This should only happen if the worker has no available memory for storing messages on global operators after having reserved memory for skew protection and increasing PathJoin tables.

Listing 2.7: Calculation of Static Initial Message Limit

```
public void preSuperstep() {
    if (getSuperstep() == 0) {
        AVAILABLE_UNUSED_MEMORY = freePlusUnallocatedMemory();
    }
    ...
    // Set new initial message limit
    if (getSuperstep() == 0 && minimumExecutionLength != 1) {
        initialMessageLimit = DEFAULT_INITIAL_MESSAGE_LIMIT;
    } else if (getSuperstep() < minimumExecutionLength - 1) {
        initialMessageLimit = 0;
    } else if (getSuperstep() == minimumExecutionLength - 1) {
        initialMessageLimit = calculateInitialMessageLimit();
        if (initialMessageLimit <= 0) {
            throw new RuntimeException("Bad initial message limit!");
}}}
```

Before all supersteps between the first and the calculation of an optimal initial message limit, each worker's *initialMessageLimit* is set to 0. In early versions of pipelined Lighthouse, this prevented bad predictions for numbers of messages to be received by PathJoin operators with incoming paths requiring different numbers of supersteps to be computed. Regrettably, this code was not removed before performing the later presented measurements. With recent versions of pipelined Lighthouse, the numbers of messages to be received by a PathJoin is predicted with use of calculatePathFactor.

Pipelining with Dynamic Initial Message Limits

As implied by describing the used initial message limits with this calculation pattern as dynamic, the limit on a worker can be changed multiple times after it once has been calculated. Similar to with use of static initial message limits, computation statistics must be gathered for every non-last global operator of the handled query plan before any calculations can take place. The main advantage of using dynamic initial message limits over static ones is that they gradually improve. They are calculated with statistics gathered after a growing number of query path computations. While the number of start vertices which have been used for computations on a worker increases, the start vertices should afterwards constitute a more representative sample for the whole graph. With improving initial message limits, less skew protection memory is consumed. The Java code enforcing the calculation pattern for pipelining with dynamic initial message limits is also placed in the preSuperstep method of Lighthouse's WorkerContext class. It is showed in Listing 2.8.

There is one significant difference between the code triggering calculations of static and dynamic initial message limits. In pipelined Lighthouse with dynamic initial message limits, a worker's *initialMessageLimit* is recalculated for every superstep larger than *minimumExecutionLength* - 1.

Listing 2.8: Calculation of Dynamic Initial Message Limit

```
public void preSuperstep() {
    ...
    // Set new initial message limit
    if (getSuperstep() == 0 && minimumExecutionLength != 1) {
        initialMessageLimit = DEFAULT_INITIAL_MESSAGE_LIMIT;
    } else if (getSuperstep() >= minimumExecutionLength - 1) {
        initialMessageLimit = calculateInitialMessageLimit();
        if (initialMessageLimit <= 0) {
            throw new RuntimeException("Bad initial message limit!");
}}}
```

2.1.3 Automatic Restart or Continuation of Failed Executions

Even though skew protection memory is made available, workers in pipelined Lighthouse executions may still run out of memory. Since memory consumption predictions are based on observed out-in ratios for operators and average message sizes, utilized workers are vulnerable to sudden creation or receipt of large amounts of messages. Many variations in numbers of present messages during a query plan execution can be caused by skewed input data. Skew may lead to the outcome of path computations in various supersteps on a worker to differ, in addition to make some workers produce or receive more messages than others. An inherent problem for executions of query plans having operators which pass very few messages is that an operator on a worker might not pass any messages before the worker calculates an optimal initial message limit. This makes the worker expect that no messages will ever be received from the operator, potentially causing it to calculate and use a large initial message limit. If messages are later unexpectedly passed by the operator, following operators can create large numbers of messages which do not fit into the available memory.

To execute query plans in a pipelined manner with calculated initial message limits is a proactive approach to prevent executions from trying to store too many messages in memory. All calculations of initial message limits are based on predicted memory consumption and require knowledge about the generated query plan and the input graph. A pipelined execution which fails with workers running out of memory, then is restarted with the same workers, is likely to fail again since the input vertices are processed in the identical order. To eventually succeed an execution of this job, pipelined Lighthouse must also be reactive and utilize information about its previous execution attempts. The used initial message limits in a retry attempt should be more defensive. If a pipelined Lighthouse execution is solely taking a reactive approach, it does not require any knowledge about the generated query plan and the input graph, as no predictions are ever performed. An execution attempt can start with small initial message limits which increase for every performed superstep. If an attempt suddenly fails, it must be restarted and set to use smaller initial message limits.

Two different mechanisms for automatically reattempting failed query plan executions have been implemented for Lighthouse with pipelining. The simplest one, described to restart failed executions, involves to delete all complete solutions written to the HDFS by the last failed execution attempt, before starting another. The more complicated mechanism, described to continue failed executions, involves to only delete complete solutions written to the HDFS by the last failed execution attempt in its final superstep, before starting another attempt.

Automatic Restart of Failed Executions

This mechanism, for automatically reattempting failed query plan executions, works in combination with both the previously described calculation patterns for initial message limits. For each new execution attempt started after a failed attempt, all initial message limits are set more defensively. This ensures that a query plan execution eventually succeeds, provided that the given input graph fits into the available memory of the used workers and space can be reserved for skew protection and PathJoin tables. The restart mechanism deletes all complete solutions written to the HDFS by failed attempts, preventing duplicates of solutions in the output folder after an execution succeeds. A successful attempt will itself find all complete query solutions and write them to the HDFS.

The created JobRetryChecker class, presented in Listing 2.9, implements the GiraphJobRetryChecker interface. Objects of this class are used to force the Giraph framework to automatically reattempt failed executions. Giraph immediately calls the method shouldRetry after an execution attempt has failed, then starts a new attempt if the method returns `true`. Currently, a maximum number of 5 attempts can be started for a specified job. If the execution does not succeed during these, other reasons than large initial messages limits are likely to make the attempts fail. Before restarting an execution, a `retry.info` file is created in the available HDFS with information about the number of already failed attempts. Before returning `true`, all previously written solutions are deleted.

Listing 2.9: shouldRetry() of Restart Implementation

```
public class JobRetryChecker implements GiraphJobRetryChecker {
    public boolean shouldRetry(Job submittedJob, int tryCount) {
        if (tryCount < 5) {
            <Create retry.info file>
            <Flush and close output stream>
            <Delete written complete solutions from HDFS>
            return true;
        }

        return false;
    }
    ...
}
```

With the restart mechanism enabled, the number of previously failed execution attempts is read from the preApplication method implemented in Lighthouse's WorkerContext class. This method is showed in Listing 2.10. Immediately after an execution attempt is started, Giraph forces the preApplication method to be called on each worker. A worker then tries to read the number of already failed attempts from the `retry.info` file in the HDFS. If successful, it sets its *RETRY_FACTOR* to 0.1 raised to the power of the number of failed attempts. As seen in Listing 2.11, a default or calculated initial message limit value is multiplied with the retry factor before being assigned to *initialMessageLimit*. The preSuperstep method is here implemented to set static initial message limits. By exponentially decreasing the used initial message limits for each execution attempt, smaller numbers of path computations are started per superstep. The default value of *RETRY_FACTOR* is 1. This value is never changed in a first execution attempt, causing used initial message limits to either have default values or be exactly as calculated by the calculateInitialMessageLimit method.

Listing 2.10: preApplication() of Restart Implementation

```
public void preApplication() {
    ...
    // Try to read retry information from file and set new retry factor
    Path retryInfoPath = new Path(RETRY_INFO_LOCATION);
    if (fs.exists(retryInfoPath)) {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(fs.open(retryInfoPath)));
        RETRY_FACTOR = Math.pow(0.1, Integer.parseInt(br.readLine()));
        br.close();
}}
```

Listing 2.11: preSuperstep() of Restart Implementation

```
public void preSuperstep() {
    ...
    // Set new initial message limit
    if (getSuperstep() == 0 && minimumExecutionLength != 1) {
        initialMessageLimit = (long)
            (DEFAULT_INITIAL_MESSAGE_LIMIT * RETRY_FACTOR);
    } else if (getSuperstep() == minimumExecutionLength - 1) {
        initialMessageLimit = (long)
            (calculateInitialMessageLimit() * RETRY_FACTOR);
        if (initialMessageLimit <= 0) {
            throw new RuntimeException("Bad initial message limit!");
}}}
```

Automatic Continuation of Failed Executions

This mechanism, for automatically reattempting failed query plan executions, also works in combination with both the previously described calculation patterns for initial message limits. However, as later explained, it does not work for failing executions of bushy query plans. For each new execution attempt started after a failed attempt, most initial message limits are set more defensively. In a retry attempt, a worker's first limit is set to the product of a limit used by the worker with the same ID in the previous attempt and a retry factor. As workers here do not need to consider the default initial message limit value, some retry attempts avoid to start with very low limits. Differently from the simple restart mechanism, the continuation mechanism only deletes complete solutions written to the HDFS by the last failed execution attempt in its final superstep, before starting a new attempt. One or more workers failed in this interrupted superstep.

A worker should after every superstep append the following to the HDFS file `fin/<worker-id>`:

1. the number of the superstep
2. its number of related initial messages which may have resulted in complete solutions
3. its initial message limit value

The `fin` HDFS directory contains worker specific computation information for all finished supersteps. If a worker can get the number of produced initial messages by workers with the same ID in previous attempts and which may have resulted in complete solutions, it can prevent creation of initial messages for complete solutions that are already stored in the HDFS. If a worker can get the initial message limit which was used by a worker with the same ID in the last successful superstep, it can set its first limit without considering the default initial message limit value.

Before a superstep is started, the worker with ID 0 writes the number of the superstep to the file `started.superstep` in the HDFS. This enables workers to read in which superstep a failed execution attempt has stopped. The number of the last successful superstep in a failed attempt is used by each worker in the next attempt to find relevant information of that enumerated above. There are two important properties of Giraph which allow workers in retry attempts to use counts for produced initial messages to skip initial message creations, but still produce correct output. Workers in different attempts with the identical assigned worker ID are responsible for the same partition of the input graph. A partition has in all attempts its vertices stored in the same order.

Listing 2.12: preApplication() of Continue Implementation

```
public void preApplication() {
    ...
    // Try to read last failed superstep number from file
    Path superstepPath = new Path(STARTED_SUPERSTEP_LOCATION);
    if (fs.exists(superstepPath)) {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(fs.open(superstepPath)));
        LAST_FAILED_SUPERSTEP = Integer.parseInt(br.readLine());
        br.close();
}}
```

With the continuation mechanism enabled, prior to any superstep being started in a retry attempt, a read of the number of the superstep in which the previous execution attempt failed takes place. The preApplication method performs the read as presented in Listing 2.12. The number of an interrupted superstep is only available from `started.superstep` in retry attempts. In a first query plan execution attempt, *LAST_FAILED_SUPERSTEP* does not change value from the default -1. In all retry attempts, it is set to a read superstep number. A non-default value for this variable is later considered by a worker on multiple occasions. First, when deleting output written by the last failed attempt in its final superstep. Then, when reading the number of related initial messages which may have resulted in complete solutions and a limit used in the last successful superstep.

All deletion of complete solutions takes place in the preSuperstep method showed in Listing 2.13. A worker checks prior to the first superstep of an attempt whether *LAST_FAILED_SUPERSTEP* is set to a non-default value. In such case, the previous execution attempt has failed. The worker is then required to participate in deleting complete solutions from the HDFS to prevent duplicates.

Each worker deletes the output written by the worker with the same ID in the final superstep of the previous attempt. Output is located in HDFS files named `solution_worker_<worker-id>_attempt_<attempt-number>_superstep_<superstep-number>`. After deleting output, a worker finds the above enumerated information for the last successful superstep in its `fin/<worker-id>` file. Before each superstep, the worker with ID 0 writes the number of the superstep to the HDFS. Every worker also opens a stream to an own superstep specific output file. In all retry attempts, *PREVIOUS_ATTEMPT_INITIAL_MESSAGE_LIMIT* is used in place of the default limit. The preSuperstep method is here implemented to set static limits. By replacing its last equality operator with a "greater than or equal to" operator, it can be changed to set dynamic limits instead.

Listing 2.13: preSuperstep() of Continue Implementation

```
public void preSuperstep() {
    if (getSuperstep() == 0 && LAST_FAILED_SUPERSTEP != -1) {
        <Delete complete solutions from HDFS written in the last failed superstep>

        // Read info about the last successful superstep from fin/<worker-id> in HDFS
        <Read number of initial messages which may have resulted in complete solutions>
        <Read used initial message limit in the superstep>
    }

    if (getMyWorkerIndex() == 0) {
        <Write number of this superstep to HDFS>
    }

    <Open own superstep output stream>
    ...
    // Set new initial message limit
    if (getSuperstep() == 0 && minimumExecutionLength != 1) {
        if (PREVIOUS_ATTEMPT_INITIAL_MESSAGE_LIMIT == -1) {
            initialMessageLimit = (long)
                (DEFAULT_INITIAL_MESSAGE_LIMIT * RETRY_FACTOR);
        } else {
            initialMessageLimit = (long)
                (PREVIOUS_ATTEMPT_INITIAL_MESSAGE_LIMIT * RETRY_FACTOR);
    }} else if (getSuperstep() == minimumExecutionLength - 1) {
        initialMessageLimit = (long)
            (calculateInitialMessageLimit() * RETRY_FACTOR);
        if (initialMessageLimit <= 0) {
            throw new RuntimeException("Bad initial message limit!");
}}}
```

The postSuperstep method of pipelined Lighthouse with the continuation mechanism enabled is presented in Listing 2.14. It forces each utilized worker to flush and close its output stream after every computed superstep. Further, it makes a worker increase its *finishedInitialMessages*, which is the number of related initial messages that may already have resulted in complete solutions. In the end, this count and the current initial message limit is appended to the worker's `fin/<worker-id>`.

Listing 2.14: postSuperstep() of Continue Implementation

```
public void postSuperstep() {
    <Flush and close own superstep output stream>

    // Add count of created initial messages in the superstep as last element in queue
    initialMessageCounts.addLast(new Long(initialMessageCount));

    <Reset initial message count>

    // Increase number of initial messages which may have resulted in complete solutions
    if (getSuperstep() >= minimumExecutionLength - 1) {
        finishedInitialMessages += initialMessageCounts.removeFirst();
    }

    // Append to fin/<worker-id> in HDFS
    <Append number of the finished superstep>
    <Append number of initial messages which may have resulted in complete solutions>
    <Append used initial message limit in the superstep>
}
```

As showed in Listing 2.15, a worker does not produce initial messages which possibly have resulted in complete solutions currently being stored in the HDFS. *messageCreationsToSkip* specifies how many initial message creations should be skipped on the worker. To enable all vertices to eventually vote to halt, a relevant vertex' started paths count must be incremented for every skipped creation.

Listing 2.15: Lighthouse compute() of Continue Implementation

```
public void compute(Vertex vertex, Iterable<Message> messages) {
    <Get worker context used to check initial message count, limit and creations to skip>
    <Get number of unstarted paths for vertex>
    if (workerContext.initialMessageCount < workerContext.initialMessageLimit
        && numUnstartedPaths > 0) {
        if (workerContext.messageCreationsSkipped < workerContext.messageCreationsToSkip
            && getSuperstep() == 0) {
            <Increase number of started paths for vertex without creating initial messages>
            <Increase number of message creations skipped>
        } else {
            <Create initial messages for paths of the query plan>
            <Pass computation for each created initial message to first query item of path>
    }}

    <Pass computation for each received message to next query item of path>

    <Vote to halt when all query paths have been started on the vertex>
}
```

In the Lighthouse versions seen until now, the join tables of a PathJoin must store all the operator's previously received messages. To produce every complete solution with the continuation mechanism enabled, the join tables of a PathJoin must also store messages passed to the operator in already failed execution attempts. This is not supported with the current implementation. A fix requires messages stored on PathJoin operators during an execution to be written to the HDFS. These must then be loaded from the HDFS into join tables in the beginning of following retry attempts.

With the continuation mechanism enabled, each computed superstep can be considered a checkpoint. If an attempt fails, the following retry starts path computations as the latest superstep in which no created initial messages can have resulted in stored complete solutions. When continuation from failures is handled by Lighthouse itself, Giraph's checkpointing is no longer needed.

2.2 PathJoin Memory Optimizations

A PathJoin in the original version of Lighthouse has similar behavior to that of a so-called Pipelining HashJoin[16]. It allows solution messages from both its incoming paths to arrive in any superstep without affecting the number or correctness of solutions passed. This permits use of PathJoin in pipelined Lighthouse, where the messages from a path can arrive over multiple supersteps. Unfortunately, this behavior also requires join tables to consume large amounts of memory. In total, join tables must store all messages received by any PathJoin until the end of the ongoing execution.

The following presented optimizations introduce changes to the implemented PathJoin, reducing the related memory consumption. The first two optimizations prevent a PathJoin from having the behavior of a Pipelining HashJoin. If either pipelining or one of these optimizations is to be used, some pre-execution analysis should be performed to determine which is the most appropriate. Such analysis is not explored in this project. The last optimization reduces the amount of memory that is needed to store a number of messages in join tables. It is particularly helpful with present Pipelining HashJoin behavior, as this behavior requires more messages to be stored during executions.

2.2.1 Forced Simultaneous Arrivals of Messages from Both Paths

This optimization can not be used in combination with pipelining. It competes with pipelining as a solution to reduce the maximum amount of required memory during an execution. At its core, the optimization permits executions of query paths to start in different supersteps and forces the paths leading to a PathJoin to deliver all messages in the same superstep. This allows a join

table on a vertex to only exist during the superstep's compute call for the vertex. Usually, a call like this will return within a small fraction of the total superstep time. By reducing the periods join tables are stored in vertex values, later computations are going to benefit with more available memory for storing messages. The optimization also avoids the extra initial supersteps which are introduced by the use of pipelining. No computation statistics for paths are ever needed to ensure simultaneous arrivals. Often, the first operators of query paths lead to large amounts of messages being transferred. By starting query paths in different supersteps, the numbers of messages being sent in the various early supersteps can be balanced out. A disadvantage of the simultaneous arrivals optimization is that some vertices may end up being active for more supersteps. If they are forced to start in later supersteps, they do not vote to halt immediately after superstep 0.

Figure 2.3 visualizes an optimized execution with multiple PathJoin operators. Instead of starting all query path computations in superstep 0, requiring join tables for both the PathJoin operators to be stored over multiple supersteps, an alternative start order is used. The top path is started in superstep 1, the middle path in superstep 0 and the bottom path in superstep 1. Each join table only needs to exist for a fraction of a superstep, but all join operations are enabled to take place.

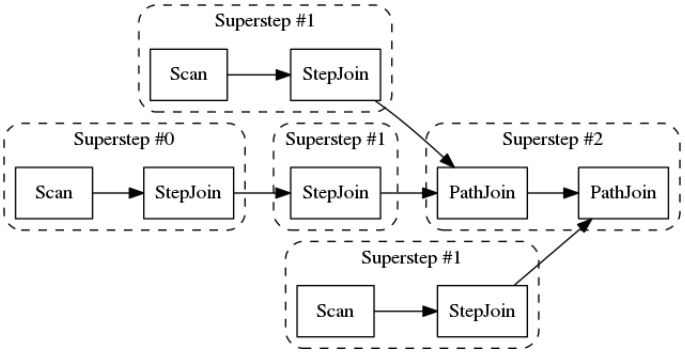


Figure 2.3: Query plan execution in which paths are started in different supersteps

The Lighthouse compute method that enables simultaneous arrivals, showed in Listing 2.16, allows the paths of a given query plan to be started in different supersteps. The method starts supersteps based on *startSuperstepToPathsMap* in the worker context. Each entry in this map has a superstep number as its key and a list of paths to be started in the superstep as its value. The start superstep numbers and paths are added to the map after the creation of the query plan tree, knowing the number of supersteps which are needed to produce a single complete solution. A set with references to visited PathJoins operators is temporary stored. It is filled when being passed as an argument in recursive compute calls for local operators. The compute method finally deletes all join tables for a vertex, using the visited PathJoin set, before deciding whether the vertex can vote to halt.

Listing 2.16: Lighthouse compute() of Simultaneous Arrivals Implementation

```
public void compute(Vertex vertex, Iterable<Message> messages) {
    <Get worker context used to check query plan and write complete solutions>

    // Create an initial message for each path to be started in this superstep
    LinkedList<QueryItemPath> paths =
        workerContext.getPathsWithStartSuperstep(getSuperstep());
    if (paths != null) {
        for (QueryItemPath path : paths) {
            <Create and add an initial message to initial message list>
    }}

    // Create a set for references to all PathJoin operators reached in this superstep
    <Pass computation for each created initial message to first query item of path>
    <Pass computation for each received message to next query item of path>

    <Remove references to old memory expensive join tables if present>

    <Vote to halt when all query paths have been started on the vertex>
}
```

The concept of deleting join tables which are not needed anymore, at the end of compute calls, can also be utilized with pipelining. However, this requires another mechanism to be implemented, involving that workers in the cluster can signalize to each other when they are finished sending messages on a path. This can be supported through use of aggregators (described in appendix A) or end-of-stream messages. When all messages from both sides to a PathJoin have been received in an execution, the related join tables can be deleted before the next superstep. This will free up memory on the workers for remaining computations without any influence on creation of solutions.

2.2.2 Forced Prior Arrivals of Messages from Path Passing Fewest Bytes

The smallest incoming path of a PathJoin is here considered to be the one which delivers the smallest number of bytes. All messages from the smallest path can be forced to arrive at a PathJoin before any messages from the largest path. Normally, when computing a PathJoin on a vertex, all messages received from both the incoming paths are stored in the related join table on the vertex. With this optimization, just messages passed from the smallest query path are stored. The messages received from the largest path are simply joined with every stored message in the join table. All joined messages are passed by the PathJoin. Statistics from the early supersteps in an execution should be used to determine which of the paths leading to a PathJoin is the smallest. An execution is started with use of pipelining and a small default initial message limit value. After having gathered the needed statistics, the used workers must agree on which path is the smallest for each PathJoin. Every present join table is then deleted and the execution is restarted to run with known smallest paths. In worst case, two incoming paths' messages consume the same amount of memory. The required memory for storing join tables can then only be reduced with 50 percent.

Figure 2.4 visualizes another optimized execution of the same query plan as in Figure 2.3. Each showed superstep number is here relative to the first superstep after the ones in which the statistics about the present paths were gathered. In this execution, all messages passed to the same PathJoin have the similar size. As without the optimization, all path computations are started in superstep 0. This causes the message from the top path to reach the first PathJoin in superstep 1 and the messages from the middle path to arrive in superstep 2. Also, the messages from the bottom path reach the second PathJoin in superstep 1, while the messages from the first PathJoin arrive in superstep 2. Each join table only needs to contain messages from its PathJoin's smallest path.

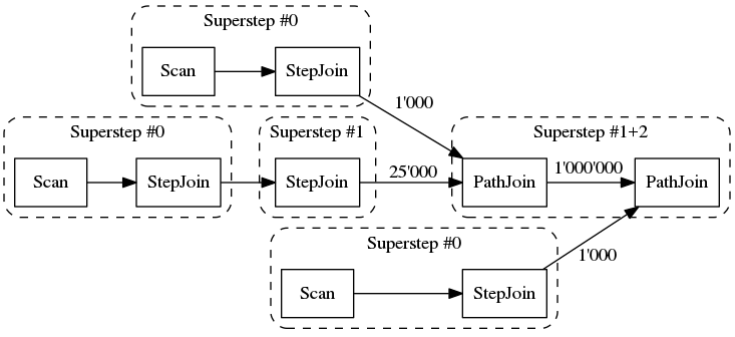


Figure 2.4: Query plan execution in which paths are started in the same superstep

The Lighthouse compute method that enables smallest-first arrivals is similar to the one which enables simultaneous arrivals. However, it does not remove references to old join tables. Further, it requires that a worker context's *startSuperstepToPathsMap*, with the information about the start supersteps for paths, is created based on gathered path statistics and made agreements. A separate OrderedPathJoin operator can be added to Lighthouse. It must ensure that all messages from one incoming path are stored in related join tables before it receives any messages from the other path.

The smallest-first optimization can be adjusted to work in combination with pipelining, despite preventing Pipelining HashJoin behavior. With pipelined Lighthouse, it is difficult for workers to recognize when every single solution from a smallest incoming path has arrived at a PathJoin. To avoid that any worker delivers messages from the largest path before all workers have finished sending messages from the smallest, either aggregators or end-of-stream messages must be used.

2.2.3 Storage of Serialized Messages in PathJoin Tables

By storing messages in join tables in a serialized form, the sizes of join tables can be kept lower. For comparison, a serialized empty message has in experiments showed to need 6 bytes of memory, while an empty message as a Java object requires 88 bytes. Considering these numbers, less than 8 percent of the bytes needed to store an empty message as a Java object are absolutely required. To store serialized messages in join tables also improves memory consumption predictions, since it enables collection of accurate information about how much memory present join tables consume. When storing join tables' messages as Java objects, the related memory consumption is guessed with assumptions about the needed space by involved data structures and sizes for primitive types.

Pipelined Lighthouse uses the predictPathPJMemoryConsumption method, showed in Listing 2.17, to predict the total amount of memory which is needed by a worker to store messages in join tables at the end of a query plan execution. This memory consumption is together with the size of reserved skew protection memory used to determine how much space should be utilized to store messages on global operators. Before a calculation of an initial message limit, the recursive prediction method is called with the main path as argument. This ensures that every PathJoin in the query plan is considered. The main path has the ID 0 and is guaranteed to end with the plan's last operator. The getAverageOutMessageSizeSerialized method is called once for each operator last preceding a PathJoin. Before being added to a path's *predictedPJMemoryConsumption*, a returned size is multiplied with the total number of messages which are expected to be passed by the operator and a factor of 1.5 to take into account the overhead that is introduced by join tables' data structures.

Listing 2.17: Prediction of Total Memory Consumption for PathJoin Operators

```
private long predictPathPJMemoryConsumption(QueryItemPath path) {
    long predictedPJMemoryConsumption = 0;
    for (int step = 1; step < path.size(); step++) {
        if (path.get(step) instanceof PathJoin
            && ((PathJoin) path.get(step)).getSide() == Side.LEFT) {
            QueryItemPath right =
                getPJRightPath(((PathJoin) path.get(step)).getId());
            predictedPJMemoryConsumption += predictPathPJMemoryConsumption(right);

            predictedPJMemoryConsumption +=
                predictPathFactor(path, step - 1) * numInitialMessages
                * path.get(step - 1).getAverageOutMessageSizeSerialized() * 1.5;
            predictedPJMemoryConsumption +=
                predictPathFactor(right, right.size() - 2) * numInitialMessages
                * right.get(right.size() - 2).getAverageOutMessageSizeSerialized() * 1.5;
    }}

    return predictedPJMemoryConsumption;
}
```

Listing 2.18: Update of Operator Statistics with Message Passed

```
protected void messagePassed(Message message) {
    outMessageCount++;
    if (message.getNumberOfBytesSerialized() != averageOutMessageSizeSerialized) {
        averageOutMessageSizeSerialized =
            (averageOutMessageSizeSerialized * (outMessageCount - 1)
             + message.getNumberOfBytesSerialized()) / outMessageCount;
}}
```

The messagePassed method, presented in Listing 2.18, is called each time a message is passed by an operator. It updates statistics for the operator, such as *outMessageCount* and *averageOutMessageSizeSerialized*, and ensures that getAverageOutMessageSizeSerialized returns correct values. In the future, for efficiency reasons, *averageOutMessageSizeSerialized* should exclusively be updated on global operators and operators last preceding a PathJoin. Only the sizes of serialized messages from these operators are needed to calculate proper initial message limits. For a message passed, a DataOutputStream is created to find the size of the message when it is stored in a serialized form. This check is time consuming and causes the JVM's garbage collector to be triggered more often. A future improvement should be to only use a single DataOutputStream per worker to find sizes.

2.3 Improved Data Structures and Serialization Formats

The introduced logic for pipelined Lighthouse requires that additional data is stored in memory. For each graph vertex, an extra 4 byte integer *numUnstartedPaths* must be handled. By changing how data for vertices, edges and messages is stored, the maximum memory consumption during Lighthouse executions can be reduced. This allows larger graphs to be loaded into the memory of utilized workers and more messages to be produced. First of all, null references should be used instead of references to empty data structures. Further, the general and serializable Writable data structures that are provided by Hadoop and Giraph are memory inefficient and should be avoided.

Listing 2.19: VertexValue with null References

```
public class VertexValue implements Writable {
    private LongArrayWritable labels;
    private PropertiesMapWritable properties = null;
    private IntToPathJoinBindingsMapWritable pathJoinMap = null;

    public VertexValue(LongArrayWritable labels, PropertiesMapWritable properties) {
        this.labels = new LongArrayWritable();
    }

    public PathJoinBindings addToJoinTable(PathJoin operator, Message message) {
        if (pathJoinMap == null) {
            pathJoinMap = new IntToPathJoinBindingsMapWritable();
        }
        ...
    }

    public void readFields(DataInput input) throws IOException {
        labels.readFields(input);
        if (input.readBoolean()) {
            properties = new PropertiesMapWritable();
            properties.readFields(input);
        } else {
            properties = null;
        }
        if (input.readBoolean()) {
            pathJoinMap = new IntToPathJoinBindingsMapWritable();
            pathJoinMap.readFields(input);
        } else {
            pathJoinMap = null;
    }}

    public void write(DataOutput output) throws IOException {
        labels.write(output);
        if (properties != null) {
            output.writeBoolean(true);
            properties.write(output);
        } else {
            output.writeBoolean(false);
        }
        if (pathJoinMap != null) {
            output.writeBoolean(true);
            pathJoinMap.write(output);
        } else {
            output.writeBoolean(false);
    }}

    ...
}
```

2.3.1 Replacing Empty Data Structures with null References

A reimplementation of Lighthouse's original VertexValue class is presented in Listing 2.19. It serves as one example showing how the representation for an entity can be changed to require less space. Specifically, the class defines how the data for a vertex should be stored in memory. Instead of keeping references to empty Writable hash maps, `null` is used. The write method, which carries

out serialization of data for vertices, is changed to prevent attempts on serialization of `null` values. The serialization format is adjusted to involve boolean values that specify whether hash maps are present or not. In addition to using null references for both *properties* and *pathJoinMap* of vertices, null references are also used for empty *properties* and *binding* of edges and messages, respectively.

The presented VertexValue class involves some changes which are irrelevant for this section's topic. Instead of using ArrayListWritable and MapWritable from Giraph and Hadoop, self-implemented Writable data structures are utilized. The rationale for this change is presented in the next sections. VertexValue has been merged with its wrapper class VertexValuePJ. Previously, the VertexValuePJ part of a vertex' data contained the reference to its *pathJoinMap*. This merge decreases the numbers of stored references and objects in Lighthouse executions, reducing the memory storage overhead.

2.3.2 Replacing General Writable Data Structures with Specialized Ones

Instead of using the general and memory inefficient MapWritable from Hadoop, new specialized serializable hash maps are utilized. During serialization, a MapWritable instance must for each key and value write the relevant class ID in addition to the bytes for the related object. This is required since the various keys and values can be of different Writable types. As the new serializable hash maps are not created for general use, they do not need to write class IDs for the objects they store. The new hash maps inherit from Java's HashMap class and implement the Writable interface. Other hash map implementations which require less memory can alternatively be used as base.

One of the several new memory-efficient Writable hash maps, the specialized IntToPathJoinBindingsMapWritable, is presented in Listing 2.20. It requires that all keys are of type IntWritable and that values are of type PathJoinBindings. During de-serialization, read fields from the given input must provide bytes for objects with these types. In addition to using IntToPathJoinBindingsMapWritable for the *pathJoinMap* of a vertex, a similar self-implemented PropertiesMapWritable is used for *properties* of vertices and edges. It requires that all keys are of type LongWritable and that values are of either type LongWritable or Text. When written to a DataOutput object during serialization, the bytes for each value have to be prefixed with a character which identifies the type.

Listing 2.20: Memory-Efficient Specialized Writable Map

```
public class IntToPathJoinBindingsMapWritable
    extends HashMap<IntWritable, PathJoinBindings> implements Writable {

    public void readFields(DataInput in) throws IOException {
        int count = in.readInt();
        clear();
        for (int i = 0; i < count; i++) {
            IntWritable key = new IntWritable();
            key.readFields(in);
            PathJoinBindings value = new PathJoinBindings();
            value.readFields(in);
            put(key, value);
    }}

    public void write(DataOutput out) throws IOException {
        out.writeInt(size());
        for (Entry<IntWritable, PathJoinBindings> entry : entrySet()) {
            entry.getKey().write(out);
            entry.getValue().write(out);
}}}
```

2.3.3 Replacing Writable ArrayList Structures with Writable Arrays

All previous use of the ArrayList class is replaced with utilization of constant sized arrays. The LongArrayWritable class, presented in Listing 2.21, is implemented to be used instead of the LongArrayListWritable class. Such array should contain *labels* for every vertex. The MessageBinding class, presented in Listing 2.22, is reimplemented to use an array for columns. Currently, not all possible column types are supported. Instead of writing the relevant class ID as a prefix for each column's bytes during serialization, a character identifying the relevant supported type is written.

Listing 2.21: Writable Array for Long Values

```
public class LongArrayWritable extends ArrayWritable {
    public LongArrayWritable() {
        super(LongWritable.class);
    }

    public LongArrayWritable(LongWritable[] values) {
        super(LongWritable.class, values);
    }
}
```

Listing 2.22: MessageBinding with Columns in an Array

```
public class MessageBinding implements Writable {
    private Writable[] columns;

    public void readFields(DataInput in) throws IOException {
        int count = in.readInt();
        columns = new Writable[count];
        for (int i = 0; i < count; i++) {
            char type = in.readChar();
            Writable value = null;
            if (type == 'B') {
                value = new BooleanWritable();
            } else if (type == 'L') {
                value = new LongWritable();
            } else if (type == 'N') {
                value = NullWritable.get();
            } else if (type == 'T') {
                value = new Text();
            } else if (type == 'V') {
                value = new VertexId();
            } else {
                throw new RuntimeException(
                    "Can not read column with type char " + type);
            }

            value.readFields(in);
            columns[i] = value;
    }}

    public void write(DataOutput out) throws IOException {
        out.writeInt(columns.length);
        for (int i = 0; i < columns.length; i++) {
            if (columns[i] instanceof BooleanWritable) {
                out.writeChar('B');
            } else if (columns[i] instanceof LongWritable) {
                out.writeChar('L');
            } else if (columns[i] instanceof NullWritable) {
                out.writeChar('N');
            } else if (columns[i] instanceof Text) {
                out.writeChar('T');
            } else if (columns[i] instanceof VertexId) {
                out.writeChar('V');
            } else {
                throw new RuntimeException("Can not write column with type "
                                           + columns[i].getClass().getName());
            }

            columns[i].write(out);
    }}

    ...
}
```

Chapter 3

Evaluation

This chapter gives an overview of the real effects of the new Lighthouse optimizations. First, in section 3.1, the experimental setup for the performed measurements is described. Next, in sections 3.2 and 3.3, performance results for each previously explained optimization are presented and commented. All evaluation of the implemented optimizations is split into two sections. The first targets left-deep query plans, while the second copes with bushy query plans. The measurements show that pipelining can allow multiple types of plans to be executed with less memory, compared to when using the reference Lighthouse implementation. The pipelined executions of bushy query plans expose problems of predicting the numbers of messages to be passed from PathJoin operators.

3.1 Experimental Setup

Here is an overview of the hardware, input graph, memory consumption measurement method and reference Lighthouse implementation used in the carried out experiments. A discussion of two common pitfalls when running pipelined Lighthouse then follows. By being aware of these pitfalls when setting configurations for Lighthouse, more pipelined executions can be permitted to finish.

3.1.1 SURFsara Hathi Hadoop Cluster

All experiments take place on the SURFsara Hathi Hadoop cluster[1]. It consists of around 100 machines and runs Hadoop 2.6. Each machine has 16 CPU cores with a clock speed of 2.6 GHz, 32 GB memory and 1 TB with local disk space. The Giraph framework with release version 1.1.0 is used on the cluster, set with its `hadoop_2` profile limiting the maximum size of every Java heap to 6.7 GB. Many jobs which are normally failing with this amount of memory should ideally succeed with the new Lighthouse optimizations enabled.

3.1.2 10K LDBC-SNB Data

A dataset generated with the LDBC-SNB Data Generator[2] is provided as input graph to all performed measurements. The generator is designed to create graphs which simulate social networks with characteristics specified via user set arguments[1]. A graph's size is determined by its number of contained persons and the number of years its simulated social network has existed. The input graph used in the following measurements has been generated as a social network of 10'000 persons, which has existed for 3 years, starting from 2010. This dataset requires only 450 MB of disk space to be stored, but with reference Lighthouse a total of 8 GB memory after being loaded onto Hadoop workers. The graph contains 1'030'132 vertices and 12'357'981 edges, making an average of 11 edges per vertex. There is skew in the dataset. As in the real world, some companies have magnitudes more employees than others, some persons have more social relations than others.

[1] https://userinfo.surfsara.nl/systems/hadoop/hathi
[2] https://github.com/ldbc/ldbc_snb_datagen

3.1.3 Method for Measuring Memory Consumption

To provide information about memory consumption during experiments, Lighthouse can create extensive logs with numbers measured via the Giraph API's MemoryUtils class. By workers requesting garbage collection before calling this class' methods, accurate consumption numbers are returned. In addition to when performing experiments, proper memory logs are helpful when adjusting pipelined Lighthouse's memory consumption prediction behavior. To decrease execution times with non-experiment versions of Lighthouse, the number of garbage collection requests are reduced, causing inaccurate consumption numbers in the memory logs. Alternatively, a system monitor such as Ganglia[3] or a performance evaluation framework like Granular[11] can be used.

3.1.4 Reference Implementation of Lighthouse

Different versions of Lighthouse with optimizations are compared to a reference implementation. This implementation contains the basic functionality for correct query processing and allows a query plan to be provided by the user via a file in the HDFS. It starts all the query paths of the plan, for every vertex of the given input graph, in the first superstep of the initiated execution.

3.1.5 Considered Pitfalls for Pipelined Lighthouse Executions

Based on experience from previously performed experiments, there are two common pitfalls for pipelined Lighthouse executions which often can be avoided with set configurations:

1. workers running out of physical memory
2. garbage collection on workers reaching the overhead limit

If the calculated initial message limits in an execution are too large, workers may run out of physical memory. This situation is often caused by the use of a small default initial message limit in early supersteps, leading to bad predictions of how many messages will be passed by the operators of the handled query plan. Such problem might be solved by increasing the default initial message limit. The disadvantage of having a high default limit value is that workers may not have enough memory to succeed the initial supersteps. Even with a suitable default limit and a large skew protection fraction size, skew in input graphs may still cause certain executions to unexpectedly run out of memory. This must be handled by Lighthouse's restart- or continuation mechanism.

The execution of a query may fail if the garbage collector on a used worker is spending a too large fraction of the CPU time recovering memory. This can often be prevented by increasing the skew protection fraction size, ensuring that the physical memory is less frequently filled up. The harms of having too much skew protection memory is that more supersteps are then required to finish an execution, in addition to less space being available for storing the input graph and join tables. Optionally, garbage collectors can be configured to not stop executions when they are heavily used.

3.2 Evaluation of Executions of Left-Deep Query Plans

Every query plan is from now on referred to with the number of the listing it is presented in. To test Lighthouse's performance with the new optimizations on left-deep query plans, the two query plans 3.1 and 3.2 are considered. They are suitable for showing the optimizations' effects, since they force workers to produce many messages. An execution may properly start to run on a cluster with little available memory, but then fail when the amount of present messages in the system grow. Most of the implemented optimizations set extra limitations for the numbers of stored messages in the system at a time, preventing workers from running out of memory. Both the query plans have similar properties as ones for path queries (described in appendix D) with multiple subsequent StepJoin operators. The two query plans also have some significant differences, increasing the possibility to find weaknesses for Lighthouse with the various optimizations.

[3] http://ganglia.sourceforge.net/

Query Plan 3.1 starts with a Scan operator performing a filter on every vertex of the input graph. Only vertices with the label 'Person' continue to compute the next StepJoin operator. For each of these vertices, a check for whether there is a chain of three relationships labeled 'KNOWS', leading to someone with the name Antonio, is carried out. This query plan can be used in an execution of a path query, trying for every person in the input graph to find the closest related persons named Antonio. It must be executed after similar plans with one and two StepJoin operators instead.

Query Plan 3.2 leads to a significantly smaller amount of messages being produced than Query Plan 3.1. It has an early filter with the name of the first person in its solution pattern. This query plan is supposed to cause difficulties for predictions for memory consumption in certain pipelined executions, in which workers in the earliest supersteps use a low default initial message limit. After new initial message limits are calculated and used, some workers might produce very large numbers of initial messages. They may not have experienced that any messages pass the first operator.

Listing 3.1: Left-Deep Query Plan without Small Out-In Ratio for First Operator

```
Select(StepJoin(StepJoin(StepJoin(Scan(Person),
                                  KNOWS),
                         KNOWS),
                KNOWS),
       =({firstName}, Antonio));
```

Listing 3.2: Left-Deep Query Plan with Small Out-In Ratio for First Operator

```
Select(StepJoin(StepJoin(StepJoin(Scan(firstName:"John"),
                                  KNOWS),
                         KNOWS),
                KNOWS),
       =({firstName}, John));
```

3.2.1 Reference- versus Pipelined Executions

Following are comparisons of Lighthouse with and without pipelining.

Memory Consumption with Query Plan 3.1

In an environment where every worker has only 6 GB of available memory, an execution of Query Plan 3.1 with the reference implementation of Lighthouse requires a minimum of 70 workers to finish successfully. The graph in Figure 3.1 shows the memory consumption on the 10 workers with the lowest IDs after each superstep. The "Memory limit" line visualizes the total amount of available memory on each worker. The "Basis consumption" line represents the average worker memory consumption just after the input graph has been loaded, before any messages are created.

Only 4 supersteps are required to finish the execution, which naturally is one more than the number of global operators in the query plan. With 70 used workers, the amount of memory required on each to store a local graph partition is very low. On most workers, less than 130 MB. The highest memory consumption on all the workers is reached right before starting the last superstep. This is a result of the exponential increase for the numbers of messages passed per StepJoin. Despite the maximum memory consumption on the workers is only about 80 percent of the "Memory limit", it is not possible to execute the query plan with slightly less workers. The execution of the plan will then fail, since the garbage collector on some workers is spending a too large fraction of the CPU time recovering memory (just small amounts on each run). If the garbage collectors are configured to not stop the execution when reaching the overhead limit, the execution will finish slowly.

Pipelined Lighthouse with dynamic initial message limits can successfully complete an execution of Query Plan 3.1 on 4 workers with 6 GB of available memory. The graph in Figure 3.2 shows the memory consumption on all the workers after each superstep. It is clear that with pipelining, the query plan can be executed on a much smaller cluster. Using dynamic initial message limits, Lighthouse regularly improves every worker's initial message limit. The "Defensive limit" line represents the available memory on each worker, but not including any skew protection memory.

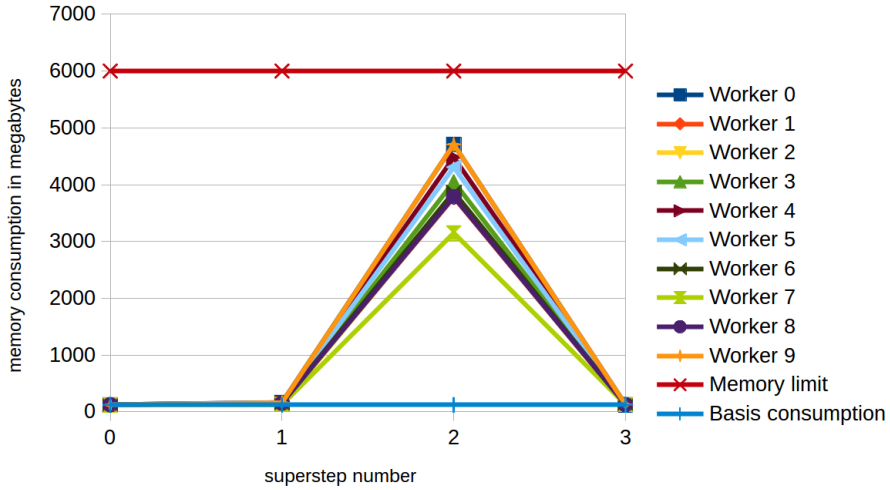


Figure 3.1: Memory consumption for reference execution of Query Plan 3.1

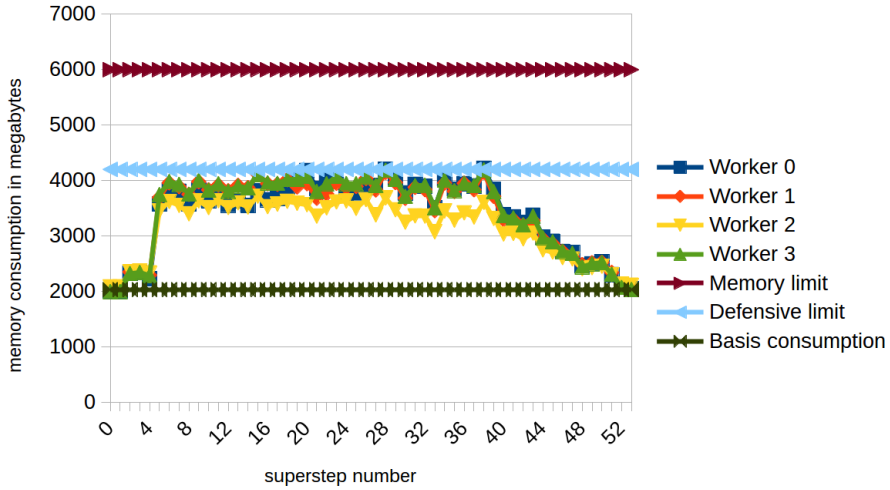


Figure 3.2: Memory consumption for pipelined execution with dynamic limits of Query Plan 3.1

Instead of requiring 4 supersteps, 54 supersteps are needed. As the skew protection fraction size is set to 0.3, the "Defensive limit" is 70 percent of the "Memory limit". This is supposed to prevent failures with workers running out of memory or garbage collectors dominating workers' CPU use. The amount of utilized memory on each worker stays close to the "Defensive limit", showing that the predictions for memory consumption are accurate. All use of memory is low in the beginning of the execution, since only a few messages are produced in the first supersteps with the small default initial message limit of 1'000. The memory consumption on the workers is slowly decreasing in the end of the execution, after more workers have created their last initial messages. The workers finish their production of initial messages in different supersteps. Worker 1 first stops to produce initial messages in superstep 37, while Worker 2 finally stops to produce initial messages in superstep 50. These variations are a consequence of skew in the input data. The workers do not store the same number of 'Person' labeled vertices, these also have various numbers of relationships. Some workers handle more messages than others, causing calculation of different initial message limits.

Pipelined Lighthouse with static initial message limits can also successfully complete an execution of Query Plan 3.1 on 4 workers with 6 GB of available memory. The graph in Figure 3.3 shows the memory consumption on all the workers after each superstep. As explained in the Lighthouse changes chapter, static initial message limits are oppositely from dynamic limits never changed after they have been calculated and used.

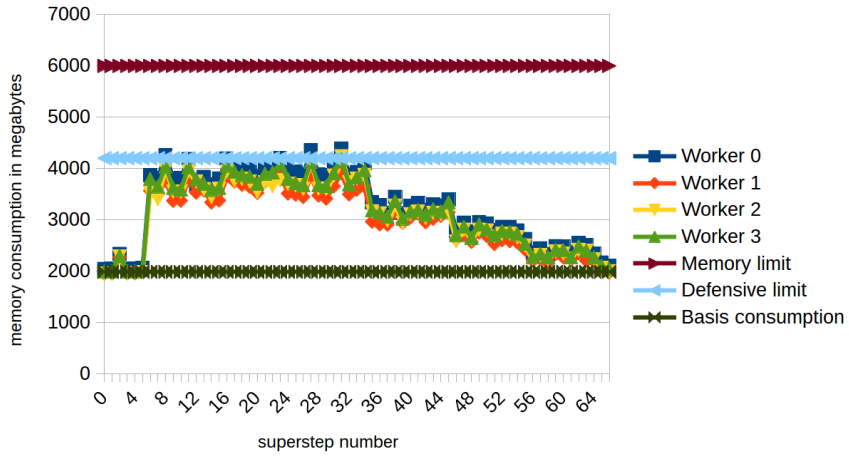
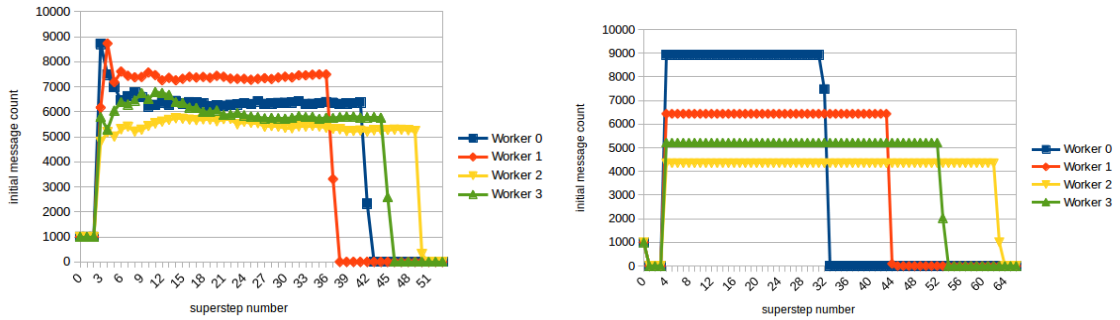


Figure 3.3: Memory consumption for pipelined execution with static limits of Query Plan 3.1

Instead of requiring 4 supersteps, 67 supersteps are needed. Some workers occasionally consume slightly more memory than the "Defensive limit". However, the calculated static initial message limits are sufficient to prevent all the workers from being close to run out of memory. The differences in memory consumption between the executions with static and dynamic message limits are relatively small, since the used default initial message limit of 1'000 is large enough to make both perform accurate memory consumption predictions. There are still some noteworthy variations between the executions. The execution with static initial message limits requires more supersteps to finish. Also, it starts to decrease memory consumption earlier. This is caused by the larger differences among the calculated initial message limits. Worker 0 first stops to produce initial messages in superstep 32, while Worker 2 finally stops to produce initial messages in superstep 63.

In Figure 3.4, the different numbers of initial messages produced by workers per superstep during executions with dynamic and static initial message limits are showed. As clearly visible, no initial messages are created on a worker in a superstep after all its path computations have started.



(a) With dynamic initial message limits

(b) With static initial message limits

Figure 3.4: Numbers of initial messages produced per superstep for executions of Query Plan 3.1

With pipelined Lighthouse using dynamic initial message limits, both Worker 0 and Worker 1 quickly reduce their production of initial messages. Their dynamic initial message limits improve while the workers get a better overview of what messages are created with the provided input. This behavior helps to avoid consumption of more memory than the "Defensive limit". In some cases, these adjustments can also prevent workers from running out of memory or garbage collectors from reaching the overhead limit. With pipelined Lighthouse using static initial message limits, Worker 0 calculates a much larger initial message limit than the other workers, based on statistics gathered after its first path computations. It does not have the opportunity to reduce this static limit at a later stage. The effect of the large limit is not fatal for any worker's memory consumption, since the other workers calculate smaller initial message limits. Additionally, the messages produced from limits are distributed across all the used workers, spreading their memory impact.

The duration of each execution of Query Plan 3.1 was here affected by extra requested garbage collection, taking place prior to calling Giraph's MemoryUtils methods before and after every superstep. The execution with the reference implementation of Lighthouse, involving 70 workers, lasted for 20 minutes and 21 seconds. Pipelined Lighthouse with dynamic initial message limits, using only 4 workers, finished after 187 minutes and 44 seconds. On a similar number of workers, pipelined Lighthouse with static limits needed 199 minutes and 18 seconds to complete.

Memory Consumption with Query Plan 3.2

On a cluster of 4 workers with 3 GB of available memory, an execution of Query Plan 3.2 with the reference implementation of Lighthouse will fail. The graph in Figure 3.5 shows the memory consumption after each superstep, with every worker instead having 8 GB of available memory. The "Memory limit" line visualizes the 3 GB of memory on each worker of the mentioned cluster.

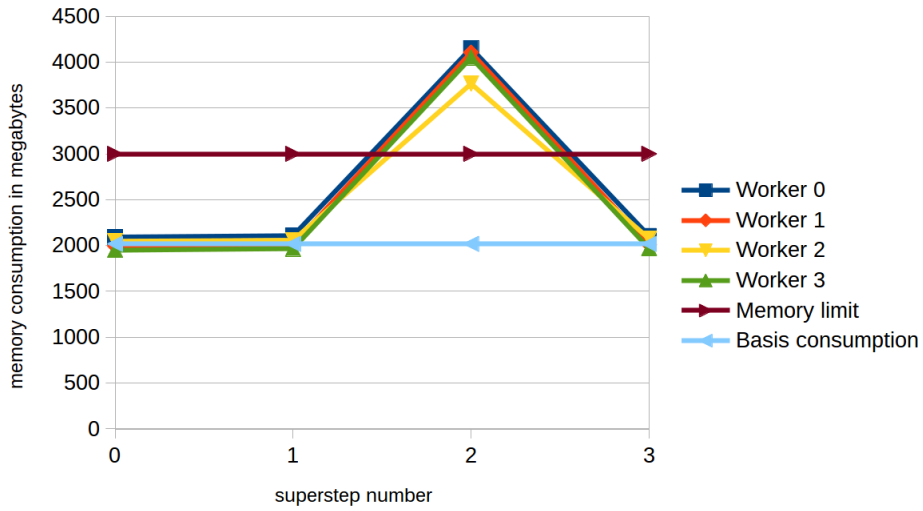


Figure 3.5: Memory consumption for reference execution of Query Plan 3.2

As for the reference execution of Query Plan 3.1, 4 supersteps are required to finish the execution. With 4 used workers, the amount of memory required on each to store a local graph partition is around 2 GB. The highest memory consumption on all the workers is reached right before starting the last superstep. This is again a result of an exponential increase for the numbers of messages passed per StepJoin. The maximum memory consumption on the workers is just above 4 GB, which is far more than the 3 GB "Memory limit" of the imagined cluster.

Pipelined Lighthouse with dynamic initial message limits can successfully complete an execution of Query Plan 3.2 on 4 workers with 3 GB of available memory. The graph in Figure 3.6 shows the memory consumption on all the workers after each superstep. It is clear that with pipelining, the query plan can be executed on the previously mentioned cluster with 3 GB of memory per worker.

Instead of requiring 4 supersteps, 17 supersteps are needed. The "Defensive limit" is 80 percent of the "Memory limit", leaving 600 MB of memory as skew protection on every worker. The amount of utilized memory on each worker stays again close to the "Defensive limit", emphasizing that the predictions for memory consumption are accurate. The memory use on the workers reaches a maximum before superstep 6, after the workers have started to produce initial messages according to larger calculated limits in superstep 3. The effect of the increased production of initial messages requires a couple of supersteps to show, since the exponential increase for the numbers of present messages takes place via the global StepJoin operators. Worker 3 first stops to produce initial messages in superstep 8, while Worker 0 finally stops to produce initial messages in superstep 12.

Pipelined Lighthouse with static initial message limits can not execute Query Plan 3.2 on 4 workers with 3 GB of available memory, using default pipelining configurations. The graph in Figure 3.7 shows the memory consumption on all the workers after each superstep in a failed execution.

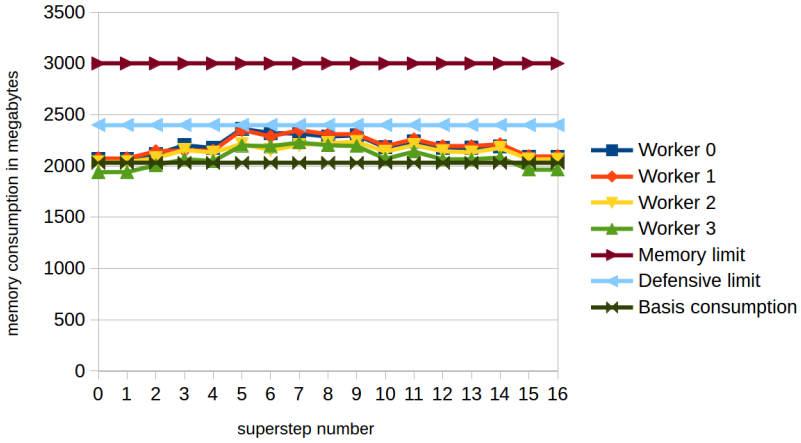


Figure 3.6: Memory consumption for pipelined execution with dynamic limits of Query Plan 3.2

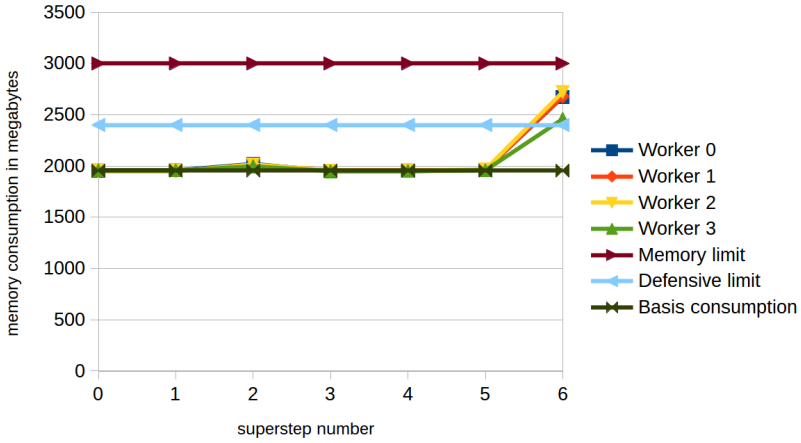
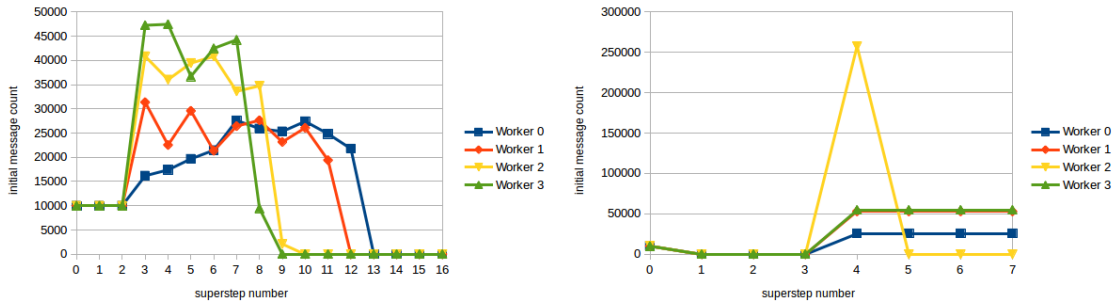


Figure 3.7: Memory consumption for pipelined execution with static limits of Query Plan 3.2

All the utilized workers run out of memory in superstep 7. The large memory consumption in this superstep is a consequence of a previous high number of initial message creations. In Figure 3.8b, the different numbers of initial messages produced by workers per superstep during the execution are showed. As clearly visible, the initial message production on Worker 2 in superstep 4 is huge. This is caused by a calculated initial message limit which is much larger than the ones used by the other workers. Since Worker 2 never experiences that vertices pass the initial filter of the handled query plan during the earliest supersteps, it assumes it can immediately start path computations on all its vertices without any significant effects on the future memory consumption. The problem can be avoided by using a higher default initial message limit. Each worker then has an increased possibility of experiencing that vertices pass the initial filter in the earliest supersteps, resulting in an improved accuracy for memory consumption predictions in the next performed limit calculation.

As apparent in Figure 3.8, the default initial message limit value used during the executions of Query Plan 3.2 with dynamic and static initial message limits is 10'000. This default limit value is handled differently by the two versions of pipelined Lighthouse. With pipelined Lighthouse using dynamic limits, each worker in every superstep before the first calculations of limits produces the default number of initial messages. With pipelined Lighthouse using static limits, each worker only produces the default number of initial messages in the first superstep. In the remaining supersteps before the calculations of limits, no initial messages are created. For Query Plan 3.2, each worker in the execution with dynamic limits tests the initial filter on 30'000 vertices before its first limit calculation. In the execution with static limits, each worker only tests the initial filter on 10'000 vertices before its limit calculation. This variation is large enough to make the execution with static limits fail, after some bad memory consumption predictions by a single worker.



(a) With dynamic initial message limits

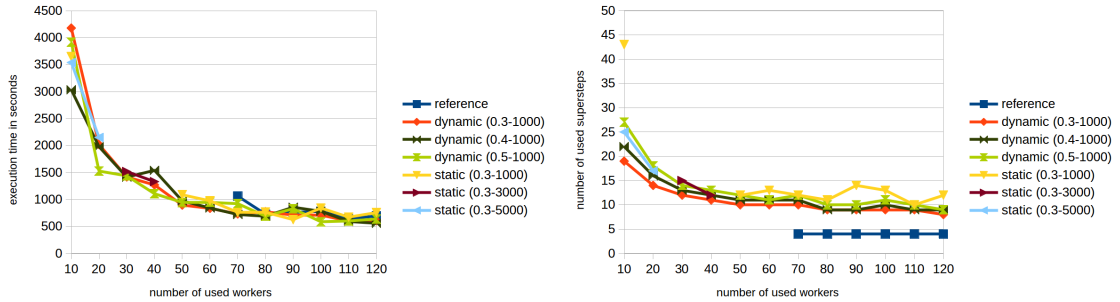
(b) With static initial message limits

Figure 3.8: Numbers of initial messages produced per superstep for executions of Query Plan 3.2

The duration of each execution of Query Plan 3.2 was here affected by extra requested garbage collection. The execution with the reference implementation of Lighthouse, involving 4 workers, lasted for 17 minutes and 11 seconds. Pipelined Lighthouse with dynamic initial message limits, also involving 4 workers, finished after 17 minutes and 35 seconds.

Execution Times with Query Plan 3.1

Following are execution time comparisons of Lighthouse with and without pipelining. Each worker used in the measurements, presented in Figure 3.9, has 6 GB of available memory. The execution times and numbers of required supersteps are measured with various numbers of workers. Every execution is performed with a minimal number of garbage collection requests. The pipelined setups are named *<calculation pattern>(<skew protection fraction size>-<default initial message limit>)*.



(a) Execution times

(b) Numbers of required supersteps

Figure 3.9: Time and number of supersteps for executions of Query Plan 3.1

The execution times achieved by Lighthouse with Query Plan 3.1 are visualized in Figure 3.9a. The pipelined setups perform relatively good compared to the reference Lighthouse implementation. When doubling the number of used workers, an execution time can at best be expected to halve. In most cases, the execution time decreases far less, specially when running on many workers. When using 120 workers instead of 40, the execution times are just roughly halved, despite having 3 times more workers. Computation and communication times are in subsection 1.3.4 described to scale quite well in Lighthouse. With large amounts of workers, other factors may prevent better speedups, such as slow loading of the input graph from the HDFS to the memory of the workers.

For every number of workers the reference implementation is observed to successfully execute with, it is not faster than all the pipelined Lighthouse setups. With 70 workers, it performs significantly worse. This is probably caused by it carrying out a higher amount of slow garbage collection, since the memory of the workers gets nearly filled up with messages. Even though reference Lighthouse uses far less supersteps than the pipelined setups, the achieved execution time is relatively weak.

As showed in Figure 3.9b, the number of supersteps required in executions with static initial message limits is commonly higher than the number required in executions with dynamic limits. The reason for this is simple. Even if only a single worker uses a low static initial message limit, it will make the whole execution require more supersteps. An increase of the skew protection fraction size is also introducing additional supersteps. Extra supersteps are not observed to result in worse execution times. This can be utilized by performing extremely defensive memory predictions during limit calculations, reducing the possibility for running out of memory without apparent overhead.

3.2.2 Pipelined Executions with Restart or Continuation on Failure

Following are measurements showing the effects of the abilities to restart or continue failed executions. Query Plan 3.1 is executed with pipelined Lighthouse using a skew protection fraction size of 0, not utilizing any skew protection memory at all. The previously presented "Defensive limit" is without the skew protection equal to the "Memory limit". This is supposed to make some workers run out of memory, at least once during a job execution. Each of the 4 used workers in the measurements has 6 GB of available memory. The default initial message limit is set to 1'000.

In Figure 3.10, the memory consumption during an execution of pipelined Lighthouse with dynamic initial message limits and support for automatic restart of failed executions is visualized.

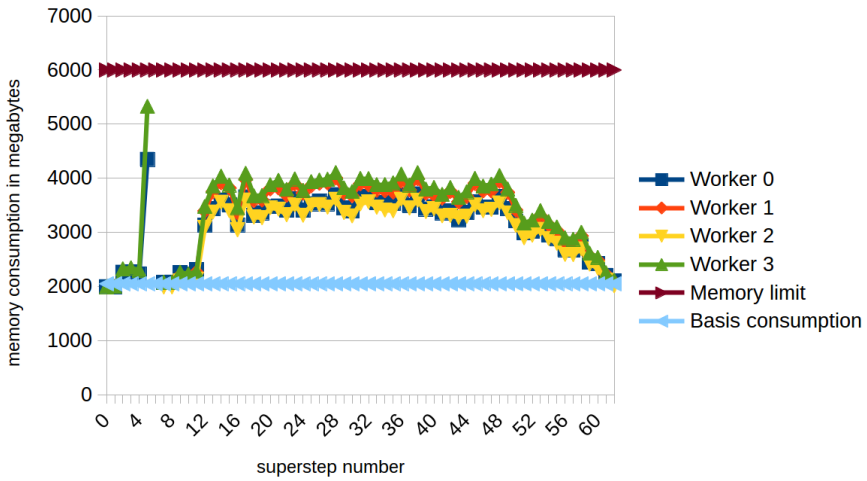


Figure 3.10: Memory consumption for pipelined execution with restart of Query Plan 3.1

The memory consumption numbers from workers 1 and 2 for the first execution attempt were not added to the execution's memory logs. However, the numbers from the other workers in the attempt show a very high memory consumption. The execution fails in superstep 6, as some workers run out of memory or spend too much time on performing garbage collection. After the failed attempt, all the solutions written to the HDFS are deleted. The execution is then reattempted with a retry factor affected by the the attempt number. Unavoidably, the restart causes the total number of needed supersteps to increase. The memory consumption is low in the earliest supersteps after the restart, since the initial message limits used in these supersteps are based on the default limit. As more messages are produced with calculated limits, the memory consumption gets stabilized around 4 GB on the workers. Following the restart, the apparent memory consumption patterns are similar to the ones seen for the pipelined execution of Query Plan 3.1 presented in Figure 3.2. In that execution, 1800 MB of memory was reserved for skew protection on each worker.

In Figure 3.11, the memory consumption during an execution of pipelined Lighthouse with dynamic initial message limits and support for automatic continuation of failed executions is visualized.

Again, the memory consumption numbers from multiple workers for the first execution attempt were not added to the execution's memory logs. The attempt fails in superstep 6, after a too high memory consumption on the utilized workers. All the solutions written to the HDFS before and during superstep 5 are not deleted prior to starting the next execution attempt. The total number

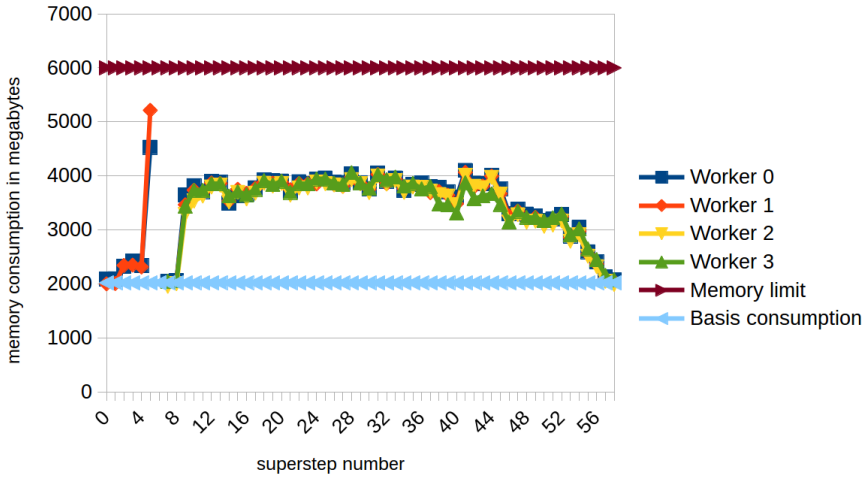
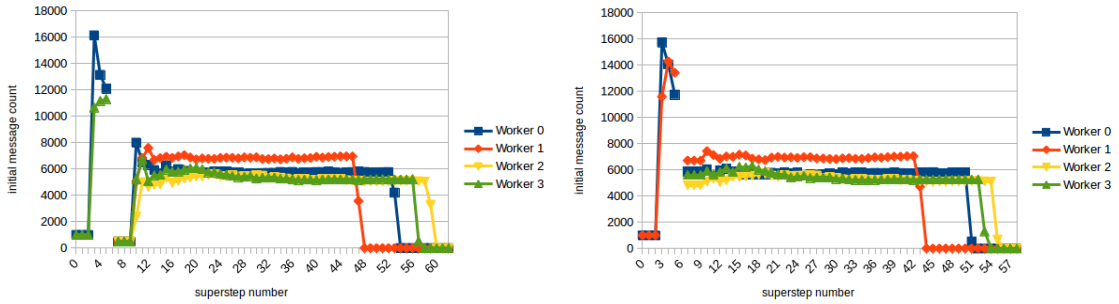


Figure 3.11: Memory consumption for pipelined execution with continuation of Query Plan 3.1

of supersteps required to finish is not significantly reduced by avoiding to reproduce these solutions. If a failure takes place at a later point, the continuation mechanism can drastically decrease the number of needed supersteps compared to when using the restart mechanism. Because the second execution attempt considers the previous attempt's initial message limits, it can immediately start to produce many initial messages. This is clearly showed in Figure 3.12, where it is visible that the restart implementation of Lighthouse goes back to use of a small limit based on the default limit value, while the continuation implementation does not. The way the continuation mechanism sets an attempt's first initial message limits is not always optimal. A bad limit may mislead the next attempt and potentially cause another failure. The continuation mechanism enables the execution to finish after 59 supersteps. When the restart mechanism was used, 63 supersteps were needed.



(a) With restart of failed attempts

(b) With continuation of failed attempts

Figure 3.12: Numbers of initial messages produced per superstep for executions of Query Plan 3.1

The duration of each execution of Query Plan 3.1 was here affected by extra requested garbage collection. The last execution attempt by pipelined Lighthouse with dynamic initial message limits and the restart mechanism, involving 4 workers, lasted for 191 minutes. The total time for the execution was 211 minutes and 57 seconds. The last execution attempt by pipelined Lighthouse with dynamic limits and the continuation mechanism, also involving 4 workers, finished after 166 minutes and 32 seconds. The total time for the execution was 187 minutes and 45 seconds.

3.2.3 Evaluation of Improved Data Structures and Serialization Formats

Following are measurements showing the effects of the presented changes to the data structures and serialization formats used for storing vertices, edges and messages. Query Plan 3.2 is executed on 4 workers with 8 GB of available memory. The graph in Figure 3.13 shows the average memory consumption on the workers after each superstep, with and without the new structures and formats.

The memory consumption of the listed versions of Lighthouse, each utilizing different data structures and serialization formats, should be interpreted in a sequence. The “Null vertex” implementation is similar to reference Lighthouse, but it also includes the “Null vertex” change. The “Merge vertex” implementation is as well similar to reference Lighthouse, but it includes both the “Null vertex” and the “Merge vertex” changes. This pattern continues and ends with “Avoid message”. “Full” represents reference Lighthouse with all of the new data structures and serialization formats.

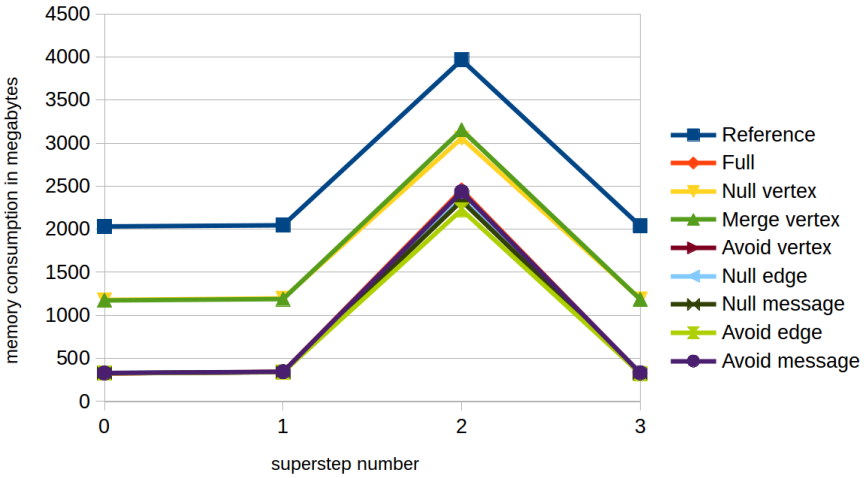


Figure 3.13: Memory consumption for executions with storage optimizations of Query Plan 3.2

Only a few of the changes lead to any significant impact on Lighthouse’s memory consumption. The “Null vertex” implementation, which instead of referencing empty data structures in vertex values uses null references, reduces the memory consumption for storing the input graph by 42 percent compared to reference Lighthouse. This makes more memory available for storing messages. Vertex values are at different times during an execution stored either as Java objects or in a serialized form. The “Avoid vertex” change, which prevents use of bad data structures in vertex values, such as MapWritable for vertex properties, further reduces the memory consumption for storing the input graph by 72 percent. None of the changes reduce the memory needed for storing messages significantly. Totally, the memory consumption for storing the input graph is reduced by 84 percent.

The execution with the reference implementation of Lighthouse, involving 4 workers, lasted for 7 minutes and 6 seconds. The Lighthouse version with all the data structure and serialization format changes enabled, also involving 4 workers, finished after 2 minutes and 41 seconds. These numbers indicate that a reduced memory consumption for storing the input graph during a Lighthouse execution has a positive effect on its duration. This may be a consequence of workers spending a smaller time loading the input graph into their memory and less triggered garbage collection.

3.3 Evaluation of Executions of Bushy Query Plans

To test Lighthouse’s performance with the new optimizations on bushy query plans, the two query plans 3.3 and 3.4 are considered. They are suitable for showing the optimizations’ effects, since they force workers to produce many messages. Both of the query plans have a single PathJoin operator, but only one of them has a StepJoin operator following the PathJoin. The differences between the plans increase the possibility of finding weaknesses for the implemented optimizations.

Query Plan 3.3 can not cause creations of complete solutions, but it serves to trigger a high memory consumption. It starts with Scan operators performing a filter on every vertex of the input graph. Because the initial filters are equal, if a vertex passes the Scan operator of one of the query paths, it also passes the Scan operator of the other. All ’Person’ labeled vertices are passed through. The next StepJoin operators produce messages for connected chains of two relationships labeled ’KNOWS’. The messages with IDs for the involved vertices are joined by the PathJoin on receiving vertices. Every joined message is filtered away to ensure that no solutions are written to the HDFS.

Query Plan 3.4 allows messages to be transferred among workers after its PathJoin is computed, via a StepJoin operator. The two query paths of the plan start similarly as in Query Plan 3.3. However, before the PathJoin is computed, messages are only sent to vertices with one 'KNOWS' labeled relationship in between. Every joined message represents a non-complete solution with a single group of persons knowing each other. For each joined message, the last StepJoin operator creates and sends messages to vertices for companies where the Person joined on works. It should be difficult to guess the numbers of solutions which are passed by the PathJoin. This can potentially lead to inaccurate predictions of how much memory is needed to store messages on the last StepJoin.

Listing 3.3: Bushy Query Plan without StepJoin Operator after PathJoin

```
Select(PathJoin(StepJoin(StepJoin(Scan(Person),
                                  KNOWS),
                         KNOWS),
                StepJoin(StepJoin(Scan(Person),
                                  KNOWS),
                         KNOWS)),
       =({firstName}, NULL));
```

Listing 3.4: Bushy Query Plan with StepJoin Operator after PathJoin

```
Select(StepJoin(PathJoin(StepJoin(Scan(Person),
                                  KNOWS),
                         StepJoin(Scan(Person),
                                  KNOWS)),
                WORK_AT),
       =({name}, President_Airlines));
```

3.3.1 Forced Simultaneous Arrivals of Messages from Both Paths

Following are measurements showing the effect of forcing query path computations to be started so that all messages from paths arrive at a PathJoin simultaneously. At the end of a superstep in which messages are received with a PathJoin, all related join tables are deleted. Query Plan 3.3 is executed on 10 workers with 6 GB of available memory. The graph in Figure 3.14 shows the average memory consumption on the workers after each superstep, with and without the optimization.

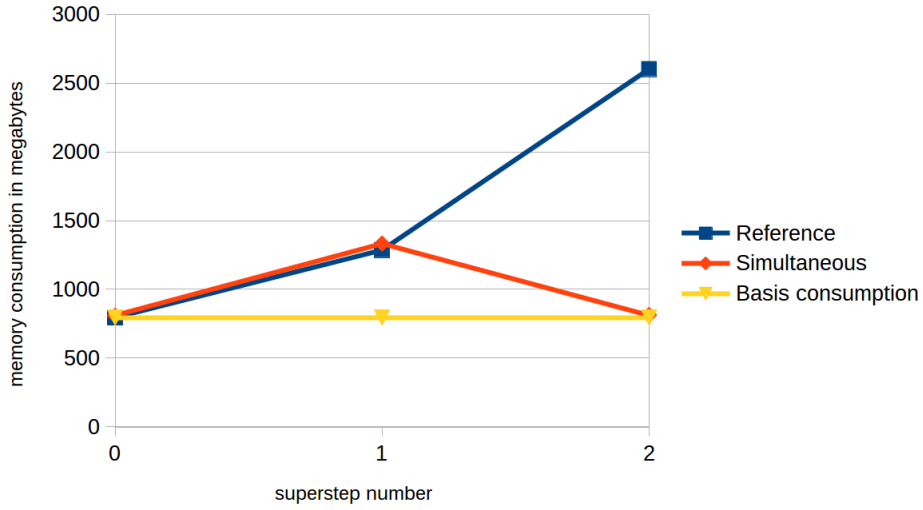


Figure 3.14: Memory consumption for execution with simultaneous arrivals of Query Plan 3.3

The effect of the simultaneous arrivals optimization shows later in the query plan execution, when the numbers of messages stored in the PathJoin tables are normally high. The required memory by the "Simultaneous" implementation, after the last superstep, is the same as when no messages are present in the system. During its execution, the memory consumption is at one point in the

last superstep similar to the highest visible memory consumption of reference Lighthouse. This is not apparent in the graph, since all join tables are deleted just before the end of the superstep. Lighthouse executions are with the optimization not having a reduced memory consumption for a PathJoin operator in the superstep it is computed. Though, following computations in the next supersteps benefit with more available memory for storing messages. The reference implementation of Lighthouse often unnecessarily consumes memory for PathJoin tables until the end of executions.

The execution with the reference implementation of Lighthouse, involving 10 workers, lasted for 16 hours and 49 minutes. The Lighthouse implementation with simultaneous arrivals of messages, also involving 10 workers, finished after 19 hours and 57 minutes.

3.3.2 Forced Prior Arrivals of Messages from Path Passing Fewest Bytes

Following are measurements showing the effect of forcing query path computations to be started so that all messages from the path which passes the fewest bytes to a PathJoin arrive before any messages from the other path. This allows every PathJoin to exclusively store messages from its smallest incoming path (passes the smallest number of bytes) in related join tables. For different jobs, the possible reduction of needed memory, enabled through use of the optimization, varies with the input graph and the generated query plan. The extent of the reduction depends on the numbers and sizes of messages passed from the query plan's paths. Query Plan 3.3 is executed on 10 workers with 6 GB of available memory. Since both the query paths in the plan are equal and produce similar messages, the stored join tables in the execution should finally consume half the amount of memory compared to with reference Lighthouse. The graph in Figure 3.15 shows the average memory consumption on the workers after each superstep, with and without the optimization.

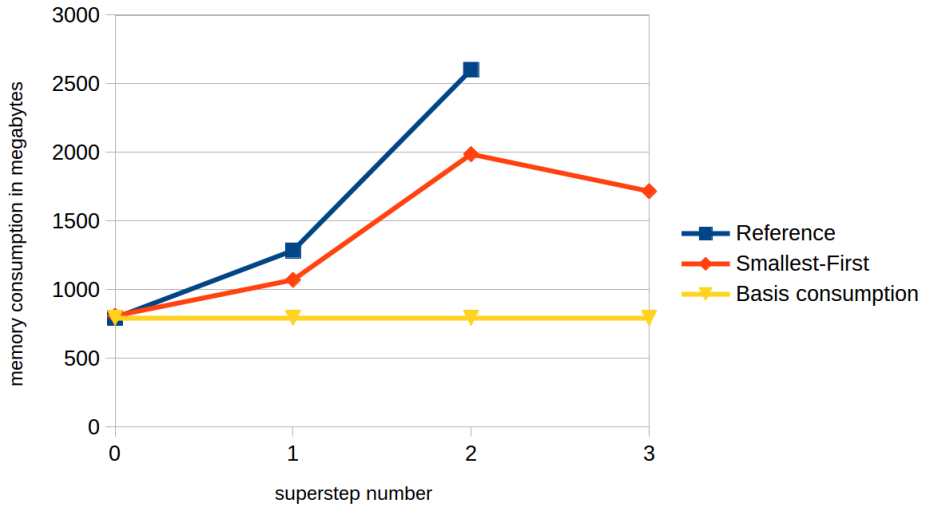


Figure 3.15: Memory consumption for execution with smallest-first arrivals of Query Plan 3.3

The effect of the smallest-first arrivals optimization also shows later in the query plan execution. The "Smallest-First" implementation requires one more superstep, since its computations on the smallest path are delayed. In superstep 1, reference Lighthouse produces almost twice as many messages as the optimized implementation. All these messages are stored in join tables, requiring a total of 1806 MB of memory. With the optimization, the join tables finally require only 908 MB of memory. During the execution of the "Smallest-First" implementation, there is a larger memory consumption right before superstep 3 than after. Prior to this superstep, received messages are stored in addition to the messages in join tables. Following superstep 2, the present tables do not increase in size. Query Plan 3.3 does not contain any non-last global operators after the PathJoin. If this would be the case, the optimized implementation benefit with more available memory.

The execution with the reference implementation of Lighthouse, involving 10 workers, lasted for 16 hours and 49 minutes. The Lighthouse implementation with so-called smallest-first arrivals of messages, also involving 10 workers, finished after 18 hours and 12 minutes.

3.3.3 Storage of Serialized Messages in PathJoin Tables

Following are measurements showing the effect of storing serialized messages in PathJoin tables. The optimization is implemented to reduce join tables' sizes. Additionally, it enables all utilized workers to calculate better initial message limits, using accurate sizes for present PathJoin tables. Query Plan 3.3 is executed on 10 workers with 6 GB of available memory. The graph in Figure 3.16 shows the average memory consumption on the workers after each superstep, with and without the optimization. The reference Lighthouse implementation stores join table messages as Java objects.

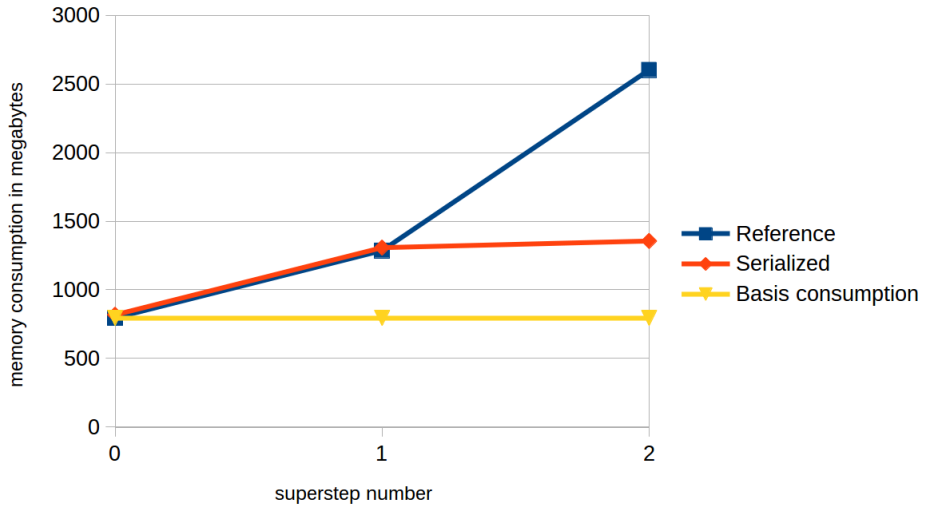


Figure 3.16: Memory consumption for execution with serialized table messages of Query Plan 3.3

The effect of the serialized table messages optimization shows later in the query plan execution. Prior to starting superstep 2, both reference Lighthouse and the "Serialized" implementation consume almost 500 MB of memory for received serialized messages. With the reference implementation of Lighthouse, the received messages are sequentially de-serialized and end up being stored as Java objects in various PathJoin tables. When storing the messages in this manner, the same data which required 491 MB before starting the superstep consumes around 1806 MB. By storing table messages in a serialized form, as done with the "Serialized" implementation, a slight increase of average memory consumption is still experienced. This is likely caused by a sub-optimal memory allocation for serialized messages in the PathJoin tables. Giraph itself stores serialized messages, received from other workers, in special message stores. If non-last global operators are present after a PathJoin, further execution with the optimized implementation have more available memory.

The execution with the reference implementation of Lighthouse, involving 10 workers, lasted for 16 hours and 49 minutes. The Lighthouse implementation with storage of serialized messages in PathJoin tables, also involving 10 workers, finished after 39 hours and 5 minutes.

3.3.4 Reference- versus Pipelined Executions

Following are memory consumption comparisons of Lighthouse with and without pipelining.

Memory Consumption with Query Plan 3.3

As apparent in Figure 3.16, the maximum memory consumption in executions of Query Plan 3.3 is reached in the end of the last superstep, when all messages from both paths are stored in the PathJoin's tables. Pipelining can not reduce this maximum memory consumption. It only affects the numbers of messages passed per superstep, not the numbers of messages stored in join tables.

Memory Consumption with Query Plan 3.4

On a cluster of 4 workers with 4 GB of available memory, an execution of Query Plan 3.4 with the reference implementation of Lighthouse will fail, since the invoked garbage collectors spend a too large fraction of the CPU time recovering memory. The graph in Figure 3.17 shows the memory consumption after each superstep, with every worker instead having 6 GB of available memory. The "Memory limit" line visualizes the 4 GB of memory on each worker of the mentioned cluster.

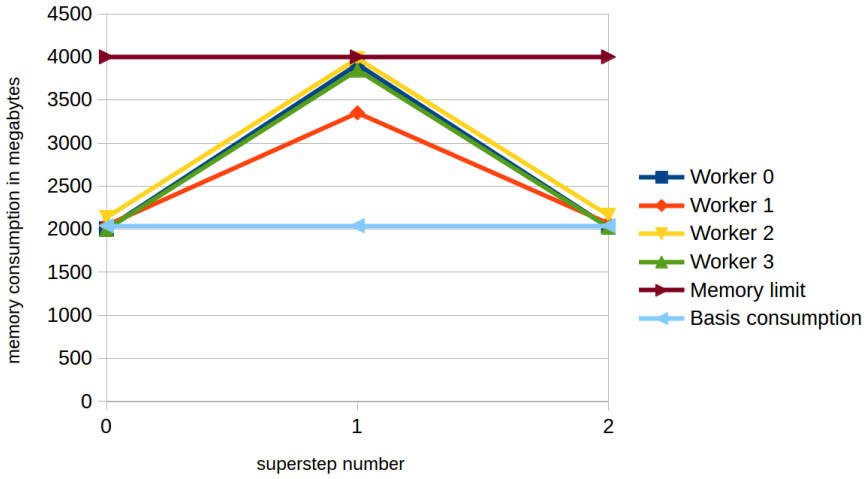


Figure 3.17: Memory consumption for reference execution of Query Plan 3.4

Only 3 supersteps are required to finish the execution, which naturally is one more than the largest number of subsequent global operators in the query plan. With 4 utilized workers, the amount of memory required on each to store a local graph partition is around 2 GB. The highest memory consumption on all the workers is reached right before starting the final superstep, after the last StepJoin operator in the query plan has been computed. The maximum memory consumption on the workers is almost 4 GB, similar to the "Memory limit" of the imagined cluster.

Pipelined Lighthouse with static initial message limits and support for automatic restart on failure can successfully complete an execution of Query Plan 3.4 on 4 workers with 4 GB of available memory. The graph in Figure 3.18 shows the memory consumption on all the workers after each superstep. It is clear that with pipelining and automatic restart on failure, the query plan can be executed on the previously mentioned cluster with 4 GB of memory per worker.

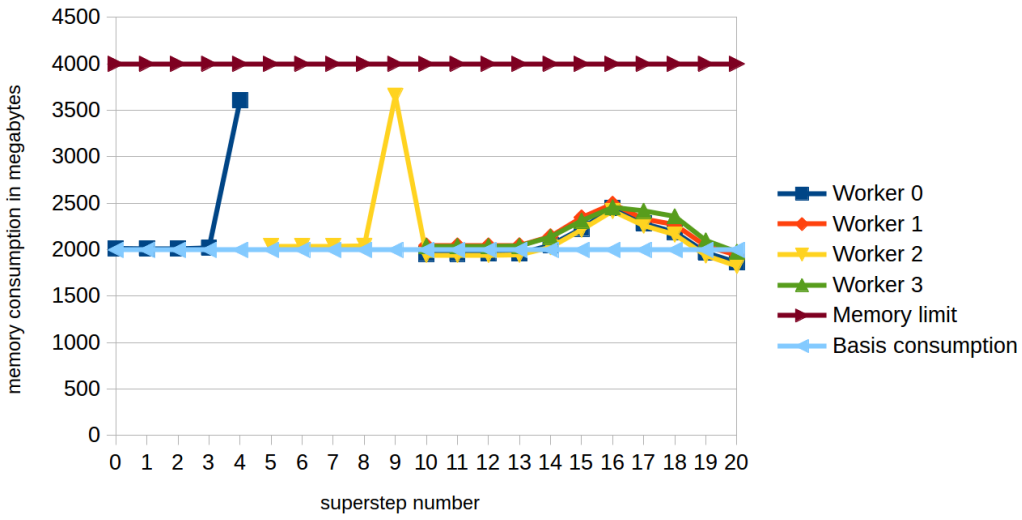


Figure 3.18: Memory consumption for pipelined execution with static limits of Query Plan 3.4

Instead of requiring 3 supersteps, 21 supersteps are needed, spread over 3 execution attempts. In each of the two earliest attempts, only memory consumption numbers from one worker were added to the execution's memory logs. The first execution attempt fails in superstep 4, after a too high memory consumption on the utilized workers. This is a consequence of very large calculated initial message limits, causing the workers to produce many messages on the last StepJoin. Worker 0, the only worker with consumption numbers for the attempt added to the execution's memory logs, has a visible increase in required memory. The second attempt fails in superstep 9. Again, the workers run out of memory, despite the calculated limits being close to one-tenth of the ones calculated in the previous attempt. The execution of the query plan finally succeeds in the third attempt. Every calculated limit in this attempt is around one-hundredth of the ones calculated in the first.

It is not implemented any continuation mechanism for pipelining with bushy query plans. Anyway, that would not help much in this case. The previously described continuation mechanism does not delete complete solutions produced in supersteps where all workers succeed. The failing attempts end before any superstep is finished with a significant amount of solutions written to the HDFS.

The duration of each execution of Query Plan 3.4 was here affected by extra requested garbage collection. The execution with the reference implementation of Lighthouse lasted for 3 hours and 32 minutes. Pipelined Lighthouse with static limits and the restart mechanism, with less available memory, finished after 14 hours and 19 minutes. The much longer duration can be explained by the earlier attempts executing slowly with a full memory, spending more time on garbage collection. The last execution attempt by pipelined Lighthouse needed 3 hours and 52 minutes to complete. To reduce the experienced execution time, it may be a better approach to start with small initial message limits and carefully increase them while tracking the memory consumption. This might prevent large amounts of garbage collection and multiple retries after workers run out of memory.

Finally, a more extreme pipelined execution is presented. Pipelined Lighthouse with static initial message limits and support for automatic restart on failure can also successfully complete an execution of Query Plan 3.4 on 5 workers with 2 GB of memory. The graph in Figure 3.19 shows the memory consumption after each superstep. Since the input graph itself roughly requires a total of 8 GB memory after being loaded onto the workers, a total of 2 GB is left for storing messages.

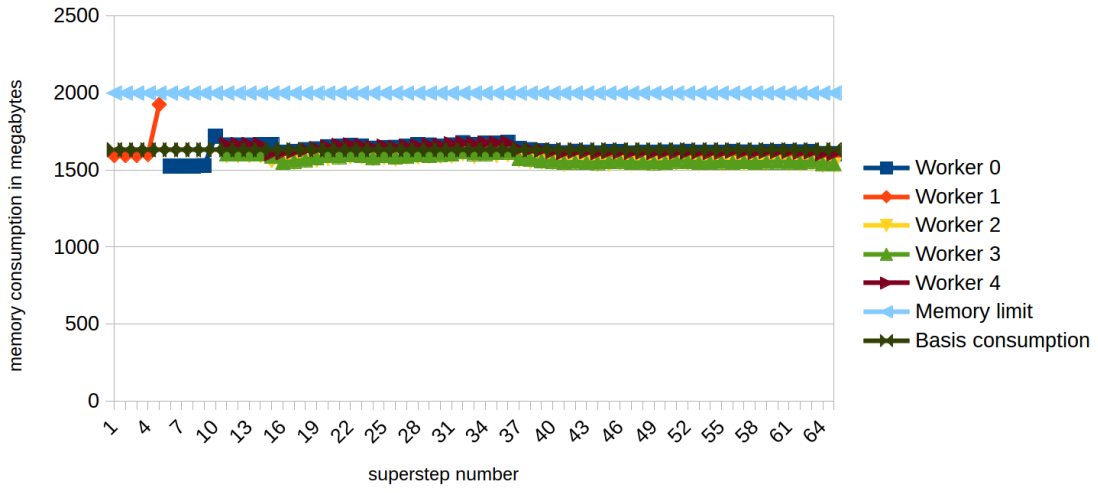


Figure 3.19: Memory consumption for pipelined execution with static limits of Query Plan 3.4

Instead of requiring 3 supersteps, 65 supersteps are needed, spread over 3 execution attempts. The skew protection fraction size is set to 0.1, preventing failures with workers calculating negative initial message limits. After the input is loaded, each worker has almost 400 MB of free memory. Only some of it can be used to store messages without extensive triggering of the garbage collector.

The duration of this execution of Query Plan 3.4 was affected by extra requested garbage collection. The last execution attempt by pipelined Lighthouse with static initial message limits and the restart mechanism, involving 5 workers, lasted for 2 hours and 19 minutes. The total time for the execution was 5 hours and 17 minutes. This is far less than the duration of the previous pipelined execution.

Chapter 4

Conclusion

As the presented measurements clearly show, pipelining can be utilized to achieve reduced and bounded memory consumption during Lighthouse executions. It enables smaller Hadoop clusters with limited memory resources to run more communication-heavy subgraph pattern matching jobs. Any left-deep query plan can now be executed without Out-of-Core Giraph disk access, provided that the input graph fits into the memory of the started workers. Any bushy query plan can also be executed without extra disk access, provided that the input graph and created PathJoin tables fit into the workers' memory. If used garbage collectors are not configured to never stop an ongoing execution, some executions may fail if only a small amount of the memory is available for storing messages and the garbage collectors spend a too large fraction of the CPU time recovering memory.

The handling of initial message limits in pipelined Lighthouse seems to work well. However, the first supersteps which are computed to gather statistics for calculations of optimal initial message limits might end with a failure. Unfortunately, a single default limit value does not fit all potential combinations of handled query plans, input graphs and utilized workers. A limit appearing small in one execution, might in another lead to workers running out of memory. The performed calculations of initial message limits give suitable values with left-deep query plans, but only when the available statistics closely reflect the whole execution. Since most executions of bushy query plans perform bad output predictions, potentially causing restarts and long execution times, PathJoin should for now be avoided if possible. Changes to PathJoin related predictions are presented as future work. Buffers are successfully used against input data skew and extensive triggering of garbage collection.

In addition to pipelining, several other implemented optimizations have shown to be advantageous for executions. Generally, the new data structures and serialization formats enable graph partitions to be stored using less memory. The presented optimizations for executions of bushy query plans also give positive effects. Memory consumption can be drastically reduced by storing the messages in join tables in a serialized form. With deletion of join tables which are not needed anymore, at the end of compute calls, memory is not unnecessarily occupied for remaining computations. If all messages from the path which passes the fewest bytes to a PathJoin are forced to arrive before any from the other path, further reduction of memory consumption is achieved. The two last mentioned optimizations require more cooperation among the used workers in pipelined Lighthouse, including agreements for when the last message on a path is passed and which paths produce the most data.

The pipelining functionality should with default configurations be utilized by Lighthouse, both for left-deep and bushy query plans, despite making the execution logic more complicated. Many extra supersteps are mainly introduced when an execution risks running out of memory. However, an increased number of computed supersteps has showed to not affect the execution time significantly. Lighthouse with pipelining may underestimate memory consumption during predictions. In worst case, this causes an execution attempt to run out of memory (which will also happen without any pipelining). Conveniently, automatic restarts ensure that pipelined executions eventually succeed. To reduce execution times with left-deep plans, reproduction of complete solutions in performed retry attempts can be avoided. The improved data structures and serialization formats, and storage of serialized messages in join tables, should be enabled by default in Lighthouse. Differently, the changes for computation start on paths need more adjustments to work properly with pipelining.

Chapter 5

Future Work

This chapter presents potential future work for improvements to Lighthouse. The proposed changes span from further development of Lighthouse's pipelining functionality to general memory optimizations and methods that reduce execution times. Each should be implemented and evaluated.

5.1 Changes to Lighthouse with Pipelining

Following is potential future work which involves changes to pipelined Lighthouse.

5.1.1 Add Broadcasting of Initial Message Limits

Currently, there is no communication based cooperation among the utilized workers for calculation and use of initial message limits. The risk of some workers having extreme initial message limits, causing the execution to fail, can be reduced by forcing workers to broadcast their calculated initial message limits. Before computations, each worker should select for use the median or smallest limit in the set of broadcasted initial message limits, reducing its chance for running out of memory.

5.1.2 Add Broadcasting of Worker States

Some overloading of workers with messages should be prevented through sharing of worker states. These states might also be used to balance out how many messages a worker can transfer to others. A broadcasted state must contain the numbers of messages passed (or to be) by a worker. Workers should consider the total numbers of messages which have been (or are to be) passed from other workers to a target worker, preventing popular targets from being drowned with messages and getting a high memory consumption. If all workers in an execution are able to track the memory consumption of each other, they can also adjust the amounts of messages they send accordingly.

5.1.3 Introduce Pipelined Global Operators

The implemented pipelining functionality for Lighthouse only restricts the numbers of initial messages which are produced in every started superstep. The pipelining functionality can thus not prevent workers from running out of memory if global operators in the middle of the query plan suddenly start to produce large amounts of messages. Work for solving this issue has been started. With some initial changes, one message buffer is used per present global operator on each worker. During a superstep, a buffer is filled with messages when the number of messages passed by the associated operator exceeds a calculated limit. These messages are handled in the next superstep. All initial message limits and operator limits are recalculated before a superstep, taking into account the current numbers of buffered messages. The changes make global operators pipelined with own effective send limits. Separate pipelined versions of Move and StepJoin should be introduced.

5.1.4 Improve Predictions for PathJoin Output

After the pipelined executions of Query Plan 3.4 (presented in the evaluation chapter), it is clear that Lighthouse's predictions for output from PathJoin operators are inaccurate. This is caused by two flaws. First, the predictions do not consider that the numbers of messages which are received by a PathJoin on different vertices may vary significantly. This gives large underestimations for the amount of output. Second, most predictions for the number of vertices that store a join table for a specific PathJoin at the end of the final superstep are far off. The simple upscaling of the numbers of vertices which store join tables, before the calculation of an initial message limit, usually results in gigantic numbers. This amplifies the underestimations for PathJoin output. In late supersteps, more messages reach a PathJoin on vertices which already store a relevant join table.

5.1.5 Enable Gradually Increasing Initial Message Limits

Instead of improving the predictions for output from PathJoin operators, a different direction for selection of initial message limit values may be taken. A small default limit value can be carefully increased, based on memory consumption observations. For bushy query plans, the extra growth of memory consumption during late computations, caused by PathJoin operators, must be considered. The risk of running out of memory might be reduced, but with a cost of more required supersteps.

5.1.6 Base Memory Predictions on Input Graph Histograms

Potentially, other methods for gathering statistics relevant to the execution can improve the memory consumption predictions. For instance, generated histograms for the input graph may be used. The histograms should contain numbers for occurrences of labels and property values. Realistically, it will be hard to characterize how each operator impacts an execution based on this information.

5.1.7 Pipelined Lighthouse versus Lighthouse with Out-of-Core Giraph

This is not a potential improvement requiring changes, but a comparison to be performed. Early on, a performance comparison between pipelined Lighthouse and Lighthouse with Out-of-Core Giraph was planned. It was supposed to involve various executions of jobs in similar memory constrained environments. Unfortunately, it has shown difficult to configure Out-of-Core Giraph. On the same set of workers, a job which completes with pipelined Lighthouse, but not with reference Lighthouse, has not yet been observed to complete with Lighthouse using Out-of-Core Giraph.

5.2 Implement Selective Loading of Input

Unfortunately, selective loading of input vertices and edges has not been added to Lighthouse during this project, despite being mentioned in the research questions. This promising optimization should soon be implemented. It is expected to significantly reduce the amount of needed memory, for storing the input graph, in many executions. As an example, if only vertices with one specific label are relevant for the production of complete solutions for a provided query, just vertices with that label and important edges should be loaded into the workers' memory before the computations.

5.3 Reduce Execution Times

The new memory optimizations' effect on execution times has not been a focus point in this project. The implementations for the optimizations may be adjusted to additionally reduce execution times, involving less overhead. For instance, improvements can reduce the time required to find sizes for messages stored in a serialized form. Such sizes, for messages received by PathJoin operators and messages passed by global operators, are often considered in memory predictions. Further, changes to reduce the amount of necessary garbage collection during executions should be explored.

Appendix A

Pregel Model

The Pregel computing model[8] was developed by Google for scalable and fault-tolerant processing of large input graphs. It was inspired by the Bulk Synchronous Parallel model[15] and enables graph processing using multiple machines and threads in parallel. The BSP model requires computation to proceed in global supersteps, each containing concurrent computation, communication and a barrier synchronization. A superstep in a Pregel job involves an iteration over input graph vertices, calling a user defined compute function once for every active vertex. The function can process messages sent to the vertex in the previous superstep, modify the vertex value or its outgoing edges and send messages which can be processed by other vertices in the next superstep. The compute function can be executed for different vertices in parallel, even on the same worker.

The vertex-centric view provided by Pregel allows implemented applications to not consider the distribution of graph vertices among workers. The forced barrier synchronization at the end of each superstep also simplifies the process of writing correct graph applications, removing the risks for deadlocks and livelocks.

A Pregel job starts with every vertex of the input graph having an active state. This causes the user defined compute function to be executed for all vertices in the first superstep. When being processed, a vertex can change its state to inactive by calling a provided vote-to-halt function. Vertices with an inactive state will not be processed in subsequent supersteps. An inactive vertex will only be reset to active if it receives a message. The execution of a Pregel job terminates when every vertex is inactive and there are no messages left to be received.

The Pregel model also introduces concepts such as combiners and aggregators. Combiners may be used to reduce communication, while aggregators are useful for global sharing of data.

Combiners
The user can in some cases utilize combiners to decrease the amount of data being passed between workers. If every worker will perform a reduction operation on values of its received messages, the senders can in advance take advantage of this knowledge and combine the messages to be sent. If every worker will perform a max() operation on received values, each sender only needs to pass its highest message value. A combiner is implemented with a function specifying how to combine two messages into a single one. Pregel does not give guarantees for which messages are combined.

Aggregators
A vertex can provide values to globally available aggregators. An aggregator performs a reduction operation on received values and provides a result to workers in the following superstep. Predefined aggregators are provided for common reduction operations, but Pregel also supports user-defined ones. Aggregators can be utilized for everything from leader election to production of statistics. One can use a sticky aggregator if the reduction operation should be performed on values from all the previous supersteps.

Appendix B

Apache Giraph

Apache Giraph[2] is an open source implementation of the Pregel framework, using the Pregel computing model. It is currently used at Facebook[4] to analyze their social graph formed by users and their relations. Following are an important class and an essential interface when using Giraph.

class WorkerContext
Data shared by vertices on a worker must be stored in a WorkerContext object. This data will not be globally shared.There is one worker context object per worker in a computation. This object can be accessed from the user defined compute method which is executed once for every active graph vertex in a superstep. Its data structures can be directly read and set. The used WorkerContext class may be implemented by the user, but must inherit from the WorkerContext class provided by Giraph. The used WorkerContext class also specifies what every worker should do before and after each superstep, in preSuperstep() and postSuperstep(), and before and after the application, in preApplication() and postApplication(). This behavior may depend on the worker data.

interface Writable
The classes used for vertices, edges and messages must implement the interface Writable. This enables the data of their corresponding objects to be serialized and de-serialized. The Writable interface requires the classes to implement readFields() taking values from a DataInput object and write() adding values to a DataOutput object. Vertices and edges are prior to a computation serialized before being distributed to workers. This reduces the amount of data which must be transferred over the network. When the workers receive their serialized parts of the input graph, they de-serialize the vertex data and store vertices as Java objects in memory. Messages are serialized before being passed to vertices. Edges and messages are de-serialized on the fly before being processed.

The Giraph framework provides additional functionality to what is described for the Pregel computing model, among others master computation, sharded aggregators and out-of-core computation.

class MasterCompute
Apache Giraph supports use of a MasterCompute class defining a master vertex which is used to perform computation between each superstep. The master vertex will be created and stored on the master, and be processed with its user defined compute method before any other worker vertex. The compute method can conveniently register and set aggregators, affecting worker vertex computations in the same superstep.

Sharded Aggregators
Sharded aggregators may be utilized to reduce the computation and communication performed by the master. In applications with normal aggregators, the master must receive, process and send data, potentially in such amounts that the master becomes a bottleneck. A sharded aggregator is differently from a normal aggregator assigned to a worker. This worker performs the aggregation and sends its result value to the master. The master will when finished send a value back to the worker, which then distributes this value to all the other workers.

Appendix C

Cypher Query Language

Cypher[5] is a query language created for use with Neo4j graph databases. It is inspired by SQL, which it shares among others keywords like WHERE, ORDER BY and UNION with. The main advantage of writing graph queries with a declarative language such as Cypher, is that one avoids to describe in detail how computations of solutions must be performed.

A Cypher query is aimed to be executed on a labeled property graph. This type of graph stores nodes and relationships with labels and properties. A label specifies a type for an entity, while a property is a pair with the property name and an entity specific value. Labels are often used to specify what properties a vertex or a relationship contains. For example, a vertex with the label 'Person' should be guaranteed to contain values for properties named 'firstName' and 'birthDate'.

In Cypher, nodes and relationships in patterns are expressed as follows:

Node: (), (id), (:LABEL), ({name: value})
: A node is expressed with a parenthesis pair. An identifier can be used for later referencing the node or its property values. Restrictions on labels for a node or stored property values can also be set as shown.

Relationship: -->, -[id]->, -[:LABEL]->, -[{name: value}]->
: A relationship between nodes is expressed with an arrow. An identifier can be used for later referencing the relationship or its property values. Restrictions on labels for a relationship or stored property values can also be set as shown.

Here are the two most important elements of Cypher read queries:

MATCH
: The MATCH clause is used to search for patterns in the stored graph.

Listing C.1: Return Stored Nodes with 'Person' Label

```
MATCH (p:Person)
RETURN p
```

Listing C.2: Return Stored Relationships with 'Friendship' Label between Specified Vertices

```
MATCH ({name:'Luke Skywalker'})-[r:Friendship]->(:Robot)
RETURN r
```

WHERE
: The WHERE keyword is often used together with MATCH to additionally filter patterns.

Listing C.3: Return 'Person' Nodes with Specified Property Values

```
MATCH (p:Person)
WHERE p.gender = 'male' AND p.country = 'Netherlands' AND p.age >= 100
RETURN p
```

Appendix D

Path Queries

The Cypher path query syntax[1] is currently not supported by Lighthouse. Though, the functionality for finding paths with variable length or shortest paths can be implemented in Lighthouse with use of left-deep query plans. The number of StepJoin operators in a used query plan determines the maximum number of relationships in the potential result paths. In path query computations, Select operators must check whether reached vertices are end-vertices. If a reached vertex is a looked for end-vertex, a result path is found and can be written to the available HDFS. If a reached vertex is not an end-vertex, the computation must continue to the next StepJoin operator of the query plan, which possibly passes new messages to related vertices.

Following is a query plan which can be used to find the shortest friend relation paths between person A and person B. The query computation starts with finding the vertex for person A via the Scan operator, then performs StepJoin passing the computation to all related friends. This involves the creation of a number of messages equal to the number of friends of person A. The vertices for the friends are then with the Select operator checked whether they represent person B. If not, the next StepJoin operator must be computed.

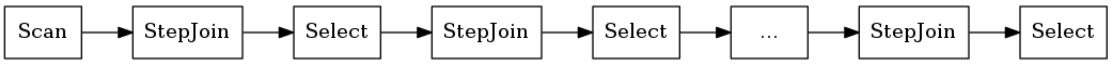


Figure D.1: Path query requiring an unknown number of supersteps

For every executed StepJoin in the query plan in Figure D.1, there may be an exponential increase of messages in the system. If too many messages are present in the system at the same time, workers may run out of available memory and cause the query plan execution to fail. To reduce the possibility for this scenario to happen, the approach for the computation must be changed. Functionality can be added to avoid messages being passed in loops, removing some unnecessary memory consumption. The implemented pipelining functionality can be used to set a limit on the number of present messages in the system. Pipelined Lighthouse should be efficient for path queries with many start vertices, but is likely to struggle with estimation of accurate memory consumption with few. This is thoroughly explained in the evaluation section of the thesis.

Some extensive work has recently been done on shortest path queries in Lighthouse[14]. It considers how path queries can be represented as query plans and introduces a new shortest path operator.

[1] http://neo4j.com/docs/stable/introduction-pattern.html#_variable_length

Bibliography

[1] Renzo Angles et al. "The linked data benchmark council: A graph and RDF industry benchmarking effort". In: *ACM SIGMOD Record* 43.1 (2014), pp. 27–31.

[2] *Apache Giraph*. URL: http://giraph.apache.org.

[3] Avery Ching. *Scaling Apache Giraph to a trillion edges*. URL: https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920/.

[4] Avery Ching et al. "One Trillion Edges: Graph Processing at Facebook-Scale". In: VLDB '15 (2015).

[5] *Cypher Query Language*. URL: http://neo4j.com/developer/cypher/.

[6] Sinziana Maria Filips. "A scalable graph pattern matching engine on top of Apache Giraph." MA thesis. VU University Amsterdam, 2014.

[7] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008.

[8] Grzegorz Malewicz et al. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146.

[9] Claudio Martella, Dionysios Logothetis, and Georgos Siganos. "Spinner: Scalable Graph Partitioning for the Cloud". In: *CoRR* abs/1404.3861 (2014). URL: http://arxiv.org/abs/1404.3861.

[10] *Neo4j*. URL: https://neo4j.com/.

[11] Wing Lung Ngai. *Fine-grained Performance Evaluation of Large-scale Graph Processing Systems*. TU Delft, 2015.

[12] *Out-of-Core Giraph*. URL: http://giraph.apache.org/ooc.html.

[13] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013. ISBN: 1449356265, 9781449356262.

[14] Peter Rutgers. "Extending the Lighthouse graph engine for shortest path queries." MA thesis. VU University Amsterdam, 2015.

[15] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181. URL: http://doi.acm.org/10.1145/79173.79181.

[16] Annita N. Wilschut and Peter M. G. Apers. "Dataflow Query Execution in a Parallel Main-memory Environment". In: *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. PDIS '91. Miami, Florida, USA: IEEE Computer Society Press, 1991, pp. 68–77. ISBN: 0818622954. URL: http://dl.acm.org/citation.cfm?id=382009.383658.