Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

Scalable motif search in graphs using distributed computing

Andrew Esler

A thesis presented in partial fulfilment of the requirements for the degree of a Masters in Computer Science

Massey University

Turitea

New Zealand

2012

Acknowledgments

I would like to thank my supervisor Jens Dietrich for the funding and motivation for this project. Patrick Rynhart provided extensive help with Massey's computing infrastructure. Thanks also to those who were sounding boards for ideas and pointed out my follies.

Abstract

Motif detection allows software engineers to detect antipatterns in software. By decreasing the number of antipattern instances in a piece of software, the overall quality of the software is improved. Current methods to find these antipatterns are slow and return results only when all antipatterns have been found. The GUERY framework is able to perform motif detection using multiple cores and deliver results as they are generated. By scaling GUERY to run on multiple machines, it was hoped that research requiring many queries on a graph could be performed significantly faster than is currently possible.

The objective of this thesis was to research and prototype mechanisms whereby GUERY could be run using a cluster of computers and results delivered as a stream to interested systems. A system capable of running on a cluster of machines and delivering a stream of results as they are computed was developed.

Contents

List of Figures vii

List of Tables ix

List of Listings xi

1 Introduction 1
- 1.1 Introduction . . . 1
- 1.2 Project Objective and Scope . . . 5
  - 1.2.1 Validation . . . 5
  - 1.2.2 Changes in the field . . . 6
- 1.3 Spikes . . . 6
- 1.4 Overview of thesis . . . 7

2 Background 9
- 2.1 Graphs and Motifs . . . 9
- 2.2 Graph Query Languages . . . 14
- 2.3 GUERY . . . 14
  - 2.3.1 GUERY Queries . . . 17
- 2.4 Graph Partitioning . . . 17
  - 2.4.1 Implementation . . . 19
- 2.5 Cloud Computing . . . 20
- 2.6 Buffering . . . 21

3 Requirements and Validation Metrics 25
- 3.1 Requirements . . . 25
  - 3.1.1 Scalability . . . 27
  - 3.1.2 Result Streaming . . . 27

3.1.3 Outside Project Scope 28
3.2 Validation Metrics 29

4 Existing Platforms and Frameworks 31
4.1 BSP 32
4.2 Pregel 33
4.3 Hadoop 34
4.3.1 MapReduce 35
4.3.2 Apache Hama 37
4.3.3 Giraph 37
4.3.4 GoldenOrb 38
4.4 Graph Databases 38
4.4.1 Neo4j 38
4.4.2 InfiniteGraph 38
4.5 Terracotta 39
4.6 HipG 39
4.7 Other Approaches 39
4.7.1 Custom GUERY clustering system 39
4.8 Summary 40
4.8.1 Comparison Table 40
4.8.2 Why Hadoop? 41

5 Hadoop GUERY Runner 45
5.1 Design 45
5.1.1 Input to HGR 46
5.1.2 Result Streaming 47
5.2 Constraints and Limitations 50
5.3 Local vs Hosted infrastructure 50
5.3.1 Hardware 52
5.3.2 Amazon Simple Storage Service 53
5.3.3 Problems Encountered when moving to Amazon 53
5.4 Experimental Evaluation 54
5.5 Requirements Validation 58
5.5.1 Set up complexity 59
5.5.2 Customization required 59
5.5.3 Find first 59
5.5.4 Find first 10 60
5.5.5 Find all 60

5.5.6 Synchronization overhead 60
5.5.7 Memory usage 60
5.5.8 Worker idle time 60
5.6 Weaknesses 61
5.6.1 Result Streaming Fault Tolerance 61
5.6.2 Amazon Infrastructure 62
5.7 Conclusions 62

6 Distributed Graph Processor 67
6.1 Why a messaging based system? 68
6.2 Messaging System Selection 69
6.3 RabbitMQ 71
6.3.1 Availability 73
6.4 Architecture 73
6.5 Message Queues 74
6.6 Components 78
6.6.1 Messaging 78
6.6.2 Coordinator 79
6.6.3 Worker 80
6.6.4 Partitioner 80
6.6.5 ResultProcessor 82
6.6.6 LiveLogViewer (LLV) 82
6.7 Scheduling and Task Management 83
6.8 Graph Partitioning and Partial Solutions 85
6.9 Job Execution Workflow 86
6.10 Scalability 87
6.11 Weaknesses 87
6.12 Fault Tolerance in Streaming Systems 88
6.13 Current State of DGPROC 89
6.14 Conclusions 91

7 Conclusions 93
7.1 Contributions 93
7.2 Future Work 94

Bibliography 95
.1 Appendix GUERY Query Grammar 97
.2 Appendix HGR 104

.3 Appendix DGPROC 105

List of Figures

1.1 *The definition of the subtype knowledge anti-pattern.* 3
1.2 *An instance of the subtype knowledge anti-pattern found in JRE 1.7.* 4

2.1 *A graph with three distinct vertex roles and three distinct edge roles.* 11
2.2 *An example motif with three roles.* 11
2.3 *An example motif with two roles and an edge constraint.* 12
2.4 *Cloud service layers.* 22

4.1 *A diagram showing the layers of abstraction with respect to GUERY.* 32

5.1 *An overview of the HGR system showing how data flows.* 46
5.2 *The path a result takes once it is computed.* 49
5.3 *A graph of job execution time per instance type vs worker count for all three instance types.* 55
5.4 *A graph of job execution time per instance type vs worker count for m1.large and c1.medium instance types* 56

6.1 *The basic architecture of RabbitMQ.* 72
6.2 *The interaction between DGPROC components.* 75
6.3 *A diagram of the result subscriber notification process.* 76
6.4 *A diagram of how partitions are created and used during job processing.* 81

List of Tables

4.1 A comparison of systems 41

Listings

2.1 ResultListener Interface 16
2.2 Subtype knowledge 17
2.3 Circular Dependency 17
2.4 Abstraction without Decoupling 18
4.1 Word Count Example map function 36
4.2 Word Count Example reduce function 36
5.1 Example of a vertex serialised in JSON 48
5.2 MotifTransformer interface 48
5.3 MotifProcessingWorker interface 49
5.4 A possible serialisation abstraction interface 64
6.1 Multithreaded GQL engine instantiation 67
6.2 DGPROC GQL engine instantiation 67
1 The ANTLR grammar file for GUERY queries. 97

Chapter 1

Introduction

> If you aren't deeply frightened
> about all the additional issues
> raised by concurrency, you
> aren't thinking about it hard
> enough.
>
> John Carmack

1.1 Introduction

A network is defined as a a set of objects called vertices, where vertices may be linked by edges. A network may be small with only a few tens of vertices, or may have billions of edges and vertices.

The ability to rapidly discover motifs [67] in networks or graphs is important in a number of fields, including software engineering and bio-informatics. Motif detection in bio-informatics is useful for examining patterns in DNA and finding proteins that behave similarly. A motif is a set of vertices and edges bound to particular roles, which can also be viewed as a subgraph that matches a defined structure.

The interest of this study in distributed motif discovery is applying it to the analysis of software anti-patterns. An anti-pattern is an undesirable pattern in code. A graph of the relationships between classes in a software package can be created using source code or static analysis of compiled executables. By defining a motif for particular anti-pattern, it is then possible to look for all instances of the antipattern in the graph of the software

package. The resulting motif instances can then be used to look at refactorings that may improve the quality of the software by decreasing coupling[82] and increasing cohesion[82].

When a motif is viewed as a subgraph, the motif discovery becomes a sub graph isomorphism problem. That is, can all sub-graphs in a graph that are isomorphic to a particular motif be found? The sub-graph isomorphism problem has been proven to be NP-complete[45], using a reduction to 3-SAT. Nondeterministic Polynomial (NP) complete problems are a class of decision problems where solutions can be verified in polynomial time, but there is no known efficient way to find a solution. An increase in the size of the problem rapidly increases the size of the solution space and thus computation can be considered unfeasible, as it may take hundreds or thousands of years, even on modern supercomputers. The most common way of finding solutions to these problems is to use approximation algorithms, which by definition are not guaranteed to find an optimal answer.

There are a number of existing systems that can perform motif discovery, either specifically for bio-informatics[37, 54], or in more general cases[26, 71, 79, 41, 48]. However, none of these can be run in a distributed system, nor are they able to stream results as they are computed.

The subtype knowledge antipattern[74] occurs when a class B (subtype) inherits from class A (supertype), and the super type is connected to the subtype indirectly. Figure 1.1 shows a diagram of the subtype knowledge antipattern. This diagram may appear simple, but finding instances of this anti-pattern is difficult. Figure 1.2 shows an instance of the subtype knowledge anti-pattern. The different coloured labels correspond to different Java packages. In this example, the super type BlockingQueue knows about its subtype SynchronousQueue via a chain of classes that involves 14 classes in seven different packages. This instance would be almost impossible to find manually due to the number of classes involved and their distribution throughout the system. This shows the importance of automated tools to help developers improve code quality by detecting these anti-patterns.

GUERY[48, 18] is a tool for defining motif queries and executing these queries against graphs. The example in figure 1.2 was found in JRE 1.7[1] using GUERY. The authors of [49] look at the impact refactorings can have on the quality of code. They are interested in developing automated refactorings

[1]JRE stands for Java Runtime Environment, and is software that is required to run Java programs.

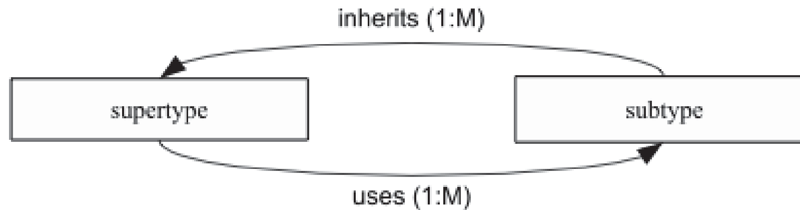

Figure 1.1: *The definition of the subtype knowledge anti-pattern.*

based on experimental testing against the Qualitas Corpus[84] of programs.

In order to find edges that participate in many antipatterns, a graph of the software is queried for antipattern instances. Each edge has a count of how many antipatterns it is involved in. Edges with high counts are ideal candidates for refactoring as removing one edge can remove many antipattern instances. This is similar to how page rank[70] and community detection[53] work; looking for edges that are used in many paths. When edges are found, the program iteratively removes them one by one, re-querying the changed graph after each edge is removed to see how the number of antipattern instances is altered. Doing this for each program in the Qualitas Corpus[30] and hundreds of iterations of edge removal takes hours. If the analysis program can be made to scale near linearly across a cluster of machines, the running time of the antipattern analysis can be greatly reduced allowing analysts to rapidly see the impact of changes.

The size of the graphs and the complexity of the problem means that motif detection is a hard problem. One way to make motif detection more usable for those without access to a supercomputer is to design the solver to be flexible enough to achieve linear scaling. In this case, if the software was running on a cluster of ten machines, then it should work nearly ten times as fast as if it was running on a single machine. It is almost impossible to achieve perfect linear scaling due to the overhead of communication required between the multiple machines. A major barrier to making use of the scaling factor is that it requires the user to have a large number of machines available that they can configure to run the software. There is significant expense in doing this, as there is the initial cost of purchase, the ongoing maintenance required, as well as power and network requirements and the

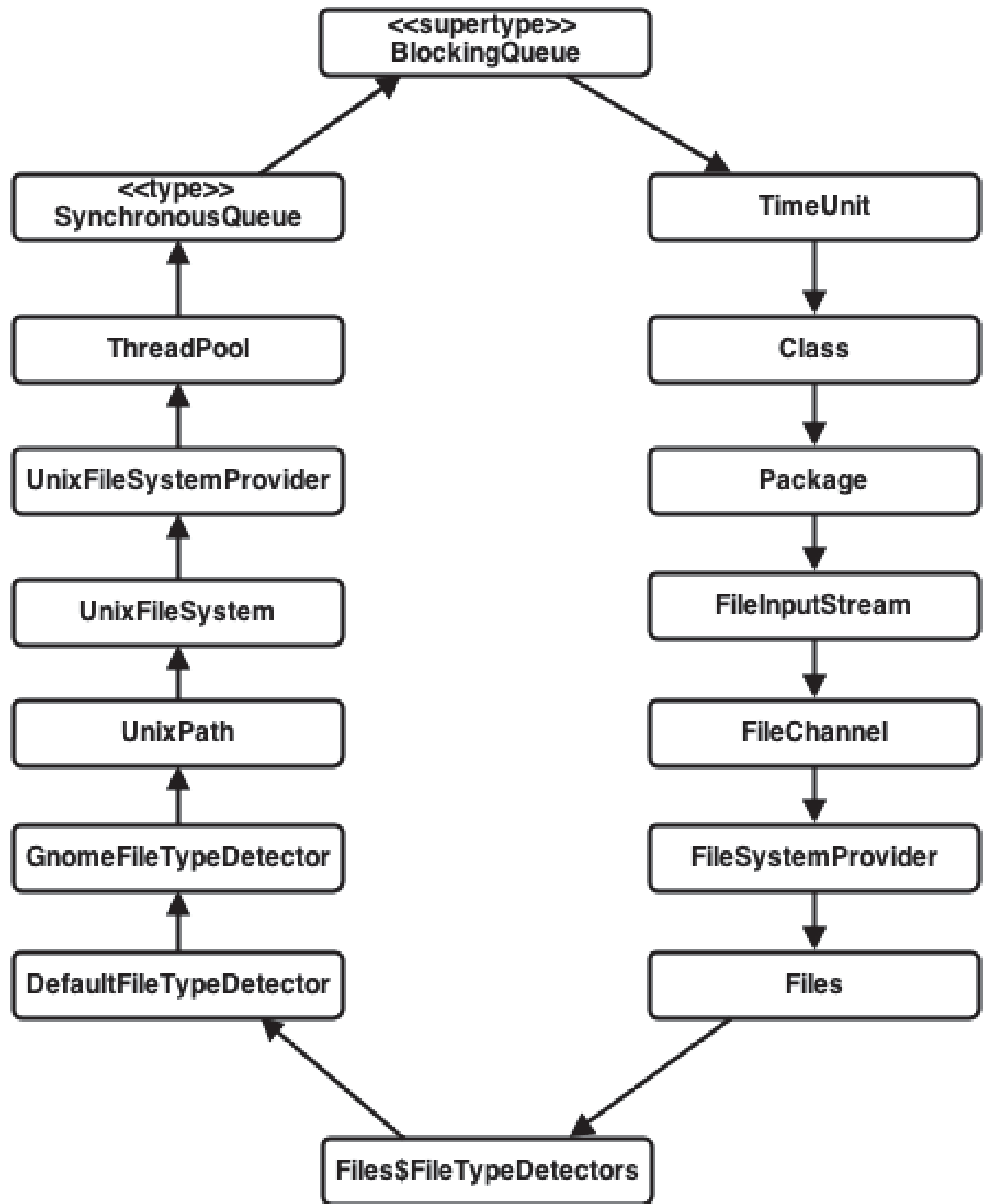

Figure 1.2: *An instance of the subtype knowledge anti-pattern found in JRE 1.7.*

physical space to put them. Cloud computing is a method that provides a solution to the ownership cost. A user may rent computing power, usually by the hour, from a cloud computing provider. There is usually a range of hardware configurations available, with the cost per hour rising as the hardware specification increases. The user has root access to the rented machines allowing them to install any software they require on them. By utilising on-demand computing power, the user of the software lowers the cost and still has the option of running jobs with a large cluster of machines.

1.2 Project Objective and Scope

The objective of this project was to engineer a system for scalable motif detection that also supports results streaming as they are generated. The question to be answered is, how close to linearly can a distributed system for motif detection scale? This involved evaluating existing distributed computation frameworks for their suitability of being adapted to support a distributed GUERY, as well as researching the feasibility of developing a system from scratch. The initial systems identified as likely candidates for adaption were Hadoop[19], Neo4j[27], and Terracotta[34].

Due to the complexity of the project and time constraints, some aspects, such as fault tolerance, were not given the attention they perhaps deserved. For the complete set of requirements see Chapter 3.

1.2.1 Validation

To validate the systems developed, there are eleven metrics defined in section 3.2. Firstly, is the distributed system both correct and complete? Correct means that there are no false positive results and complete means that every motif instance that should be found, is found. This can be tested by comparing the results of GUERY running on a single instance to the results of the distributed GUERY run. Note that only relative correctness and completeness with respect to GUERY and its test cases can be established.

Secondly, how close to linearly does the system scale? Amdahl's law[36, 75] describes the maximum speedup an algorithm can have when split across many processors. The speedup is limited by the proportion of the problem that is parallelisable. If a large part of the code must be run sequentially, then only a small increase in speed could be expected by parallelizing the code.

Additionally, factors such as coordination and synchronization overheads further reduce the speedup that can be obtained. As such, the system is not going to be n times faster when run on n machines, but will be significantly faster than a single GUERY instance.

To validate the scaling properties of new systems, the metric used is the change in running time. By comparing the overall running time of a new distributed system to GUERY, where the distributed system has access to many machines of a specific configuration, and GUERY has access to a single machine of the same configuration, it can discovered how well the new system scales.

1.2.2 Changes in the field

There is a lot of work occurring in the field of distributing graph problems. As such, much of the work in this thesis will be outdated or invalidated by the time it is read. This thesis should not be treated as an explanation of how to tackle distributed motif analysis with current tools, but rather paths that were explored before many of the new tools became available. For example, the Surfer [43] paper published by Microsoft research has overlapping goals to this thesis, but the developed system is not available.

1.3 Spikes

A spike[80] is a concept from agile project management, specifically SCRUM. It is a time boxed experiment, intended to investigate a new technology or to see if an idea is technically feasible. Code written during a spike is not intended to be production quality. It is intended to provide new knowledge, which may then help developers create a production ready artifact.

A number of spikes were used to build prototypes, some of which were eventually developed into the full systems described in later chapters (5 and 6). Many of the spikes were never further developed, as they either proved the technology under investigation was not right for the task, or was not mature enough to achieve the objectives within the timeframe of this project.

Some technologies that were investigated but are given little mention in this thesis are the graph databases Neo4j and InfiniteGraph, and Terracotta. Graph databases proved promising, but time constraints prevented a complete

system being developed around one. Terracotta was discarded as not being a good fit with the projects objective.

There was a brief investigation into the feasibility of using graphics cards for processing, given their massively parallel nature. Further investigation was halted when more promising options such as Hadoop and graph databases were identified. It was found that there are existing distributed computation frameworks such as MARS[55] that can run on a graphics processing unit (GPU).

List of Spikes

Terracotta Seven days were spent investigating Terracotta[34] to see if it could be used in scaling GUERY. Terracotta did not seem appropriate for the envisioned usage.

GPU Acceleration Seven days were spent investigating whether GUERY could be made to run using graphics processing units (GPUs). Using GPUs to run GUERY was not feasible for a number of reasons.

Neo4j Investigating if Neo4j[27] could be used to provide a graph and motif storage backend to GUERY took ten days. The conclusion was Neo4j was potentially of use, but it would be necessary to modify how queries are run using Gremlin[17].

InfiniteGraph Neo4j would likely be a better choice if a graph database was going to be used was the conclusion after seven days of research in to InfiniteGraph[20].

1.4 Overview of thesis

The thesis is organised in the following manner. Chapter two covers background knowledge of graph theory, motifs, graph partitioning, and GUERY that will be relevant in understanding the remainder of the thesis.

Chapter three describes the requirements of the system, and the metrics that can be used to compare systems. Existing solutions that could be used or adapted are discussed in chapter four.

HGR, a system based on Hadoop, is covered in chapter five, along with a discussion relating the requirements and metrics to HGR are outlined in chapter three.

Chapter six explores the feasibility of an alternative design of a system based around message passing.

Chapter seven discusses the findings and contributions of the project, and explains what further work is required.

Chapter 2

Background

The concepts discussed in this section are relevant in later parts of the thesis, and an understanding of the concepts will help with understanding some of the decisions made in later chapters.

2.1 Graphs and Motifs

This section provides formal definitions for the following: path, motif, motif graph, binding, valid binding, motif instance, partial motif.

An undirected graph can be defined as $G = (V, E)$, where V is a set of vertices and E is a set of edges where each member of E connects two members of V. This is known as a undirected graph. A directed graph is defined as $D = (V, A)$ where V is the a set of vertices, but A is a set of ordered pairs of vertices called directed edges. Directed and undirected graphs differ in that edges have a specific direction and can only be traversed from source to target vertices, and not in the opposite direction.

In this thesis, only directed graphs will be analysed because the primary use case is modeling class relationships in software. It is common that a class A will reference a class B, but class B will have no references to class A. Unidirectional links between classes is desirable in software.

The following definitions and associated explanations are taken verbatim from [48], with the exception of the partial motif definition.

Definition 1 (Path) *Let $G = (V, E)$ be a directed graph consisting of a set of vertices V and a set of directed edges E (we use the term edge to denote a directed edge throughout this work.) A path is a finite sequence of edges*

$(e_1, .., e_n)$ such that for adjacent edges e_i and e_{i+1} the target vertex of e_i is the source vertex of e_{i+1}. $SEQ(E)$ is the set of all paths that can be constructed from given set of edges E. For a given path p, $start(p)$ denotes the first vertex (source vertex of e_1), $end(p)$ the last vertex (target vertex of e_n) and $length(p)$ the number of edges in the path.

For the puropses of this thesis, only regular paths are of interest. A regular path is where any edge may occur once only in the path.

Definition 2 (Motif) *Let $G = (V, E)$ be a directed graph consisting a set of vertices V and a set of directed edges E. A motif over G is a structure $M = (VR, PR, s, t, C_V, C_P)$ consisting of:*

1. *A set of vertex roles VR.*
2. *A set of path roles PR.*
3. *Two functions $s : PR \rightarrow VR$ and $t : PR \rightarrow VR$, associating path roles with vertex roles.*
4. *A set of vertex constraints C_V, consisting of n-ary predicates between vertex tuples, $c_V \subseteq \times_{i=1..n} V$, where n is the cardinality of VR.*
5. *A set of path constraints C_P, consisting of n-ary predicates between tuples of paths, $c_P \subseteq \times_{i=1..n} SEQ(E)$, where n is the cardinality of PR.*

Vertex and path assignments are restricted by constraints. Vertex constraints are defined with respect to vertex labels, while paths may be restricted by length and the labels of edges on the path.

To make the concept of motifs clearer, consider the directed graph shown in Figure 2.1. There are three vertex roles shown: square, circle, and triangle. The graph also shows three types of edges: red, green, and blue. The letters A - H are node labels so that each vertex in the graph can be clearly referred to.

Figure 2.2 defines a motif using all three vertex roles, which will be refered to as the ternary motif. The vertex role constraints are that there must be three vertices, one of each role. The path constraints are more complex. The black edge colouring is used to indicate that there are no edge label restrictions, that the edges connecting the vertex roles may be of any colour. The arrows on the edges indicate that there must be an edge of a specific

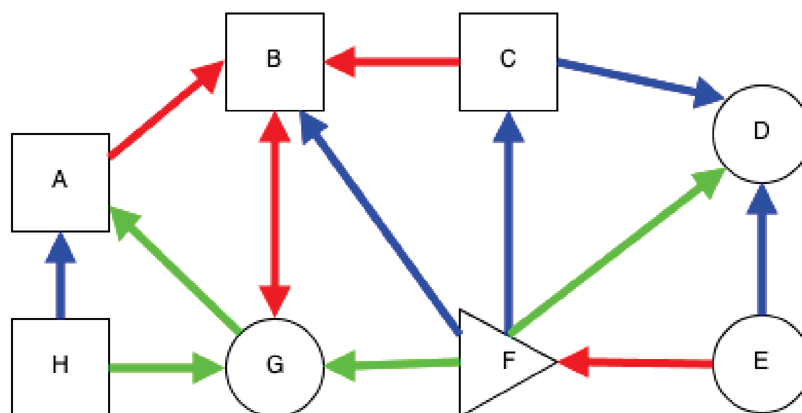

Figure 2.1: *A graph with three distinct vertex roles and three distinct edge roles.*

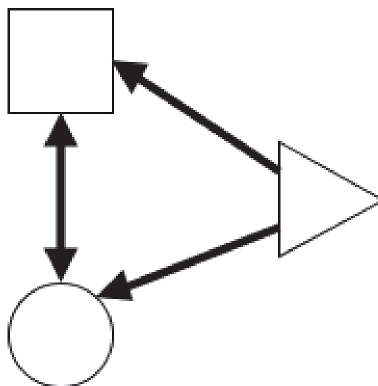

Figure 2.2: *An example motif with three roles.*

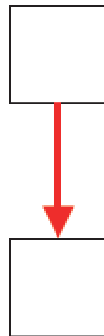

Figure 2.3: *An example motif with two roles and an edge constraint.*

direction connecting the sources and target vertex roles. In summary, figure 2.2 defines a motif where a triangle has outgoing connections to both a circle and a square, and the same circle and square have a bidirectional edge joining them.

There is a single instance of the ternary motif in the graph in 2.1. Vertices F, B, and G form an instance as they form a subgraph matching the motif. The subgraph formed by F, C, and D is not an instance of the ternary motif as C and D do not have a bidirectional edge connecting them.

Figure 2.3 defines a motif using a single vertex role, the square, and a single edge constraint, a red edge. This will be refered to as the binary motif. Instances of the binary motif are formed when a square is connected to another square via a red edge. Looking at the graph, there are two subgraphs matching these constraints: A $\rightarrow$ B and $C \rightarrow B$. The subgraphs $B \leftrightarrow G$ and $E \rightarrow F$ are not binary motif instances as in neither case are both vertices squares. $H \rightarrow A$ is not a binary motif instance because the edge connecting H to A is blue not red.

A third example of a motif could be a path composed of two or more green arrows, with any vertex type. For this motif, there is a single motif instance: $F \rightarrow G \rightarrow A$.

Definition 3 (Motif graph and connected motifs) *Let $M = (VR, PR, s, t, C_V, C_P)$ be a motif over a directed graph. The motif graph $G_M = (V_M, E_M)$ is then defined as follows: $V_M := VR$, $E_M := PR$, for each $pr \in PR$, $start(pr) := s(pr)$ and $end(pr) := t(pr)$. M is called connected iff G_M is*

weakly connected.

A binding associates vertex roles with vertices and path roles with sequences of edges. The s and t functions associate path and vertex roles: path roles connect source and target vertex roles. The conditions define a structural homomorphism between the motif and the target graph.

Definition 4 (Binding) *Let $G = (V, E)$ be a directed graph and $M = (VR, PR, s, t, C_V, C_P)$ a motif. A binding is a pair of functions $\langle inst_V, inst_P \rangle$, where $inst_V : VR \to V$ and $inst_P : PR \to SEQ(E)$, such that $inst_V(s(pr)) = start(inst_P(pr))$ and $inst_V(t(pr)) = end(inst_P(pr))$.*

A binding that satisfies the constraints given in the motif definition is called a valid binding.

Definition 5 (Valid binding) *Let $G = (V, E)$ be a directed graph, $M = (VR, PR, s, t, C_V, C_P)$ and $bind = \langle inst_V, inst_P \rangle$ A binding is called valid iff the following two conditions are fulfilled*

- $(inst_V(vr_1), .., inst_V(vr_n))) \in c_V$ *for all vertex constraints* $c_V \in C_V$
- $(inst_P(pr_1), .., inst_P(pr_n))) \in c_P$ *for all path constraints* $c_P \in C_P$

In the worst case, using regular paths, there are up to $\sum_{i=0,..,|E|} \frac{|E|!}{(|E|-i)!}$ possible paths to consider. If the motif contains n path roles, then there are up to $(\sum_{i=0,..,|E|} \frac{|E|!}{(|E|-i)!})^n$ different assignments of paths to path roles.

Definition 6 (Motif Instance) *Let $G = (V, E)$ be a directed graph, $M = (VR, PR, s, t, C_V, C_P)$ a motif and $B = \{\langle inst_V^i, inst_P^i \rangle\}$ the set of all valid bindings. Then the relationship $\sim \subseteq B \times B$ is defined as follows:*
$\langle inst_V^1, inst_P^1 \rangle \sim \langle inst_V^2, inst_P^2 \rangle$ iff $inst_V^1 = inst_V^2$.
$\sim$ is obviously an equivalence relation. We define a motif instance to be a class of valid bindings modulo $\sim$.

Definition 7 (Partial Motif) *A partial motif is a motif where not all of vertex roles VR and path roles PR are bound to vertices or paths, respectively.*

2.2 Graph Query Languages

Graphs are a good model for data from many domains where the interest is on small sets of attributes on related entities. Graphs allow the explicit modelling of the relationships between entities. For example, the social networking website Facebook[1] uses graphs of relationships between users to suggest other people a user may know but not have "friended" on the site, based on finding others who have friends in common with the user.

In order to make use of the data contained in a graph, there needs to be a way of performing queries against the graph. There are many existing graph query languages, such as SPARQL[72], Gremlin[17], and Crocopat[40]. Not all graph query languages are compatible with all graph databases. SPARQL and Gremlin are both compatible with many different databases, whereas the queries used in Crocopat are only understood by the Crocopat tool. Gremlin is also more general purpose, whereas SPARQL only works with data stored in the Resource Description Framework[2] (RDF) format.

The languages also differ in their expressiveness. For example, Gremlin can represent irregular paths through a graph, whereas SPARQL cannot.

2.3 GUERY

GUERY[18, 48] is a Java library that can be used to query graphs for instances of motifs (a.k.a patterns). GUERY supports motifs that are complex in the following sense: motifs describe sets of vertices connected by paths. Both the vertices and edges have to satisfy constraints with respect to vertex and edge labels, as well as the length of the paths. This definition of motif is significantly more expressive than the concept of motif generally used in bioinformatics where the vertices in the motif are only connected by edges.

A feature of GUERY that is uncommon among its peers is that GUERY supports the transitive closure of relationships. Motif definitions can also contain aggregation clauses, allowing queries to return aggregated patterns rather than every instance of a pattern, of which there may be many similar results.

GUERY uses an observer based Application Programming Interface (API), whereby the solver produces a stream of results, similar to a XML sequential

[1]http://www.facebook.com
[2]http://www.w3.org/RDF/

access (SAX) parser. By default, GUERY operates on in-memory graphs using the JUNG API. However, GUERY has its own graph abstraction layer, so it is possible to consume other graphs that can be mapped to the GUERY graph interface. One way that GUERY could process graphs that are too large to fit in a single machine's memory would be to use a disk-backed graph format. Instead of keeping the entire graph in memory, portions of the overall graph are kept in an in-memory cache, and vertices are loaded into the cache from disk, as required. This may slow the overall processing speed of queries due to the amounts of disk input / output (IO) required, but could be useful as only a single machine would be needed to work on large graphs.

There is also the possibility of integrating with a graph database using the GUERY graph abstraction layer. Graphs databases are explained in section 4.4 of the Existing Platforms and Frameworks chapter.

The solver component of GUERY uses a path finder to traverse the graph. There are currently two path finder implementations available: a breadth first path finder and a path finder that caches reachable nodes using chain compression applied to the condensation of the graph computed using Tarjan's algorithm[83]. The transitive closure can be pre-computed using chain compression[58].

GUERY reads graph input from GraphML[16] files. GraphML is an XML format used to describe graphs. It has a schema for data to be attached to vertices and edges in the graph. As such, it can represent arbitrary graphs in any domain. Graphs in the software engineering domain, which is the main focus of GUERY so far, generally have several different types of vertices and edges, each with a different set of properties. These different vertices and edges are represented by specific Java types. The fast processing due to the exact match of the representation to the domain is good, but it is problematic extending GUERY to deal with other domains in that each graph needs specific implementations of GUERY vertex and edge classes to represent graphs in the particular domain. This may be a barrier for use with non-developers in other domains.

The use of GraphML allows a generic graph loader could be written to load a graph in any domain represented in GraphML. The main barrier to implementation is that GUERY uses Java types to represent graphs and edges. Having a specific type for each type of vertex allows GUERY to work directly on objects representing the graph data. However, this means for each type of graph a set of domain classes must be created and supplied to GUERY to be used when the graph is loaded from XML. It is possible

to make a generic vertex type for GUERY to use but this has not been implemented in GUERY.

```
public interface ResultListener<V,E> {
    /**
     * Notify listener that a new result has been found.
     * @param instance
     * @return a boolean indicating whether to look
     *     for more results.
     */
    boolean found(MotifInstance<V,E> instance);
    /**
     * Notify listener about progress made.
     * @param progress the number of tasks performed
     * @param total the total number of steps that have
     *      to be performed
     */
    void progressMade(int progress,int total);
    /**
     * Notify listener that there will be no more results.
     */
    void done();
}
```

Listing 2.1: ResultListener Interface

GUERY uses the ResultListener interface shown in 2.1 to provide callback functions for classes interested in GUERY results. Classes register their interest in GUERY results by passing a ResultListener implemenation to GUERY. Each time a result is found, the found method will be called on each registered listener. What the implementation does with the `MotifInstance<V,E>` object is up to the specific implementation.

GUERY is able to run multiple solver threads in parallel and thus the solving time for a query decreases as the number of threads run in parallel increases. The existing parallel mode proves that the underlying algorithm used by GUERY is able to take advantage of the level of concurrency available in modern servers, so it becomes a matter of distributing the algorithm over separate machines.

GUERY was chosen as the motif discovery engine because of its ability to be parallelized and its support for complex querying. See [48] for a comparison of GUERY and alternate systems.

The correctness and completeness of GUERY has not been formally verified due to the difficulty of doing so.

2.3.1 GUERY Queries

GUERY has a very expressive query language. The full ANTLR grammar file of the GUERY query language is included in Appendix 1. Figures 2.2, 2.3 and 2.4 show examples of GUERY queries for three common antipatterns.

```
1 motif stk
2 select type,supertype
3 connected by inherits(type>supertype) and uses(supertype>type)
4 where "inherits.type=='extends' || inherits.type=='implements'"
5 and "uses.type=='uses'"
6 group by "supertype"
```

Listing 2.2: Subtype knowledge

```
1 motif cd
2 select inside1,inside2,outside1,outside2
3 where "inside1.namespace==inside2.namespace"
4 and "inside1.namespace!=outside1.namespace"
5 and "inside1.namespace!=outside2.namespace"
6 connected by outgoing(inside1>outside1)[1,1]
7 and incoming(outside2>inside2)[1,1]
8 and path(outside1>outside2)[0,*]
9 group by "inside1.namespace"
```

Listing 2.3: Circular Dependency

2.4 Graph Partitioning

Graph partitioning is the problem of dividing the vertices of a network into a number of non-overlapping groups of similar sizes such that the number of edges between the groups is minimised[69].

Using the definition of a graph given earlier, $G = (V, E)$, the partitioning of a graph can defined like so: G_i denotes a subgraph of G and V_i is the set

```
1 motif awd
2 select client, service, service_impl
3 where "!client.abstract" and "service.abstract"
4 and "!service_impl.abstract"
5 connected by inherits(service_impl>service)
6 and service_invocation(client>service)[1,1]
7 and implementation_dependency(client>service_impl)
8 where "inherits.type=='extends' || inherits.type=='implements'"
9 and "service_invocation.type=='uses'"
10 and "implementation_dependency.type=='uses'"
11 group by "client" and "service"
```

Listing 2.4: Abstraction without Decoupling

of vertices in G_i. A partitioning of a graph G can be defined as $P(G) = G1, G2; : .., Gk$, where $\forall i \in [1...k]$, $k \leq |V|$, $\bigcup_{i=1}^{k} V_i = V$, $V_i \bigcap V_j = \emptyset$, where $i \neq j$.

Edges can be classified into two types; a cross-partition edge crosses a partition boundary by having the source and destination vertices in two different partitions, and edges that have both source and destination vertices in the same partition.

Generally graph partitioning involves partitioning a graph into k partitions of approximately equal size. A good partitioning of a graph minimises the cross-boundary partition edges. Graph partitioning is relevant to this thesis as it is a common way of splitting large graph problems into chunks that are small enough too be run on a single machine in a cluster. This way, the cluster does not have to have the capacity to process the entire graph at once, but can instead process it in manageable chunks. This means the cluster can process graphs that would exceed the capacity of any single machine in the cluster. The downside to partitioning is that unless the graph can be perfectly split into k partitions with no cross-partition edges, communication is required between nodes to share partition data.

There are two options when traversing edges that cross partition boundaries, send the target vertex data to the partition containing the source vertex, or send the path traversed so far to the partition containing the target vertex and continue traversal from there. Both approaches have their pros and cons.

Graph partitioning is a well studied problem in combinatorial optimisation. As such, there are a large number of algorithms available. The Kernighan-Lin[60]

algorithm works by trying to partition a graph into two disjoint subsets such that the sum of the weights of the edges between nodes in the two partitions is minimised.

Sophisticated partitioning algorithms are available, such as mini-max grid partitioning[62] which accounts for variation in processing power of the nodes that will process each partition.

The simplest method is unweighted graph bisection, which involves dividing a graph into two sets of equal size such that the number of edges connecting the two sets is minimised. However the quality of partitioning can be heavily influenced by the initial partition split.

There is another class of partitioning algorithms that use multiple levels of computation. Each stage reduces the size of the graph by collapsing vertices and edges, partitions the smaller graph, then maps back and refines this partition of the original graph[56]. See [59, 77, 39, 81] for more detailed explanations of various multi-level partitioning algorithms.

Creating good partitions can be viewed as similar to community detection in graphs. Hence, community detection algorithms such as the Louvain method[46] can be used.

Looking at the access patterns for data in the graph being partitioned can help improve partitioning. With ad-hoc queries, it is hard to know what aspects to optimise for, so this aspect of partitioning can be ignored. The main interest is in algorithms that deliver a reasonable split in a short time.

The overall benefit of partitioning depends on the ratio of in-partition traversals compared with between-partition traversals. Between-partition traversals mean that data has to be sent between nodes, which could add significant time to a job, as well as using large amounts of network bandwidth.

2.4.1 Implementation

When implementing graph partitioning with the objective of finding antipatterns in the partitions, there are a number of factors to consider. The partitions may have varying densities of antipattern instances, resulting in partitions of the same size to be processed at differing rates. Load balancing is recommended to ensure that the distribution of partitions across worker nodes is managed efficiently.

Partitions should be as large as possible to ensure that boundaries are not frequently crossed when processing partitions. The cost of sending a request over the network to another node with the required vertex is orders

of magnitude higher than finding a local vertex. A balance must be found between the maximum partition size that can be loaded into memory and the requirement to have enough memory remaining to allow for the processing of the partition. Relaxing the constraint that partitions must be non-overlapping is one idea, though this may have implications in scheduling work based on which node a partition belongs to. Dividing the graph into overlapping subsets of vertices may decrease the number of cross-boundary edges required.

2.5 Cloud Computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [66].

There are several key characteristics of cloud computing: on-demand resources, resource pooling, elasticity, and measured service. On-demand means that users can provision more resources as they need them, while resource pooling allows a provider to use the same physical hardware to provide virtual resources to multiple customers, achieving economies of scale as each customer does not need dedicated hardware. In general, the customer has no control or knowledge of the exact location of the resources, but may be able to specify a high level location such as country or data center. An elastic service allows customers scale their applications as needed, requesting more resources when demand is high, and turning them off when demand is low. This may be more cost effective for customers than having hardware that is only used infrequently during times of high demand. The elastic properties of cloud computing mean that many applications built to run in the cloud are designed to be highly scalable. Due to the ease of increasing an application's resources, many aspects of a cloud computing application may be automated. For example, if load on an application reaches a preset threshold, new resources can be automatically started to reduce the load.

Finally, a measured service means that customers are only billed for the resources that they use, be it CPU time, network traffic, or storage space. Services are transparent in that users can see exactly what they are being charged for.

There are three types of cloud computing: software as a service (SaaS),

platform as a service (PaaS), and infrastructure as a service (IaaS).

SaaS means that an application is run in the cloud, and end users interact with it via the internet, using a browser or API. Users have no control over anything the application requires to run (such as OS, networking, load balancing, etc), other than user specific settings.

PaaS describes a system that provides users with a platform they can use to build applications on top of. This covers libraries, services and tools that the user may need. The user does not manage the network or OS, etc, but is able to configure the deployed applications.

IaaS is the most basic service, providing raw resources. It is then up to the user to configure these resources for the desired use. The user does not control the underlying hardware, but has control over the resources provided by the hardware.

Figure 2.4 shows the relationship between the three types of cloud services.

Resources that may be available from a cloud computing provider include processing capability, storage, network, load balancing, caching services, queuing services, database services, and load balancers.

Popular IaaS providers include Amazon Web Services, Rackspace Cloud, Right Scale and GoGrid. Some of these providers also have PaaS offerings, such as Amazon's Elastic Map Reduce, which provides a platform for running on-demand Hadoop jobs. Hadoop will be discussed in the next chapter.

Notable PaaS providers include Microsoft Azure, Heroku, and Google App Engine.

Providing a SaaS or PaaS product built on IaaS would be very useful for this thesis's objective. A system based on an IaaS platform would allow rapid scaling of computational resources to meet the required use cases.

2.6 Buffering

A data buffer is a temporary location for storing information as it is moved between locations. Buffers are typically used around input and output as a way to account for differences in the data transfer or processing speeds of the sender and receiver. Data is put in a buffer so that the sender or receiver does not have to wait for its counterpart. Another use for buffering is to reduce the amount of overhead when data is transfered over a network. If there is a relatively large amount of overhead required to setup the connection that is independent from the amount of data being sent, then sending larger

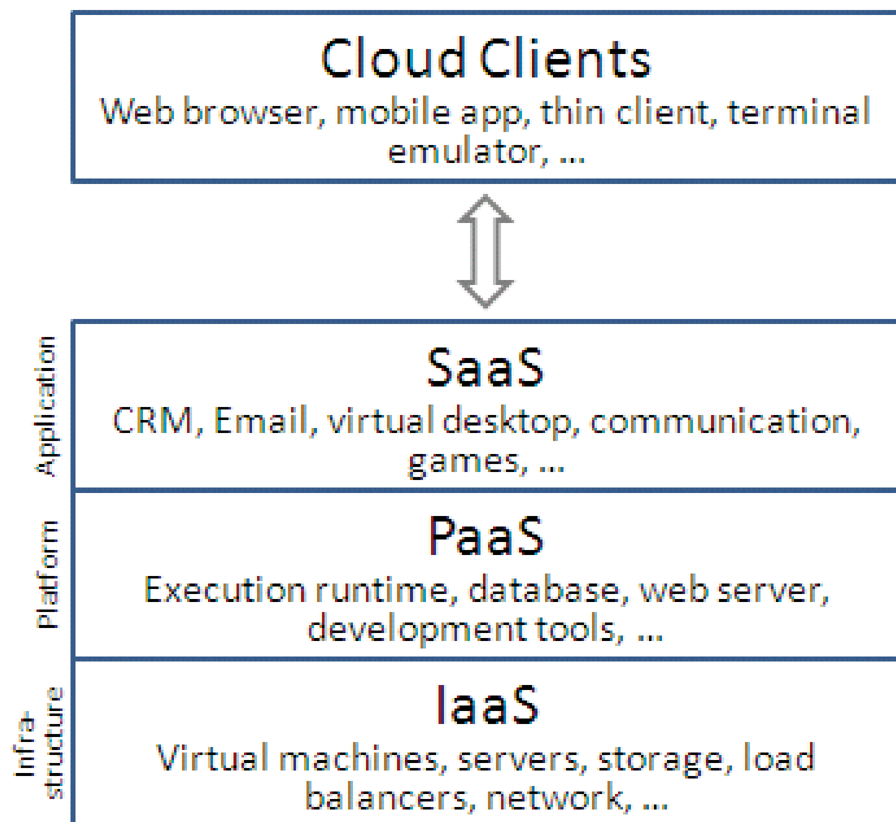

Figure 2.4: *Cloud service layers.*

blocks of data can help reduce the amount of time spent on overheads. To get larger blocks of data, the system can wait until it has several smaller blocks to send, then package and send them as one block. The receiver is usually able to break the larger block down into the original smaller blocks and process them.

Chapter 3

Requirements and Validation Metrics

In this chapter a set of requirements for the distributed system to be built is introduced. Not all of the requirements have equal importance. Metrics that can be used to compare designs and measure the usefulness of developed systems are then described. The systems looked at in the next chapter, 4, will refer back to this chapter as existing frameworks and platforms are evaluated.

3.1 Requirements

The following are a list of requirements for the distributed system. Numbers one and two are the most important requirements, three to eight are nice-to-haves.

1. **Scalable** This is one of the core requirements, and is explained in more detail later in this section.

2. **Result Streaming** The other core requirement, also explained in detail later in the section.

3. **Open Source** The resulting system should be released under an open source license so that it can be used by other academics and the wider software community. This is not required from a research perspective, but helps validate the results by allowing other to carry out the same experiments. Once researchers are statisfied with the validitiy of the system, it can be used as a basis for new research.

4. **Simple API** By hiding much of the complexity of the system behind a simple API, the system can be made easier to use.

5. **JVM-based** The GUERY motif detection engine that will be used is written in Java. Using the same platform, the JVM[1], throughout the project reduces complexity.

6. **Graph Partitioning Support** By supporting graph partitioning, the system can process graphs that cannot be processed by a single node.

7. **Fault tolerance** Failures, either in software or hardware, are inevitable. As the size of the system grows, more hardware must be used, increasing the chances of a component failing. Fault tolerance allows the system to accommodate some failures, such as losing a worker due to hard drive failure, while the rest of the system remains functioning. Some failures, such as a core switch failing, will cause system failure, as the parts cannot communicate. Ideally, the system should be able to handle common types of failures gracefully.

8. **Modular** In order to make the system maintainable and easily changed, the system should be modular, such that software components can be modified without unrelated components requiring modification. This should also make it easier for third parties to use.

There are two primary requirements of the resulting system: scalability and result streaming. These two requirements are explained in detail in the following subsections.

GUERY integration is made simple when the system runs on the Java Virtual Machine (JVM).

To ensure the final product is usable by anyone without having to pay license fees, all of the software used must be available under an open source license.

The final system should have a simple API to add new jobs to the system. The minimum required data is the graph file to analyze, and a file describing the query to run. Other information, such as the number of workers to use, or the number of partitions to divide the graph into may be included if necessary.

[1] JVM stands for Java Virtual Machine and is the platform that Java runs on, along with many other languages such as JRuby, Clojure, and Scala.

Graph partitioning is desired as it enables the system to process graphs that are too large to be processed by a single node.

3.1.1 Scalability

Scalability is the ability of the system to increase the amount of data it can process as hardware capacity is added. An ideal system scales linearly, meaning that if the number of nodes in the system is doubled, the processing power / throughput of the system is doubled. Linear scaling is rare due to overheads associated with initializing data processing, communication between nodes, etc. Hence, the aim to be as near to linear scaling as possible.

There are two main axes of scalability, vertical and horizontal, which are also known as scaling up and scaling out respectively. Horizontal scalibility means that a system can be scaled by adding more machines. Vertical scalability means that more resources, such as a better CPU or more RAM, are added to existing nodes in the system.

Ideally, nodes can be added or removed from the cluster without requiring a restart of the job or cluster. This is so that the system can be scaled down and up on demand, to avoid wasting resources when there are few jobs running and keeping job completion times low when there are many jobs. This is known as elastic scalability.

3.1.2 Result Streaming

Result streaming means that results are availabe in real-time as they are produced. This is in contrast with batch mode systems, where results are not available until all data has been processed. This is useful as it allows users to start analysing the data produced as soon as the job starts, rather than waiting until its completion. Another advantage is that it is not necessary to store the complete result set at any one time as results can be processed as they are computed.

There are a couple of common design patterns used to implement streaming systems. The pattern used depends on who is controlling the processing; client or server driven. Client driven means that a client obtains an iterator object from the result producing engine and requests more results from the engine by calling the next method of the iterator. This causes the engine to calculate more results and make them available to the client via the iterator. This is known as the iterator pattern[52].

Engine driven result streaming is based on a client observing an engine. A callback method on the client is called with the new result as an argument each time the engine computes a result. This design pattern is known as an observer[52]. The client may communicate to the engine that the client does not want any more results by having a boolean return value on the client callback method, and the engine halting processing when the callback method returns false.

Engine driven result streaming is analogous to how a sequential access (SAX) parser works. SAX parsers were originally designed for processing XML documents. An XML document consists of a hierarchy of nodes. The usual method for processing an XML document is to use a document object model (DOM) parser, which loads the entire document into memory. A SAX parser processes an XML document by iterating through the nodes, and firing different callback methods for different kinds of nodes. A client can then implement the callback methods and wait for these to be called by the SAX parser. A major advantage of SAX style processing is that the entire input and result sets do not have to be in memory at once; only the input currently being processed, making it useful for working with large files and large result sets.

3.1.3 Outside Project Scope

The system is not required to support mutating graphs, since GUERY does not need to do this to support its operations. This greatly simplifies some aspects, as the system will not have to deal with propagating changes to the graph structure between nodes.

Fault tolerance is a desired property of the final system. Some faults, such as losing a network switch that connects a cluster of workers, cannot be remedied without manual intervention. However, if a worker in the system fails due to a hardware fault on the worker, the system needs to adjust and carry on running without having to be restarted or repaired, and with no result data duplicated or lost (though the order of outputted data may change). If the time restrictions become too tight, fault tolerance may be ignored in preference for a running system, even if it is not perfect.

3.2 Validation Metrics

These are the metrics that can be used to evaluate the usefulness of the final system. Most of these are not measured values, but properties of the system that should be looked at closely.

Correctness and Completeness If a system is correct, it means that the system does not generate false positive results. This is defined relative to the results produced by GUERY when running on a single machine.

A complete system is one that returns all results, i.e. there are none missing. This means that every motif instance that exists, is included in the result set. In order to test completeness, the results of the distributed system can be compared to an instance of GUERY running on a single computer. If the results are the same, then this suggests that the correctness and completeness of the distributed system are equivelent to that of a single multi-threaded GUERY instance. Note that the ordering of the results may differ. The level of consistency demonstrated with GUERY is only as good as the test cases used to prove consistency.

Setup complexity The complexity involved in setting up a cluster to run the distributed software. This includes aspects like installation and configuration difficulty, etc. This is a subjective measure, categorised as hard, medium, and easy.

Customization required The amount of work required to run GUERY in a system. This includes work such as custom implementations of required functions (a la Hadoop). This can be measured objectively in the amount of time required to perform the customisation.

Find first The time taken in seconds, to find the first result, from the moment the job is started. In the cases where buffering is used, this may be slower than expected.

Find first 10 The time taken in seconds to find the first ten results from the moment the job is started. This may be affected by buffering, i.e, if a buffer size of 100 is used, a worker may not return results until the buffer is full (i.e. there are 100 results).

Find all The time taken to find all the results for a given query from the time the job was started.

Synchronization overhead The cost (in time and computation power) required to keep the various parts synchronized. This can be measured in time, but may have vastly different values for systems of different computation modes. For example, a system that does a lot of work up front to avoid needing to communicate between nodes cannot be measured in the same way as a system that supports communication between nodes over the lifetime of a job. Hence, this will be refered to as a subjective measure, defined as low, medium, and high, where the value defines the overall impact of synchronization overheads on a running cluster.

Memory usage The amount of memory required by the system, with particular attention to the memory required by worker nodes.

Worker idle time The amount of time that a worker is active, but has no work assigned to it. This should be minimised to ensure the system makes best use of available capacity.

Chapter 4

Existing Platforms and Frameworks

In this chapter a number of frameworks and platforms for solving graph problems using a distributed system are reviewed. A selection of Java-based frameworks and systems that implement the described computation modes are presented.

Most existing systems are built to handle a batch style of computation, or processing a continuous stream of input. These kind of systems are not compatible with GUERY's observer style interface. GUERY requires a system that supports a continuous output stream from a fixed set of inputs.

Several of the listed systems were not released when this research was started, and were only available after research had begun to explore other options.

A number of systems were excluded for lack of documentation, or no longer being under active development. Given the number of available systems in this area, it made sense to focus on high quality systems.

This phase of research only looked at open source systems. This is because it is possible to understand how they work by looking at the code, as well as the fact that there is usually documentation available. This is in contrast with many commercial systems, where the internal mechanisms are not public knowledge.

The aim of this project is to create a system that runs on the layer shown as GUERY+ in Fig. 4.1. Each layer in the diagram runs using the layer below. This chapter is an evaluation of various systems to establish the suitability of the system for running GUERY, with the idea of using an

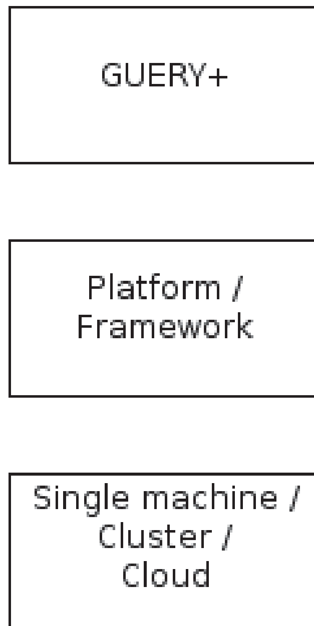

Figure 4.1: *A diagram showing the layers of abstraction with respect to GUERY.*

existing system to build the required distributed functionality.

4.1 BSP

Bulk Synchronous Parallel (BSP)[85] is a bridging model for designing parallel algorithms. BSP is based around having many processors connected via a communications network. Each processor has a fast local memory. A BSP computation is a series of steps, where each step is broken into three stages:

Concurrent computation Each processor may run different computations, but only using values stored in local memory. The computations are independent of one another.

Communication During this stage, processes may exchange data via messages.

Barrier synchronization At this point, the system waits until all processors have finished their communication stages.

By restricting the period that processes can communicate, an upper bound on the time taken to send a message of a certain size can be calculated. This makes it easier to reason about a system. Barriers avoid introducing livelocks or deadlocks as they do not create circular data dependencies. A downside to BSP is that if one process takes a considerably longer time than the others to finish in the computation stage, then the rest of the processors will have to idle until this single processor is finished.

BSP is useful if graph mutations are occuring, but graph mutability is not a requirement for this project. The nodes in the system will spend time waiting for the slowest nodes in the system to finish so the job can advance to the next superstep. GUERY does not require that all nodes advance in step, so this time spent waiting is wasted for no reason.

4.2 Pregel

Pregel is a distributed graph processing system developed by Google. It has been described in [64], but the source code has never been released. A number of open source implementations have been started, such as JPregel, Giraph, GoldenOrb and Apache Hama.

Pregel takes a vertex orientated view of a graph, and works by passing messages between vertices. The model it uses is a parallel programming model - a compute-communicate-synchronize loop, very similar to BSP. Global checkpoints are created during the job execution to ensure fault tolerance. If a node fails, then the system is reverted to the last global checkpoint, and processing is restarted from there.

In order to run a job, the executable program implementing the required Pregel methods is copied to many machines. One machine then becomes the master, and the rest workers. The master partitions the graph and the input, and ensures distribution among the workers, keeping track of which workers own which partition.

Workers keep their assigned partition in memory. A job is composed of a series of supersteps, the same as in BSP. During each superstep, the framework executes a user-defined function on each vertex. This function can read messages sent to the vertex during a previous superstep, send messages

to other vertices which will be read in the next superstep, and modify the state of the vertex and its outgoing edges. A program is complete when all vertices in the graph have voted to halt and there are no messages for delivery.

Each worker keeps a section of the graph in memory. When the cluster is started, one of the workers becomes a master, responsible for keeping track of which workers have which vertices and for coordinating actions.

To reduce the number of messages, the Pregel model supports combiners and aggregators. Combiners define methods for combining several messages into one. There is no guarantee that combiners will be used. Aggregators are a mechanism for global communication and monitoring. Each vertex can provide a value to an aggregator in each superstep. The system then combines those values and makes the result available in the next superstep.

4.3 Hadoop

Hadoop is a framework for distributed processing of large data sets with a cluster of computers using MapReduce[47] computations. The programming model is kept simple, abstracting away many of the low-level concerns. It is designed to scale from a single machine to thousands of machines. Each machine has its own processors and local storage. Hadoop is designed to detect and handle hardware failures so that commodity hardware can be used. It is the responsibility of the software to deliver a highly-available cluster.

When talking about Hadoop, there are three subprojects that are usually included in the definition: Hadoop Commons, Hadoop Distributed File System (HDFS), and Hadoop MapReduce.

Hadoop Commons is a set of utilities to support the Hadoop subprojects that includes file system, remote procedure call (RPC), and serialisation libraries. HDFS is the primary storage system used by Hadoop applications. HDFS creates multiple replicas of data blocks and distributes the replicas across nodes in a cluster to provide reliable data storage. Hadoop MapReduce is an implementation of the MapReduce programming model for parallel processing of huge amounts of data using clusters of nodes.

The Hadoop project is used as the foundation for many other projects based around large scale computing. Giraph, GoldenOrb, Apache Hama, HadoopDB, and HStreaming are all built upon Hadoop. These systems will

be discussed in following sections.

4.3.1 MapReduce

MapReduce is a framework for processing problems that are embarrassingly parallel. Embarrassingly parallel problems are problems that can be separated into a number of parallel tasks with little or no effort, and usually have no dependencies or communication between tasks. An example is finding the number of times individual words are used in a set of documents. Each document in the set can be processed independently, and the word count of each document combined to give a total word count for all documents in the set.

The definition of the MapReduce framework used in this document is based on a paper[47] published by Google, who also hold a patent for the framework. Other systems with features similar to MapReduce existed prior to Google's MapReduce.

MapReduce is based on two functions, map and reduce, which are inspired by similar functions used in functional programming. The framework is run on a cluster or grid of nodes, with one node having the role of master, in charge of job preparation, execution and management, and one or more worker nodes. MapReduce is an abstraction that allows users to write computations without having to know how data distribution, load balancing and fault tolerance are handled.

A problem is broken into two steps, a map step and a reduce step. In the map step, the master node takes input for a problem and partitions it into many smaller subproblems. This commonly entails splitting a large input set into many smaller sets. Each smaller subproblem is then allocated to a worker to solve. The worker processes the problem and passes the answer back to the master. Because there are no dependencies between the subproblems, all the workers can run in parallel. The reduce step involves the answers to all the subproblems being combined into the answer to the original problem. The output of each worker is processed by a set of reducers to produce the final answer. Although there is often significant network overhead in running a MapReduce computation, MapReduce can be used on much larger data sets than a commodity server can handle. MapReduce can also recover from partial failures during a computation as if a worker fails, the work can be rescheduled if the input data is still available.

The map function takes in a key value pair and produces a set of intermediate

key value pairs. The MapReduce library then takes all the values with the same intermediate key and passes them to a reduce function. The reduce function accepts a key and a set of values for that key and merges the values together. Listing 4.1 and Listing 4.2 show the map and reduce function implementations for a word count program. Word count is the canonical example used in map reduce tutorials, as it shows off the advantages of a map reduce computation, namely splitting of input data, many disparate computations, and reassembling results.

```
1 /*
2  * The key parameter is the name of the file.
3  * The file parameter is the contents of the file.
4 */
5 map(key, file):
6   for each word w in file:
7     emitIntermediatePair(w, 1)
```

Listing 4.1: Word Count Example map function

```
1 /*
2  * The key parameter is a word.
3  * The values parameter is a list of all the counts.
4 */
5 reduce(key, values):
6 result = 0;
7
8 for each v in values:
9   result += v;
10
11 emit(key, result)
```

Listing 4.2: Word Count Example reduce function

The map and reduce steps can be combined to form multistep job flows i.e. the output of a reduce step may be used as input to the next map step. There is no requirement that there has to be a reduce step. Some jobs may not need one. A job can also be composed of a map step with two reduce steps. Some map reduce frameworks require there to be a map step before a reduce step. In this case an Identity mapper, which outputs the input, can be used before the second reduce step.

Although map reduce may not seem ideal for graph algorithms, given the lack of communication between nodes, there are many papers on graph

algorithms with map reduce [68, 73, 86, 57, 63, 42, 44]. Graph algorithms run with map reduce include: shortest path, page rank, maximal clique enumeration, subgraph partitioning, connected component search, and processing RDF graphs. This suggests that there are reasons for using map reduce for graph processing.

4.3.2 Apache Hama

Apache Hama[8] is a pure BSP computing framework on top of the Hadoop Distributed File System for massive scientific computations using matrix, graph and network algorithms. It is not complete, but has the following features.

1. Job submission and management interface
2. Multiple tasks per node
3. Input / Output formatter
4. Checkpoint recovery
5. Supports message passing
6. Guarantees the impossibility of deadlocks or collisions in communication mechanisms

4.3.3 Giraph

Giraph[13] is a graph processing framework built on Hadoop, and is launched as a Hadoop job. Giraph is based on a Pregel like design, but adds additional fault tolerance to the coordinator process through the use of ZooKeeper, a centralized service for providing configuration information, naming and distributed synchronisation. Giraph uses the BSP model where vertices can send messages to other vertices during a given superstep. Checkpoints are initiated at user defined intervals, and can be used for automatic job restarts should any worker in the job fail. If the application coordinator fails, any worker can take over.

4.3.4 GoldenOrb

GoldenOrb[14] is an open source project for large scale graph processing. It is designed to help users better understand dynamic graph data as a whole and extract the knowledge embedded within.

It is similar to Giraph in that it enables developers to write vertex-centric algorithms to solve graph problems by passing messages between vertices.

It is based upon Hadoop and modelled after Pregel. GoldenOrb was launched in June 2011.

4.4 Graph Databases

Graph databases store nodes which have properties. Nodes are organised by relationships, which also have properties.

4.4.1 Neo4j

Neo4j[27] is an open source high performance graph database. It is designed to provide a programmer with an object orientated, flexible network structure that enjoys all the benefits of a fully transactional, enterprise strength database.

Neo4j supports querying the graph by using traversers. These are functions that are able to walk the graph from a starting node, making decisions about whether they will follow a particular path based on the properties of a node and its edges. Neo4j also supports a generic graph traversal language Gremlin[17] and its own declarative graph query language Cypher[12].

The major downside to Neo4j in terms of the objectives of this project is that it does not seem to support a streaming result set. Queries do not return until the entire graph has been searched and all results have been found. This means that it is not compatible with the streaming nature of GUERY.

4.4.2 InfiniteGraph

InfiniteGraph[20] is a proprietary distributed graph database, based on Objectivity DB[28]. It supports distributed data processing and massively parallel processing, with a variety of consistency modes. The core is written in C++, but there is an API wrapper for Java, and it will run across Windows, Mac and Linux. The core data model is a labeled directed multigraph. A multigraph is

a graph where each pair of vertices may have multiple edges connecting them. Graphs stored in InfiniteGraph can be queried using a traverser and predicates. Indexes can be defined on the graph to improve query performance.

4.5 Terracotta

Terracotta[34] is a collection of tools for creating clustered Java applications. Of particular interest is Terracotta Distributed Shared Objects (DSO). DSO is a system for sharing objects between machines on a network. This might prove to be a good basis for developing a distributed graph that can be read transparently by all machines in the cluster.

Terracota DSO works by using byte code manipulation to inject clustered meaning into existing Java language features. Mutating of the shared objects occurs in transactions. Each client has a window into the global heap, and objects are moved in and out of a machine's local memory based on use.

4.6 HipG

HipG is a framework for parallel processing large-scale graphs using high level abstractions and a cluster of machines. It is implemented in Java and designed for a distributed-memory machine. As well as basic distributed graph algorithms, it also handles divide-and-conquer graph algorithms.

HipG was released in April 2011.

4.7 Other Approaches

4.7.1 Custom GUERY clustering system

Building a system from the ground up designed specifically for GUERY was another option. If the requirements only specify graphs that fit in memory, and result counts are of interest rather than the motif instances themselves, then a management system that is capable of controlling many nodes running GUERY instances with local data could be useful. This would have all the speed benefits of running GUERY on a single machine while allowing for many graphs to be processed simultaneously by the system. In paper [49],

the authors are interested in scoring the edges that could potentially remove the most antipatterns from a system, but state that

> Investigating alternative combinations of antipattern sets and scoring functions is an interesting and promising field. There is no evidence that the combination we have used is optimal. Unfortunately, the validation for each set of parameters against the corpus is computationally expensive and takes several hours to complete, this makes a trial and error approach difficult. [49, pg 10]

A system that enabled multiple runs simultaneously allows for more experimentation. If the graphs are small, then there is no need for much communication between the nodes.

4.8 Summary

Given the overview of systems listed earlier, Hadoop is suggested as the best fit. BSP-style systems did not seem appropriate as GUERY already exists to discover motifs. Since the ability to mutate the graph is not required, many of the problems that come with running BSP-style algorithms that mutate a graph, such as maintaining data consistency across nodes and distributing changes, can be ignored. The interest in read-only data was another point in Hadoop's favour, as Hadoop has features useful for read-only data, but is lacking in ways to communicate changing data between nodes. The distributed graph databases Neo4j and InfiniteGraph offered useful features such as distributing graphs across multiple nodes, but did not have support for streaming results.

If there had been a mature implementation of a Pregel[64] style system, then that may have been a good choice. However, at the time, there was not. At the end of this project, there are a number of systems in varying stages of completeness: Giraph, Apache Hama, GoldenOrb, and JPregel.

4.8.1 Comparison Table

Table 4.1 shows a comparison between the listed systems. The cluster support column indicates in the system supports clustering over multiple machines. Documentation and community are rated either unacceptable, poor, acceptable,

System	**Apache Giraph**	**GoldenOrb**	**Apache Hadoop**
Core Language	Java	Java	Java
Open Source	Yes	Yes	Yes
Cluster Support	Yes	Yes	Yes
Computation Style	Vertex Oriented	Vertex Oriented	Map Reduce
Graph Data Support	Yes	Yes	No
Documentation	Acceptable	Acceptable	Excellent
Community	Acceptable	Acceptable	Excellent
System	**Apache Hama**	**HipG**	**InfiniteGraph**
Core Language	Java	Java	C++
Open Source	Yes	Yes	No
Cluster Support	Yes	Yes	Yes
Computation Style	Vertex oriented	Vertex oriented	Vertex oriented
Graph Data Support	Yes	Yes	Yes
Documentation	Poor	Poor	Very good
Community	Acceptable	Poor	Very good
System	**Neo4j**	**Pregel**	**Terracotta**
Core Language	Javaj	C++	Java
Open Source	Yes	No	Yes
Cluster Support	Yes	Yes	Yes
Computation Style	Vertex oriented	Vertex oriented	N/A
Graph Data Support	Yes	Yes	No
Documentation	Very good	Acceptable	Excellent
Community	Very good	N/A	Excellent

Table 4.1: A comparison of systems

very good, or excellent. Documentation is a rating of how much information (presentations, blogs, wiki, articles, books, etc) is available for a particular project. Community covers how widely used the system is, based on development activity, mailing lists, conferences, etc. The ratings were generated as part of the research into existing systems.

4.8.2 Why Hadoop?

Many of the systems reviewed did not have a feature set that met the requirements, or were not mature enough. A custom system to handle the underlying tasks such as communication between nodes and server management

was ruled out due to the complexities involved and the time required. Hence, it was simply a matter of choosing the system that was closest to the feature set required, and adding those that were missing.

Hadoop has been around for a number of years, and has been used by many different companies[1] to perform data analysis in finance, technology, telecommunications, media, research institutions, and other markets with large data sources[2]. Others have built or started building various types of systems on top of Hadoop (Hama, Giraph, GoldenOrb, HStreaming). This seems to suggest that the Hadoop application programming interface (API) is flexible enough to accommodate the problem to be solved; namely distributed motif discovery in graphs. Hadoop has been proven to work at a huge scale[3] (thousands of servers in a cluster) by companies such as Facebook, Twitter, and Yahoo!.

It is possible to add or remove servers to a Hadoop cluster at will. The system will detect new servers and adjust accordingly. This meets the scalability requirements discussed in the Requirements chapter. The actual scaling achieved is discussed in detail in 5.4.

Another compelling reason for choosing Hadoop was the wide range of documentation available: articles, books, websites, forums, mailing lists, and example code.

Hadoop provides a framework for running computations on large amounts of data. The input format must be specified, and an implementation of a map function and a reduce function provided. By default, Hadoop takes in key value pairs, and emits key value pairs. These can be reduced to a key value list.

Because the motif discovery software GUERY was already developed there was no need to implement the motif discovery algorithms in Hadoop, provided GUERY could be run on task nodes with the graph partitions as input.

Another factor suggesting Hadoop may be appropriate is the number of other groups developing graph computation frameworks on top of Hadoop[44, 68, 51, 14, 13]. The authors of [63] present methods for improving the run time of graph algorithms based on message passing between vertices, when running on Hadoop. The primary message passing example algorithm used

[1] http://wiki.apache.org/hadoop/PoweredBy
[2] http://www.cloudera.com/why-hadoop/
[3] http://www.quora.com/What-are-the-largest-Hadoop-clusters-to-date

is PageRank[29]. This evidence suggests that running graph algorithms on Hadoop is achievable.

The major differences between the aim of this project and the aims of many of the reviewed frameworks are threefold:

Scale of Graphs: This project was initially looking at graphs that will fit in the memory of an ordinary server, rather than graphs with millions or billions of vertices and edges. The main bottleneck in this case is the computation time. Being able to process larger graphs would be useful.

Support for many programs / algorithms: The systems and frameworks reviewed earlier are designed to accommodate a range of use cases. In the case of this project, there is only a single use case that needs to accommodated. Because the requirements of this single use case are known, the list of required features is well defined. For example, there is no need to support graph mutation, a feature that can add significant complexity to systems.

Streaming results: Existing BSP/MR frameworks are designed around batch processing. One of the goals of this project was to stream results as they were generated. The reason for this requirement relates back to the motivation for creating the system in the first place, which is architectural analysis of software programs. The idea is that a user of the system can submit their software for analysis, and immediately start seeing results. While the first results are investigated, more are computed in the background, removing the need for the user to wait for the analysis to completely finish before results are reviewed.

After reviewing the existing systems that might be appropriate for scaling GUERY, Hadoop seemed to be the best choice for building on due to its maturity and feature set. In particular, the following features are of interest.

Custom input formats supported It does not matter what format the input is in, as a custom input format reader can be easily implemented. Based on examples, these seem to generally be simple to write. Using a custom input format reduces the amount of code needed for parsing the input as part of Hadoop job.

Distributed cache and file system Ideal for distributing the complete graph and motif files to each worker in the cluster.

Counters Useful for showing how the job is progressing. Workers can update counters indicating statistics of interest; for example, how many motif instances have been found so far, for a given job. These counter values are viewable in the Hadoop web administration interface.

Proven Scalability Yahoo!, Facebook, and others have built very large clusters which proves that Hadoop is highly scalable in principle. The scalability this project achieves may be lower than the standard use cases as the requirement for streaming results

Community support A large number of companies have implemented Hadoop based systems, and there are many projects (open-source or otherwise) based on Hadoop. There is also an active mailing list, and companies that provide commercial support for Hadoop.

Written in Java Having all project code in the same language will help reduce maintenance and integration issues. It is possible to run any program that can read from standard input and write to standard output in Hadoop, however as GUERY is written in Java, using the Java MapReduce API's is logical.

Chapter 5

Hadoop GUERY Runner

Hadoop GUERY Runner (HGR) is a system for processing a graph with GUERY using Hadoop. It provides many of the features that were not available from Hadoop that were required for distributed graph processing. In particular, result streaming and distributing a section of the graph as input to each worker node. These features and others will be discussed in more detail later in this chapter.

HGR was initially designed to run on a local Hadoop cluster, but after infrastructure difficulties, was ported to use Amazon Elastic Map Reduce (EMR) and Amazon Simple Storage Service (S3). More information on HGR's use of Amazon EMR and S3 will be given later in this chapter (see 5.3).

The source code for HGR is available at `https://s3.amazonaws.com/msc-data/hgr-code/hgr-source.zip`. Instructions for extracting and running HGR are available in the appendix (see .2).

5.1 Design

A feature considered important in the design stage was modularity. As the desired features were still being considered and best implementation method were still being considered, it was important to be able to change and evolve parts of the system with minimal effort.

Figure 5.1 shows an overview of how the HGR system works. There are two main components, the Hadoop cluster and the result processing server. Input data is given to the Hadoop cluster, which is divided up between the

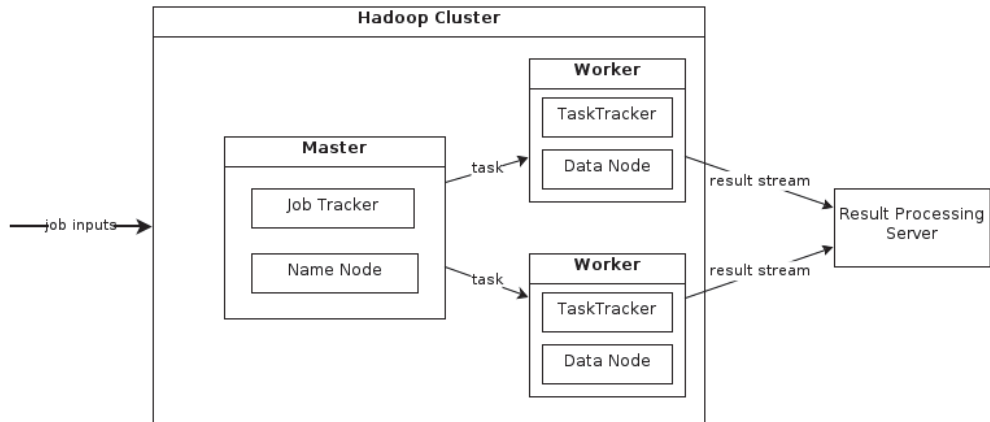

Figure 5.1: *An overview of the HGR system showing how data flows.*

task nodes by the master, and the results generated by the task nodes are streamed to the result processing server.

5.1.1 Input to HGR

In order for Hadoop to process the graph, it had to be converted into an input format that Hadoop could split and distribute to task nodes. This was achieved by creating a file containing all the vertex identifiers (ids) in a graph, and creating a custom input format reader that could split this file into input splits, which were then feed by Hadoop to the task nodes for processing. Each task node would load the complete graph into memory as part of its initialisation when given a new job. Vertex IDs would then be fed as input to the task node. For each vertex id, the task node will attempt to discover all motif instances that can be bound to a path starting from the given vertex. For example, if a task node received the input 1,2,3 it would attempt to find all the motifs starting at vertex 1 then, once it had exhausted the search space, would look for all motifs starting at vertex 2 and so on until there was no more input. Once the end of input has been reached, the task node will request a new set of input, or if there is no more input to be processed for the original job, a new job.

The one constraint with this method is that each vertex in the graph has to have a unique identifier. This is not a problem in practice, as most graphs

do, but for those graphs missing one, a preprocessing step can be done to add an artificial unique ID to each vertex in the graph. This has to occur before the splitting step.

One advantage in only dealing with graphs that fit in memory is there are no data locality issues as each worker has a complete copy of the data. This means that there is no overhead keeping track of file parts or which task node has what graph data.

5.1.2 Result Streaming

The idea behind streaming the results, as has been outlined earlier, is that we do not want the user of the system to have to wait for the entire job to finish before they can start analysing the results. To this end, results are converted to a format that allows them to be transported across a network. The serialisation format decided upon was JSON[23]. JSON is a human readable format, which assists with debugging, and is easy to work with thanks to its simple structure, and the maturity of JSON libraries. The downside is that it requires more network bandwidth and storage space than a binary format. The serialisation component of HGR is designed to be modular to make it simple to plug in another serialisation protocol, if required.

When a task node starts, it makes a transmission control protocol (TCP) connection to the result processing server. The address of the result processing server is given to the task nodes as part of the job configuration. An optional port may be specified. If no port is specified, the default will be used. All serialised data is then sent from the task node to the result server via the TCP connection. When the task node has completed, it sends a command to the result processing server indicating that it has completed processing.

One major issue with the current method of serialising motif instances is that each edge and vertex involved in the motif is sent by using its unique ID in the graph.

A motif instance contains information like $V_1 \rightarrow_{e_1} V_2$, which specifies vertex V_1 is connected to vertex V_2 by edge e_1. In order to make use of the serialised motif instance, it needs to be deserialised, reattaching the relevant vertex and edge information. In order to do this, a copy of the graph information is required on the server that will do the deserialisation. This may be a problem if the server only has a small amount of memory or multiple jobs using different graphs are being processed simultaneously. The server may receive many motif instances in a small amount of time, and the

more graph information that can be kept in memory, the faster the motif instances can be processed. It is important to prevent a backlog developing.

The alternative is to serialise the complete information for each vertex and edge involved in the motif instance, which removes the need to look it up each time a motif instance is received. The downside to this method is that it can substantially increase the size of each serialised motif instance, and hence increases the overall network bandwidth required.

For example, the JSON formatted example in Listing 5.1 contains the full vertex information instead of a vertex reference like 'v1181'. The set of properties listed under "vertex" is the same as the node has in the original graph.

```
{
    "vertex": {
    "id": "v1181",
    "container": "Azureus3.0.3.4.jar",
    "namespace": "com.aelitis.azureus.plugins.startstoprules",
    "name": "StartStopRulesDefaultPlugin\$TotalsStats",
    "cluster": null,
    "type": "class"
  }
}
```

Listing 5.1: Example of a vertex serialised in JSON

In order to reduce the amount of work required to change serialisation mechanisms, the process of converting a motif instance into its serialised format consists of applying a transformation function to the motif instance.

```
public interface MotifTransformer<V extends Vertex<E>,
  E extends Edge<V>, T, U> extends Runnable {

  public void initialize(String jobId, BlockingQueue<T> in,
    BlockingQueue<U> out);

  U transformMotif(String jobId, MotifInstance<V, E> motif);

  public void close();

}
```

Listing 5.2: MotifTransformer interface

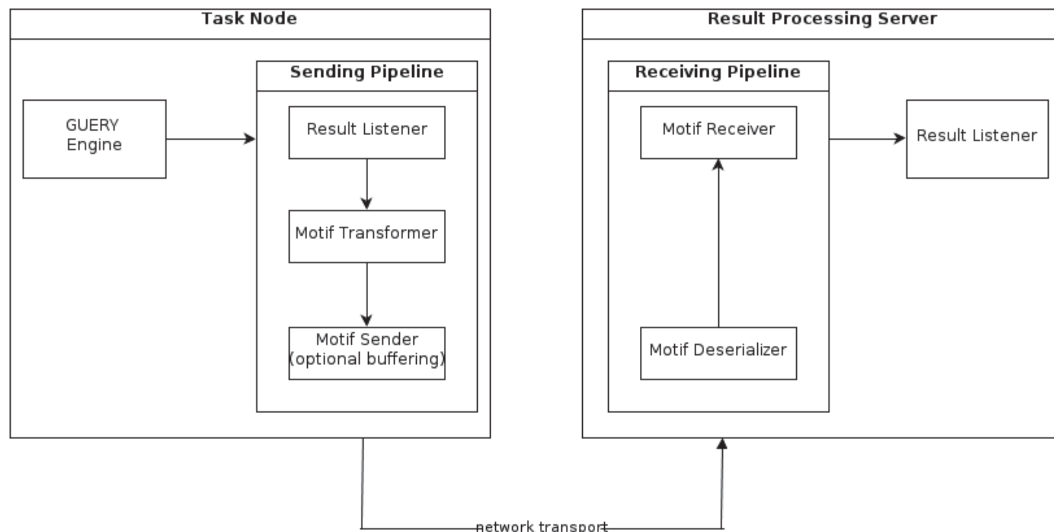

Figure 5.2: *The path a result takes once it is computed.*

The MotifTransformer is complemented by a MotifProcessingWorker that can deserialise the motif instance from the serialisation format to a generic instance of type MotifInstance<V,E>, which is an internal GUERY type.

```
1 public interface MotifProcessingWorker<V extends Vertex<E>,
2   E extends Edge<V>,U> extends Runnable {
3
4   public void initialize(Map<String, JobContext<V,E,U>> jobContexts,
5     BlockingQueue<U> inputBuffer, WorkerCompletedCallback wcc);
6
7   public void close();
8
9 }
```

Listing 5.3: MotifProcessingWorker interface

There is no function returning the deserialised motif directly. Instead, it is reported to a listener via the *ResultListener<V,E>interface*, using the *found(MotifInstance motifInstance)* callback method. The type represented by U in Listing 5.2 and Listing 5.3 must be the same in both.

Figure 5.2 shows the process of transporting a motif instance from a task

node to the result processing server.

Programs interested in the results of a particular job can register themselves on the result processing server and will be notified when new results come in.

Each serialised motif instance needs to include a field indicating which job it belongs to. This is so it can be correctly deserialised if there are multiple concurrent jobs active, each of which may be using a different graph.

5.2 Constraints and Limitations

The two biggest issues with the implemented solutions are the requirement to have all the graphs of the currently active jobs in memory on the result server, and the lack of fault tolerance in the streaming results section.

The result streaming component is the most significant weak point of the system due to the single point of failure introduced by the result server. There is no back up so if it fails, the system cannot continue until it is fixed. There is currently no way to change the result server address once a job has started.

One solution to the result streaming fault tolerance problem is to implement a form of localized check pointing. If task nodes are only fed very small sets of input, then by waiting until the node has finished processing all the input data, it can be guaranteed that there were no faults during processing. The advantage of this method is that the amount of processing that has to be redone should a task node fail, is reduced. The disadvantage is that it introduces a delay in the stream which may be noticed by the user. However, if the input set is sufficiently small, then the user may not notice.

The design is not infinitely scalable. This is due to sending all the results to one result processing server. It is possible to overwhelm the server when results are coming in faster than the server can process. This will cause a backlog. The simple fix for this would be to use multiple result processing servers, with each job allocated to a server using a load balancing algorithm.

5.3 Local vs Hosted infrastructure

Initially, Hadoop was installed on a cluster of 16 Pentium 4s with 1GB RAM. There were two key problems with this. First, the machines in the cluster did

not have enough memory to cope with the requirements of Hadoop as well as running HGR instances. The second problem was a lack of experience in running and tuning Hadoop clusters. This led to a large amount of time being spent trying to improve the performance of the cluster, often for very little gain. Hadoop is very customisable. There are over 160 environment variables that can be changed. These environment variables can be configured individually for each machine in the cluster, or set cluster wide.

Because of the complexity involved in administering the cluster, and the lack of high powered hardware to test with, it was decided to outsource the hosting of the cluster. The major provider for on-demand Hadoop clusters is Amazon, with its Elastic MapReduce (EMR) service. EMR allows you to run your Hadoop jobs on Amazon maintained infrastructure and only pay for the time you use the hardware. Amazon offers a large range of hardware types with prices varying according to the hardware specifications.

Amazon EMR[3] runs Hadoop clusters using Amazon's EC2[2] infrastructure. EC2 is web service for obtaining and configuring computing resources, with users paying for servers by the hour. In order to use EMR, a jar containing the Hadoop map and reduce functions to be used is uploaded. The input and output locations are then specified, and the number and type of instances to use for the job are selected. There are further options, such as giving a bootstrap action that will be run on the cluster nodes to prepare them for the job by installing custom software or configuring the environment, but these are not required for every job.

A major benefit of running on Amazon infrastructure is that it is very easy to experiment with cluster configurations and instance types. Since everything can be controlled via command line programs, the benchmarking process can be scripted. This allows the same benchmark to run concurrently on different instance types. Each cluster is independent of any others, allowing multiple configurations to be run at the same time. Amazon's Java software development kit (SDK) made it easy to create a benchmarking application that could test a variety of configuration parameters and output a comma separated values (CSV) file with the benchmark run information when completed.

Spot instances are unused Amazon EC2 instances that users can bid on. If the price you bid is higher than the current spot price, you will get the instances you requested. If the price rises above your bid, the spot instances you are using will be terminated immediately. Using a mix of spot instances and normal instances has the potential to reduce the running time of a job

for only a small increase in cost.

5.3.1 Hardware

Amazon measures the amount of computing power available on each instance type in terms of an EC2 Compute Unit. They define a compute unit as follows: one EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. This is also the equivalent to an early 2006 1.7 GHz Xeon processor referenced in our original documentation. [1]

As of January 2012, there are 11 instance types available for use in Amazon EMR, grouped into the following categories: standard, high-memory, high-CPU, and high performance computing.

The three hardware instance types used for the benchmarking discussed in the section 5.4 are listed below for reference.

m1.small (Standard Small Instance)
1.7 GB memory
1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)
160 GB instance storage
32-bit platform
I/O Performance: Moderate

c1.medium (High-CPU Medium Instance)
1.7 GB of memory
5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each)
350 GB of instance storage
32-bit platform
I/O Performance: Moderate

m1.large (Standard Large Instance)
7.5 GB memory
4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)
850 GB instance storage

[1] http://aws.amazon.com/ec2/instance-types/

64-bit platform
I/O Performance: High

5.3.2 Amazon Simple Storage Service

Amazon S3 [4] is a web service interface that can be used to store and retrieve data. It is highly scalable, reliable, secure and fast, being the same infrastructure that Amazon uses to run its own websites[2]. As at June 2012, S3 was storing over a trillion objects[3].

Amazon EMR requires the input for a job to be stored in S3 meaning the input has to be uploaded to S3, if it is not already stored in S3. Additionally, the output and log files of a job will be written to S3. HGR uses S3 to store graph and query data. Having data stored in S3 means EMR has faster access to it than uploading it from a local computer each time a job is run, and helps with scripting benchmarks because there is a set of unchanging reference data. Amazon provides a software development kit (SDK) that makes it easy to integrate S3 usage into programs. HGR uses the S3 SDK to store details of jobs and benchmarks so that they are not lost when the cluster shuts down. This allows benchmarks to be started with the knowledge that they will be shut down automaticaly when complete. The user simply has to download the resulting data from S3 at their convenience for analysis.

5.3.3 Problems Encountered when moving to Amazon

Moving from local infrastructure to Amazon hosted infrastructure was not without problems. In particular, the log files HGR produces were not being saved to the S3 logging directory that was specified. Logs generated by EMR were being saved. This made debugging parts of HGR very difficult as it was hard to get a complete picture of what was happening.

As a result of having so many different systems involved in a job, it was difficult at times to understand where a problem was occurring. This was mostly due to having log files on each machine, and no simple way to view them all simultaneously while the cluster was running. This issue was

[2]http://aws.amazon.com/s3/, accessed on 15/7/2012
[3]http://aws.typepad.com/aws/2012/06/amazon-s3-the-first-trillion-objects.html, accessed on 18/8/2012

further aggravated by a lack of experience administering Unix machines, as EMR nodes were running Ubuntu linux.

5.4 Experimental Evaluation

This section discusses the experimental evaluation of HGR. The experiments consisted of benchmarking HGR in a variety of configurations using a fixed graph and query.

The Amazon Machine Image used in the benchmarks was ami-e2af508b, running linux kernel version 2.6.38-8-virtual. The Java version installed was OpenJDK Runtime Environment (IcedTea 1.10.1) (6b22-1.10.1-ubuntu) with OpenJDK Client VM (build 20.0-b11, mixed mode, sharing).

The graph used to run the benchmarks was of the Azureus3.0.3.4 jar [4] from the Qualitas Corpus[30, 84]. It was selected as it was one of the largest graphs in the corpus. An Azureus graph has been used for testing in the development of GUERY[48]. It has 5378 vertices and 29231 edges. Using the graph and a circular dependency query [5], the query was executed on the graph five times for each of the Amazon EC2 instance types m1.small, c1.medium, m1.large using a range of cluster sizes: 1, 2, 3, 4, 5, 10, 15, 20, and 30 workers. This resulted in 135 separate benchmark runs. The results of the runs for each combination of instance type and cluster size were averaged to create an average run time. There are several other instance types that might be useful for running HGR (Amazon EC2 instance types c1.xlarge, cc1.4xlarge, and cc2.8xlarge) due to large amounts of processing power and memory. The cluster size is the total number of workers in the cluster, and does not include the cluster master server, or the result processing server. The result times do not include the time required to provision the cluster. Given that the loaded graph consumed less than 40 MB of memory, there was adequate memory for GUERY to run without needing to store data to disk.

The result processing server was a single m1.small instance for all the benchmarks. This proved adequate to keep up with the stream of results, even when running 30 workers on m1.large instances. This was tested by sampling the number of motifs awaiting deserialisation. It was found that this

[4] Available at `https://s3.amazonaws.com/msc-data/data/graphs/Azureus3.0.3.4.jar.graphml`

[5] Available at `https://s3.amazonaws.com/msc-data/data/queries/cd.guery`

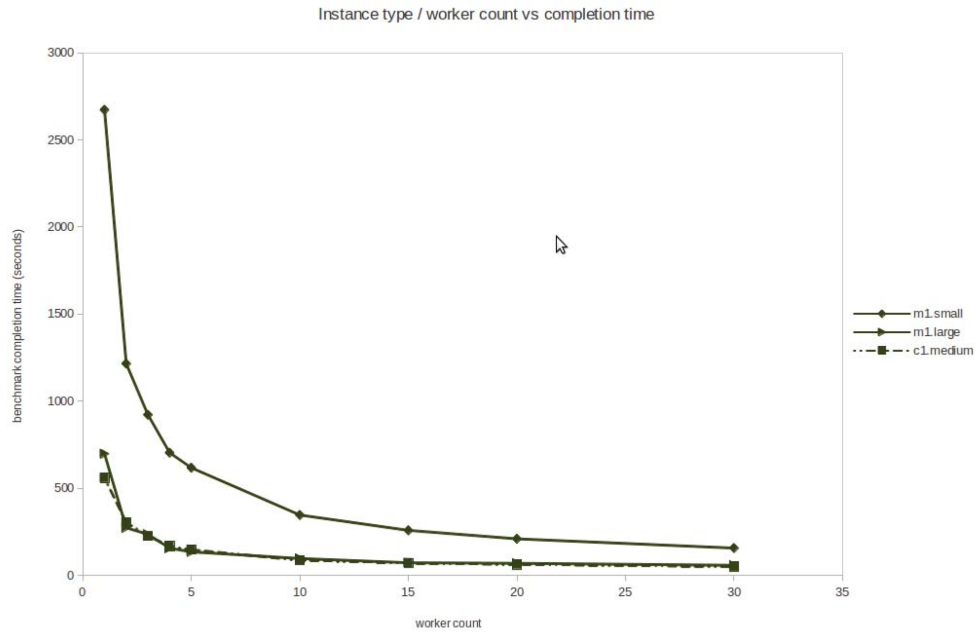

Figure 5.3: *A graph of job execution time per instance type vs worker count for all three instance types.*

was usually small (<1000 motifs). Given that the motifs were being buffered in groups of 1000, this indicates that they were being deserialised within a very short period of arriving on the result processing server. The total number of individual motif instances resulting from the circular dependency query on the graph was 611566. The raw benchmark results are available from `https://s3.amazonaws.com/msc-data/hgr-code/benchmark-results.zip`.

As the graph in Figure 5.3 shows, HGR does not scale linearly as the cluster size increases. The decrease in processing time by increasing the cluster size is significant. A perfectly linear increase is impossible due to the overhead of running Hadoop. This overhead includes job preparation steps such as distributing the jar file containing the computation implementation onto every node, setting up a working area, distributing work files, starting the job, and communication between nodes (counters, status messages, etc.).

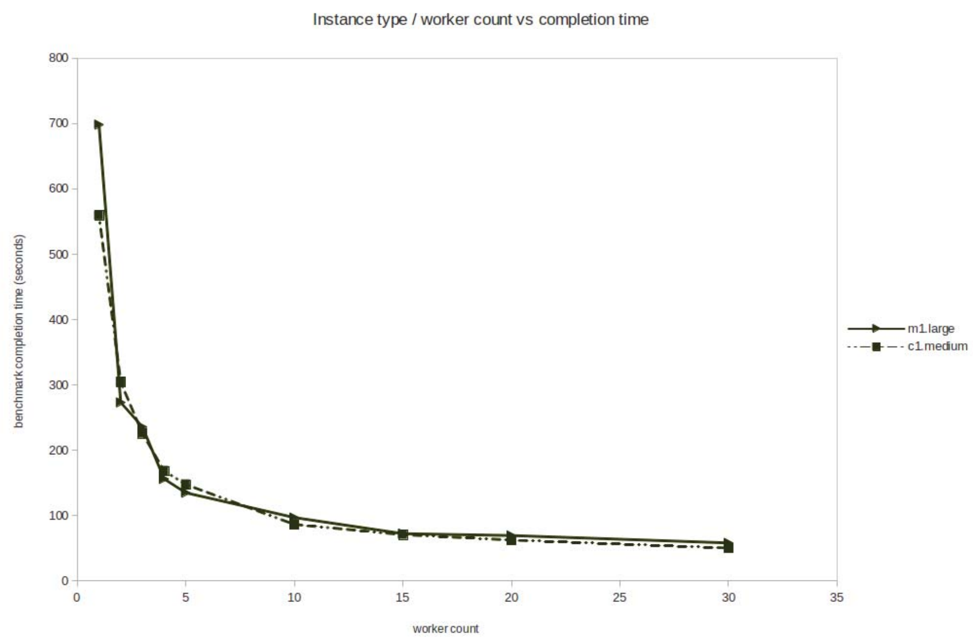

Figure 5.4: *A graph of job execution time per instance type vs worker count for m1.large and c1.medium instance types*

Some HGR specific overhead is included, such as each task node having to load the graph into memory which can take several seconds. This overhead increases with size of the cluster, but as the running time of the job becomes smaller, the percentage of job running time that is due to overhead increases.

Each task node in the cluster ran a single mapper that used a multi-threaded GUERY engine instance, setting the number of threads used by GUERY to the number of cores available. The number of cores used was dependent on the instance type the task node was running on.

Figure 5.4 shows only m1.large and c1.medium instances. Despite the difference in cost ($0.32 per hour for m1.large vs $0.165 per hour for cl.medium) there is very little difference in the benchmark times.

In order to test HGR's correctness and completeness with respect to GUERY, the output was compared with that of GUERY running on a single machine, running the same query on the same graph. The two outputs were identical in total motif instances discovered, though, as expected, the ordering was different. This means that HGR is as correct and complete as GUERY is for the Azureus graph used for testing. A sampling of motif instances was taken and there were no results found that appeared in GUERY, but not in HGR, or vice versa.

HGR was also run with a number of graphs, other than the Azureus graph used for benchmarking. In each case, the number of results was identical to the number produced by a single GUERY instance.

This does not prove that no input was lost in HGR, only that the input that may be lost does not result in any motifs. In order to check that all input is correctly processed, each task node keeps a record of the input sets it has processed. These task node input records can be reassembled and compared to the original input file, in order to verify that all the given input was processed by the system. A script produced a list of vertices in a graph, a job was run using the graph with the job, recording which vertices were used. The results of the job vertex use were then compared with the original list of vertices for the graph. If every vertex in the graph was used exactly once, the GUERY was processing every vertex in the graph. This procedure was run with several graphs, and in each case every vertex was processed once.

5.5 Requirements Validation

It has been shown in the previous section that HGR is highly horizontally scalable. Now it remains to compare HGR against the rest of the requirements outlined earlier. Live streaming of motif instances has been implemented successfully, with some issues around fault tolerance.

HGR is relatively easy to use, especially if used with Amazon's EMR service. HGR can be run on local Hadoop clusters, but would still need access to Amazon S3. Replacing or removing S3 would not be very difficult, as its use is limited to storing input and output. When using a local cluster, this could be replaced by a persistent Hadoop Distributed File System (HDFS) installed on the nodes in the cluster. The reason HDFS is not used in EMR is that the cluster is destroyed after each job is complete, and there would be no way to access the data after the job completed. In a local Hadoop cluster, the file system will remain intact once the job completes, so the output will be available in the future. The steps to use HGR are as follows:

1. Extract the vertex IDs from the graph file, and list them all in a single file. There is a program to do this included as part of HGR.

2. Upload data (graph file, vertex IDs file, and query file) to Amazon S3.

3. Start a result processing server. This involves running a program that was developed as part of HGR to receive motif instances as they are discovered. As workers discover instances, they send the serialized motif instances to an specific host and port. The program listening on this port recieves the motif instances, deserializes them, and makes the results available to other programs in real time.

4. Run a script that starts the job, providing as parameters the address of the result processing server, the S3 address of the input data, the S3 address for the output, the desired cluster size, and the instance type to use for the cluster.

No graph partitioning is occurring in HGR. The input is simply split between task nodes, but each node still needs a complete copy of the graph in memory. The size of the graph HGR can process is limited by the memory available in the task nodes.

HGR is not fault tolerant. As outlined earlier, the streaming result infrastructure is not fault tolerant. If the result processing server fails, any

active jobs fail. If a task node becomes unresponsive, then the entire job must be restarted, as there is no way of keeping track of which motif instances have been processed by the result server. This is contrary to Hadoop's fault tolerant design, which tries to avoid single points of failure.

All software used in the construction of HGR is open source so that requirement is met. HGR is freely available to anyone who wants to use it. See .2 in the Appendix for installation instructions.

The following subsections evaluate HGR with respect to the validation metrics outlined in the Requirements and Validation Metrics chapter.

5.5.1 Set up complexity

In order to get Hadoop running locally, there is a lot of installation and configuration to be done. In the early days of the project, this was taking up a large amount of time, and was not contributing anything useful to the project. Using Amazon's hosted infrastructure increased the project's velocity considerably, as it took care of a lot of the more complicated infrastructure set up. Amazon's EC2 and EMR API's were extremely helpful in allowing much of the repetitive work in setting up clusters and running benchmarks to be scripted.

5.5.2 Customization required

There was a large amount of customization required. Hadoop provided a good base to work from, but required the addition of the result streaming functionality. A custom input format reader was created to make it easier to process the graph data.

5.5.3 Find first

The time taken to find the first result when running on a Hadoop cluster is considerably longer than when running GUERY on a single machine. This is because of the overhead in setting up and running the job. Because of buffering done in the serialisation mechanisms, results are delivered in batches rather than one at a time. The buffer size can have a significant influence on the time until the first result.

5.5.4 Find first 10

This has the same issues as discussed in find first above.

5.5.5 Find all

Depending on the type of hardware used and the number of task nodes in a cluster, the find all time can be a much shorter time than running on a single machine, even with the Hadoop and network overheads. Obviously, running the same job on an Hadoop cluster with one task node and pure GUERY on a server with the equivalent hardware as the Hadoop node will result in the pure GUERY being faster. However, performance does scale as the number of task nodes increases, as Figure 5.3 in the Experimental Evaluation section shows.

5.5.6 Synchronization overhead

Because of the overhead of setting up the job across multiple machines, distributing files, etc, the delay is usually 20-30 seconds. This time is a relatively small amount of the overall processing time.

5.5.7 Memory usage

The amount of memory used is higher than pure GUERY due to the need to run the various Hadoop processes on the task nodes. These processes take care of things like task coordination, statistics reporting, etc. The memory required by the distributed GUERY client for the Azureus graph was less than , plus up to 200MB heap space used in processing.

5.5.8 Worker idle time

Worker idle time can be minimised by having multiple jobs running simultaneously on the cluster. This way, when a task node finishes, it can start work on another job. If only one job is running at a time, minimizing the work unit size may balance the work load across multiple workers more effectively than large work unit sizes. The reason for this is to lower the average time a task node takes to complete each unit of work. This thereby decreases the chances that all but one task node will be finished, with the task nodes having to wait for the longer running one.

5.6 Weaknesses

HGR's key weaknesses are due to the introduction of the result streaming component. Hadoop itself has been designed to be very fault tolerant, expecting that in a reasonable sized cluster problems will be encountered regularly. As such, it has robust mechanisms for dealing with failure. The main points of weakness in Hadoop are the name node and job tracker. If either of these nodes fail, then the cluster cannot continue working.

5.6.1 Result Streaming Fault Tolerance

Since there is no intermediate storage of results, if a task node fails then the whole job has to start again. This is because the system does not know which results have been computed already. One solution would be to have a very small input per task node, so that all the results are stored locally as they are calculated. Only when all the input for a given node has been processed are the results sent to the result processing node.

However, this does not get around the problem of result processing server failure. If the result processing server fails, then there is no way of deserialising the motifs or of notifying listening programs that results have been received.

Suppose there are N processing nodes and a graph of $|V|$ vertices. If a node dies, then the results for up to $\frac{|V|}{N}$ vertices will be lost, assuming an even distribution of vertices across nodes. Assuming a fixed probability that each node will fail during the running of a job, then the larger the number of nodes used in a job, the larger the probability that results for a job will be lost. This is unlikely to be a problem with small cluster sizes and small job sizes, as the job can be rerun with a low probablility it will fail. However it is likely to become a major issue when running long jobs on a large cluster.

Another issue arises due to GUERY's strong typing of nodes. Each graph has a custom vertex and edge objects which provide a native Java class that holds the properties for each vertex or edge. Using GUERY with a graph of data in a different domain would require the appropriate vertex and edge classes for that domain to be created. HGR would have to be re-compiled to work with a different data domain. A solution would be to define the class files used for vertices and edges in a configuration file, to be read when HGR starts the job. This would enable multiple simultaneous jobs in different domains, without having to re-compile HGR for the new classes. Each job could then include a jar with the edge and vertex classes to use.

5.6.2 Amazon Infrastructure

Provisioning a cluster in EMR is not instantaneous, and can take several minutes. This is because EMR must find and ready the number of machines required.

Another issue is that Amazon limits to the number of jobs that can be processed on a cluster. By default, each cluster runs a single job and shuts down once the job is complete. Having each job run on a separate cluster makes sense in many use cases, but for the purposes of this thesis there is more interest in running many smaller jobs rather than a few large ones. Hence the overhead of starting a cluster for each job may be significant (several minutes) if many benchmarking runs are being performed. This is obviously not acceptable when running tens or hundreds of benchmarks. In a Hadoop run each job may be composed of a number of steps where a step may use the output of a previous step as input. This provides a way to run multiple jobs on a single cluster. Each job is run as a step. This still does not completely solve the problem, as Amazon limit the number of steps in a job to 256. So after 256 graph query jobs, a new cluster must be provisioned. While not a perfect solution, this does have the advantage of removing the requirement to provision a cluster for all jobs.

The running jobs as steps feature was originally introduced to get around the problem of being charged the full hourly price per node, even though many benchmark runs were only taking a couple of minutes to run. The job limit only applies when running on Amazon EMR. When running on a local cluster there are no such limits, unless imposed by the cluster administrators.

5.7 Conclusions

HGR is a success in that it is able to perform the required processing with an increase in speed over a single machine. However, the costs associated with task node, or result processing server failure, are high. There is no fault tolerance, so if a single component fails then the whole cluster fails. This is not a great design.

HGR does what it was designed to do. There were a number of issues that were unclear at the beginning of the project, and HGR has helped highlight some problem areas that should be considered in the next design. One of the causes of these problems is that a batch orientated system is being used

in a non-batching manner. With batch processing, failure is reasonably easy to deal with as the output is isolated to one machine until the job finishes. At this point errors can be checked for and the job restarted if errors were encountered. When streaming results, the output of a job is rapidly passed through many components, each of which may act upon it and hence can be very hard to roll-back if there are errors in processing the job. This was compensated for initially by assuming errors would not happen, which is counter to the design of Hadoop where errors are assumed to happen and so the software is very tolerant of failure. If you want to be able to roll-back streaming results on an error, you need a way to track the results, such as giving a unique ID to each result. Even this method is fraught with problems. If results are shown to users as soon as they are calculated, it cannot be undone. There needs to be a way to deterministically assign a unique ID to each result so that if a job fails, the system only needs to calculate the results that should have been found after the point of failure, ignoring all results prior to the point of failure.

Another potential problem is that there was no particular attention given to treating graphs differently to other input. As such, there is no graph partitioning heuristics used, simply splitting the graph into small sets of vertices with little thought given to how the vertices relate. This works currently, but is a big barrier to working on larger graphs that will not fit in the memory of a task node. MapReduce is great for unrelated data, but when data is highly linked, such as graphs, it creates some problems. When workers need to communicate, MapReduce is not an appropriate computing paradigm.

Further research into comparing serialisation formats is needed. There may be an opportunity to reduce the amount of data each motif requires when serialising it, if another format is used. To further increase the ease of changing and comparing serialisation mechanisms, implementing the interface given in Listing 5.4 would be beneficial. This would allow the handing over of more control of the serialisations internals to the serialisation provider, and allow the provider to manage the streams of data. Having the MotifInstanceSerialiser interface could allow much more effective network utilization, as many instances could be compressed together, rather than the current method of individually compressing each result and buffering. If the output format is text based, like JSON, then a compression codec could be applied before sending to further reduce the output size. Another option for compression would be to use a transparent network layer compression system such as gzip. The compression

ratios that could be achieved depend on the format and content of the data being compressed. However, this could be a simple way to experiment with compression without having to make significant changes to the result streaming modules. Experimentation would be needed to determine time vs compression ratio trade offs.

```
interface MotifInstanceSerialiser {

    public void serialise(MotifInstance instance);

    public void serialiseCollection(Collection<MotifInstance>
      instances);

    public void flush();

}
```

Listing 5.4: A possible serialisation abstraction interface

On the programming side, HGR should have had unit tests for key functionality implemented early on. This would have helped check that the behaviour remained constant as new changes were introduced, and saved some time debugging certain aspects of HGR.

Hadoop was not a bad choice initially, as it was well documented and allowed rapid prototyping early on. As the project has evolved it has become less appropriate. For example, to solve the fault tolerance problems with HGR and task nodes, low level Hadoop internals would have to be used. These do not seem designed for public usage. As such, this prototype has allowed some ideas to be tested, and has shown areas that need further consideration. The problem was now more clearly understood.

A map reduce style program could be very beneficial for some use cases. In particular, the work described in [49] where the authors needed to make a small change to a graph, then query the graph to see the change in the number of antipatterns. By creating an input that contains the graphs and queries to be run, it would be possible to use Hadoop to massively speed up the experiments. One reason for the potential speed increase is that the authors are only interested in the total motifs found, not in further processing individual motif instances. This means that there is no requirements for nodes to be able to communicate, and no need for a results server. Developing HGR has helped identify some use cases for map reduce computations. While

HGR meets many of the requirements in Chapter 3, it is not an ideal system.

Chapter 6

Distributed Graph Processor

This chapter is a feasibility study into an alternative design that attempts to address the weakness of the HGR system. Using the lessons learned in designing and building HGR, this is an attempt design a better system. The alternative was named Distributed Graph PROCessor (DGPROC).

An important focus was simplifying the system to make it easier to use. Ideally, moving to DGPROC would be as simple as changing how the graph query language (GQL) engine is initialized and providing a reference to the input files. For example, Listing 6.1 could be replaced with Listing 6.2 and the same query could now be run in parallel across a cluster. A number of changes may be required to GUERY, adding new classes to support distributed graph functionality, but this should be transparent to the users of the API. After researching existing platforms and frameworks that may provide a platform to build upon, it was decided that a custom built messaging based system would be the best option for meeting the goals of the project. Many of the more heavy-weight approaches such as Giraph and GoldenOrb were designed to deal with problems that were not preset in motif discovery, such as supporting graph mutations.

```
1 GQLEngine gqlEngine = new MultithreadedGQLEngine();
```

Listing 6.1: Multithreaded GQL engine instantiation

```
1 GQLEngine gqlEngine = new DGPROCGQLEngine(configurationFile);
```

Listing 6.2: DGPROC GQL engine instantiation

6.1 Why a messaging based system?

The previous prototype, HGR, was based on Hadoop, and required a complex setup process. There were many features of Hadoop that were not required and which added unnecessary complexity to HGR.

The computation style required to meet the goals of DGPROC was a much better fit with a message passing paradigm than a batch style map reduce. The reason for the necessity of message passing is that workers need to be able to communicate with workers processing other partitions of the same graph. The need for this feature will be explained in more detail later on in the chapter.

Using a system where the workers can communicate provides many opportunities for improving on HGRs design. Instead of each worker loading a complete copy of the graph, each worker could be assigned a partition of the graph, and start computing motifs from vertices in that partition. If the motif requires an edge that crosses a partition boundary, the roles bound so far could be bundled up and sent to the worker who has the partition that contains the vertex at the other end of the edge. This receiving worker would deserialise the partial motif and carry on with motif discovery computation. This would be impossible in a map reduce style system such as Hadoop because workers cannot communicate with other workers.

Graph mutation features are not required, so multiple workers are able to use the same partition and not have any problems with graph structure losing synchronization between the workers.

Another advantage of using a message passing system is that a very low coupling between components can be achieved. This makes it simple to experiment with alternative implementations of components because there is a standard message protocol that the components must conform to. The presence of a standard protocol isolates implementation details from the interface. It allows the flexibility of having components written in different languages, but still able to communicate with each other through a common protocol. A message passing design aids testing and debugging, by increasing the isolation of components. Having a component with a single responsibility makes it simpler to test.

6.2 Messaging System Selection

There are a large number of messaging and queueing systems available, many of which are free to use. Only Java-based systems, or systems with a well documented and mature Java client, were evaluated. Another important requirement was that the systems be easy to install and maintain.

Some messaging systems such as Amazon SQS were ruled out because they were not available for local install. Having all of the infrastructure required for the system installed locally was desired to remove the need for external dependencies. Also, using a cloud hosted system such as SQS may introduce a large latency between broker and clients (terminology explained in section 6.3). Building a custom message delivery service was disqualified due to time constraints, the complexity involved, and the availability of free to use, widely implemented solutions.

Evaluated Messaging Systems:

RabbitMQ RabbitMQ[31] is an open source implementation of the AMQP[6] protocol, implementing a broker architecture. Message persistence is provided, but is highly tunable for different scenarios. The core of the messaging broker is implemented in Erlang, a language often used for highly concurrent applications. Clients are available for many languages. The documentation is focused on making it very easy for new users to get started.

ZeroMQ ZeroMQ[35] is an open source socket library developed in C++ that acts as a concurrency framework for clustered software and supercomputing. It provides a message queue designed for high throughput/ low latency scenarios, but runs without a dedicated message broker. ZeroMQ is based around asynchronous IO for scalable multicore message passing. It claims to be faster than TCP through the use of message batching. ZeroMQ supports N-to-N via fanout, pubsub, and request-reply messaging models. It has an active community, and clients for 30+ languages including Java, .NET, C, and Python. It is released under the LGPL license, with commercial support available from iMatix.

Apache ActiveMQ Apache ActiveMQ[7] is an open source messaging and integration patterns server. It also supports Java Messaging Service (JMS). Mulitple clients (Java, C, C++, .NET, Ruby, Perl, Python, etc.) and protocols (OpenWire, Stomp) are supported. Transport

protocols are pluggable, so a wide variety are supported. ActiveMQ is released under the Apache 2.0 license.

JMS Implementation The Java Messaging Service API[21] is a Java message orientated middleware API, defined in JSR914[24]. It is for sending messages between two or more clients, allowing different components of a distributed application to be loosely coupled, reliable, and asynchronous. Instead of tightly coupled transports like TCP or CORBA[11], an intermediary is introduced to allow components to communicate indirectly. JMS does not specify a wire protocol. There are a number of implementations of JMS, including Apache ActiveMQ, Apache Qpid, JBoss HornetMQ, RabbitMQ, and OpenJMS.

Apache Qpid Apache Qpid[9] is an open source cross platform enterprise messaging system which implements AMQP, provides message brokers written in C++ and Java, along with a variety of clients, including C++, Java, .NET, Python, and Ruby.

SQS Amazon Simple Queue Service[5] is a reliable and highly scalable hosted message queue. SQS supports an unlimited number of queues, each with an unlimited number of messages. The message body size is restricted to 64 KB. SQS runs in Amazon's data centers, so any computer needing to use SQS requires an internet connection.

Some other messaging systems were initially investigated, but quickly discarded after it became obvious they were not what was required, most commonly due to lack of features or likely integration issues. Systems looked at were generally restricted to Java based, unless they demonstrated that they had a good Java client and installing any other components in a non-JVM language was simple.

After a period of evaluation and research into the strengths and weaknesses of the above systems, RabbitMQ was chosen because of its reported throughput, feature set, and easy to use Java client. There is a good community around RabbitMQ (based on mailing lists and number of blog posts, etc.) as well as detailed documentation, and a book about it. Some of the other options like ActiveMQ supported so many features, it made it hard to focus on the relevant parts.

6.3 RabbitMQ

A RabbitMQ broker is a logical grouping of one or more Erlang nodes, each running the RabbitMQ application and sharing users, queues and exchanges. Often there will only be one node, but for high availability, multiple nodes may be clustered together with the state replicated across all of the nodes in a cluster. In essence, a broker takes messages from producers, and delivers the messages to consumers. The broker can also route, buffer, and persist messages as required.

A queue is a queue of messages. Queues may be made persistent, such that they will survive a restart of the broker, or they may be transient and only stored in memory. Each queue has a binding to an exchange that tells the exchange what criteria must be met in order for a message to be delivered to a queue, i.e. how the message should be routed. Routing controls which queues a message will be sent to, based on the routing key of the messages, and the type of exchange the message is sent to.

Messages have a label, composed of an exchange name and an optional topic, as well as a binary payload. This payload can be data in any format. It is up to the consumer to know how to read the payload. The broker only needs to be able to read the message header in order to determine how to route the message.

When using the AMQP protocol, a message is not sent directly to a queue. Instead, the message has a routing key attached, and is sent to an exchange. The exchange then decides which queues to put the message into by looking at the message's routing key and using the exchanges bindings.

There are three main types of exchanges: direct, topic, and fanout. Direct exchanges pass a message to a queue based on a direct match with the messages routing key. A message can only be received by one consumer of that queue. In direct exchanges, queues are often bound to an exchange by the queue name. Any message with a routing key matching the queue name will be delivered to the queue. The topic exchange does a wild card match on a messages routing key and the routing pattern specified in the binding. The message is delivered to the queues matching the pattern. Fanout exchanges send any message they receive to every consumer that is listening to a queue that is bound to the fanout exchange.

Direct exchanges are useful for supporting round-robin consumers, as each message can only be received by a single consumer. Topic exchanges are useful for logging, as it is possible to send a message to different consumers

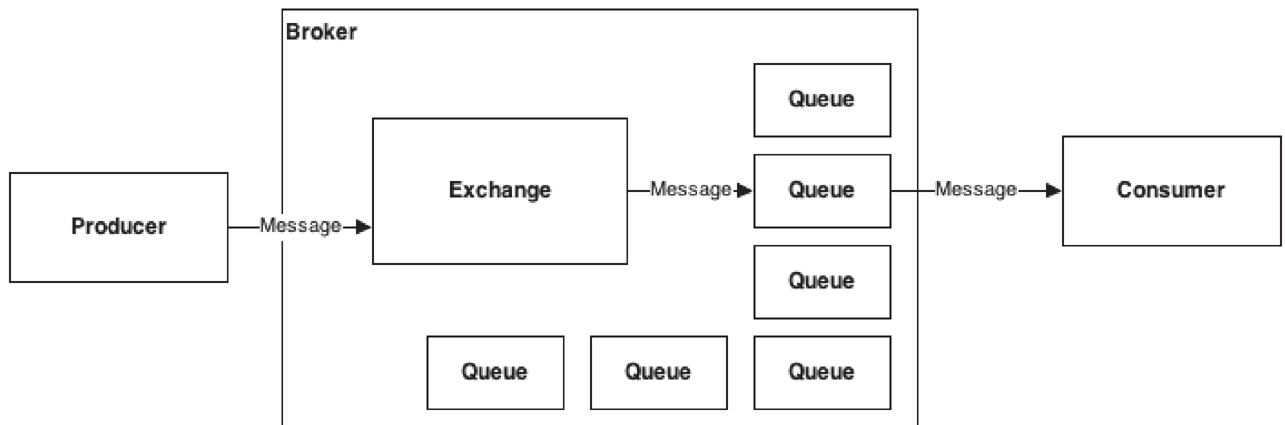

Figure 6.1: *The basic architecture of RabbitMQ.*

based on the component the log entry relates to, or the severity of the log entry.

A single TCP connection is shared by multiple channels on a client. This is more efficient than opening many TCP connections. Each channel is private, despite the fact they share the same TCP connection. It is very fast to create channels. Each channel is a connection to a different queue / exchange.

A useful feature of RabbitMQ is message acknowledgments. When a message is delivered to a consumer, the message will be held by RabbitMQ until a receipt acknowledgement is received from the consumer. If the connection to the consumer dies before a message is acknowledged, then the message will be requeued and delivered to the next consumer. This is useful behavior when creating tasks, as it means that even if a node takes a very long time to process a message, it will still be handled correctly without having to introduce timeouts.

Queues and messages can be marked as durable. This means that they are written to disk so that if the broker fails, they will survive a restart. This does introduce a significant decrease in message throughput however, as the disk must be accessed for each message.

6.3.1 Availability

Using a messaging system does introduce a single point of failure. If the server the message broker is running on fails, such that it can no longer send or receive messages, then the system as a whole can no longer function as nodes are not able to communicate.

This can be mitigated by using the high availability and clustering options of RabbitMQ. This allows two or more RabbitMQ servers to be connected together so that if one fails, another can take over message brokering. There are some types of failure that this cannot protect against, such as a network switch failure, but does reduce risk of being affected by a single server failing.

A peer to peer (p2p) messaging system would solve some of these problems, but may introduce other issues relating to tracking which nodes are available, etc. Cassandra[10] or a similarly distributed system might be a good model to base this on, depending upon the messaging semantics required.

6.4 Architecture

The goal was to build a system for processing motif queries that can run distributed across heterogeneous commodity hardware using a message passing framework. The system should be resilient to the failure of any worker node. The system should be able to process graphs that are too large to fit in the memory of a single worker, by partitioning the input graph to minimize edges that cross partition boundaries. This partitioning step will take place before the data is made available to workers. The system will be horizontally scalable supporting the ability to add workers during run time. Ideally all processing will be done in RAM, without the need for disk IO. The system must support streaming results as they are calculated. Data locality should be maximised by moving partial solutions to the node with the most applicable data. To this end, the system will need to support a scheme for identifying which partition contains a given vertex in the graph without having to contact a central authority.

There are four types of nodes in the system:

- **RabbitMQ Broker**
  This node type hosts a RabbitMQ server. It is responsible for running a broker that receives and delivers messages between DGPROC's components.

- **Coordinator**
  The Coordinator role is responsible for organizing the worker nodes by scheduling tasks and administering jobs.

- **Worker**
  The worker nodes run the motif detection queries. Workers should also be able to run the graph partitioning algorithms, depending on their hardware.

- **Result Subscriber**
  The result subscriber is designed to be run on the same server as a listener. The listener the component in GUERY that receives discovered motif instances. It is the result subscribers responsibility to notify listeners that new results have been received. The deserialisation step in Figure 6.3 is optional as some programs may work with the serialised motif form. A job may have many result listeners, so there may be many result subscribers for a job.

Figure 6.2 shows how the four node types are related. Arrows indicate that the source component can send messages to the target component. For example, the message broker can send messages to the log aggregator, but the log aggregator cannot send messages to the message broker.

6.5 Message Queues

In order for the system to work, a set of queues that will store messages for each separate part of the system are required. The queues used are explained in the following list.

- **Logging**

  This queue is an aggregate point for logging. The administrator of the system can use a log4j[25] configuration to define which queues messages will be logged to.

  This allows a consumer to be attached to particular queues to receive any logging messages, providing a central logging aggregator. Logs can be written to disk as well as on each worker. The queues are organised into topics so that subscriptions to only certain logging events are possible.

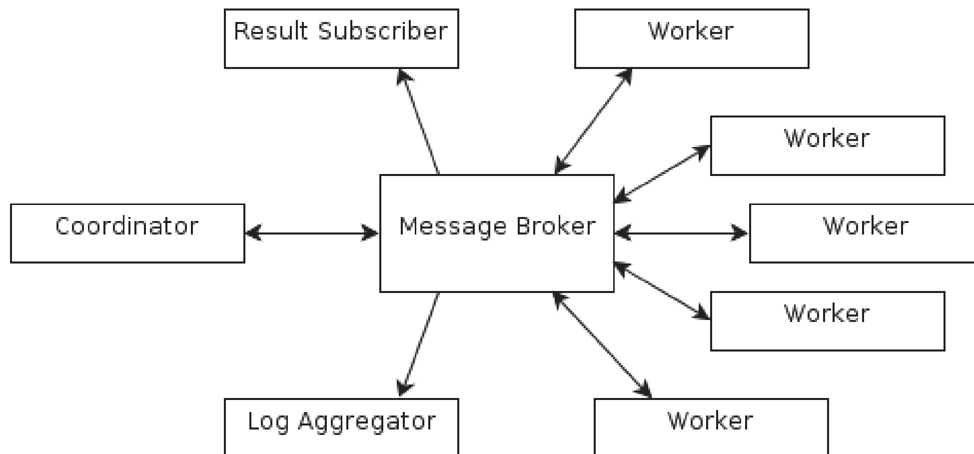

Figure 6.2: *The interaction between DGPROC components.*

- **Awaiting Partitioning**

  A queue for newly added graphs that need partitioning. Message information includes the storage location of the graph to be partitioned, the job ID it is for, and an optional hint to the number of partitions that should be created. Messages in this queue are picked up by a worker which then downloads the specified graph and partitions it. A hash of the graph file may also be included. This hash enables the processor of the message to check whether it has already processed the graph. If so, the existing partitions can be used.

- **Completed Partitioning**

  This queue is for graphs that have been partitioned, and are awaiting being added to the active jobs queue. It is the responsibility of the Coordinator to remove and process items from this queue.

- **Job Remote Procedure Call (RPC)**

  An exchange for RPC calls related to jobs. These calls are handled by the node with the coordinator role. Each message submitted to this exchange is automatically given a unique ID so that the reply can be

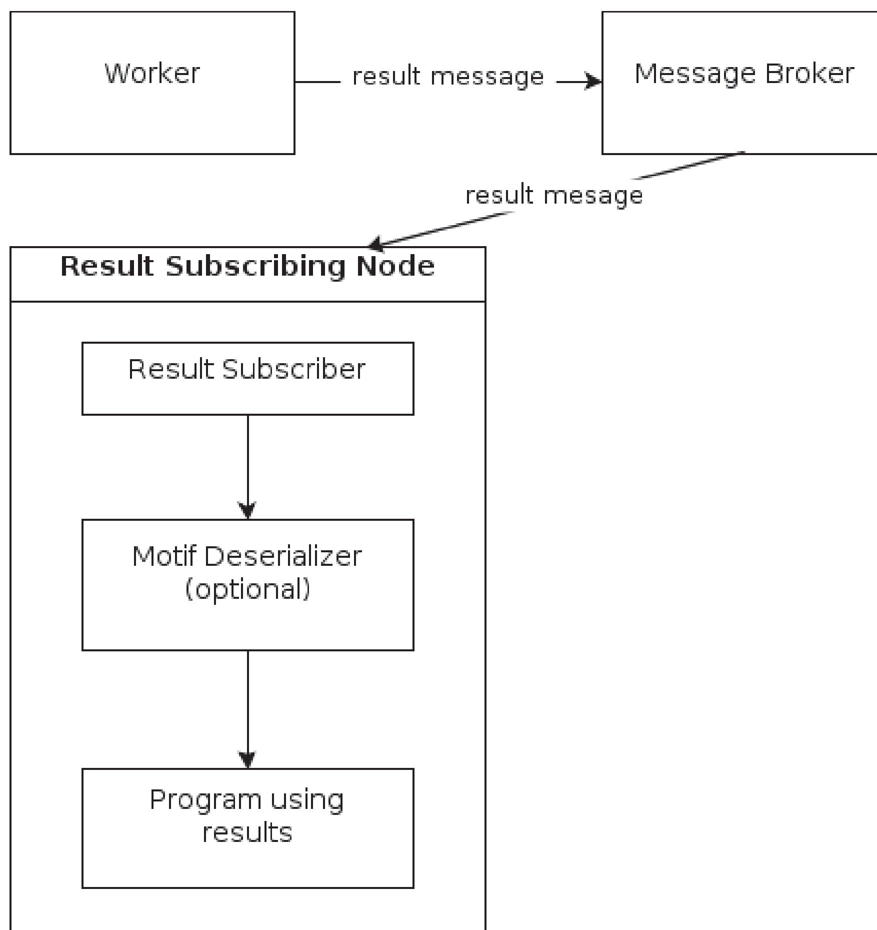

Figure 6.3: *A diagram of the result subscriber notification process.*

sent to the original submitter. Examples of RPC calls include finding out job status information, and reporting an issue with the worker.

- **Failed Tasks**

  A queue that workers subscribe to in order to receive the IDs of failed task attempts. It will be broadcast to all workers, so that they can ignore messages if the message ID is in this list.

- **Partition Tasks**

  Each message in the this queue contains a jobid/partition pair that requires processing. Workers get a message from this queue that tells them where to find the query and partition files they need to download for a job. Once the files have been downloaded and loaded, the workers subscribe to a specific ActiveJobs queue that will provide the actual units of work for the jobid/partition pair the worker has loaded.

- **Active Jobs**

  This is a set of queues, one for each jobid/partition pair. A message in an ActiveJob queue contains a list of vertex IDs that a worker is to process, or a partial solution where the next vertex bound to the motif is in the partition the worker holds. Partial solutions are created when a motif binding crosses a partition boundary.

- **Job Results**

  As each worker discovers motifs, it sends the motifs to a queue that has been set up specifically to receive motifs from a particular job. By using a fanout type exchange, there can be many subscribers to a single job's result set.

- **Heartbeat**

  The coordinator listens to this queue to hear worker heartbeats. If a worker does not deliver a heartbeat within a specified time, it is considered to have failed and the coordinator will have to adjust the task scheduling accordingly.

6.6 Components

DGPROC is designed to be modular and the use of messaging allows very loose coupling between components.

6.6.1 Messaging

All communication between nodes is done through AMQP messages. There are five key classes involved:

- **QEDeclarer**

  The QEDeclarer class holds information about the exchange, queue and routing keys a channel is using. It is an interface that is implemented to ensure that when different components send messages to a particular queue, they are all going to the same place. For example, any messages being sent to or subscribed to the logging exchange use the LoggingQEDeclarer class to obtain the exchange and queue names. This provides a single place to update, and all components will be updated with the new queue information.

- **MessagePublisher**

  This class is responsible for publishing messages to exchanges. It requires a QEDeclarer to specify where the message is going.

- **MessageConsumer**

  The MessageConsumer is able to be sub-classed via the MessageConsumerSkeleton class, or used as is. It can be used to process incoming messages indefinitely or halt after a timeout when no messages are received.

- **RPCServer**

  The RPCSever class is used to support making RPC calls. All incoming messages are handled by a method that can decode the message and make method calls based on the content of the message. The key difference between the RPCServer class and the MessageReceiver class is that the RPCServer has a mechanism for sending a result back to the calling client. It does this through the use of temporary queues, which the original caller subscribes to, in order to await a response to a specific call.

- **RPCClient**

  The RPCClient is able to send a message with a reply ID to an RPCServer, then waits for a message from the RPCServer with the correct reply id. In this way, clients can match responses to specific RPC calls.

All messaging in DGPROC is based around the use of these five classes. AMQP does not define a protocol for the message body, meaning that any format can be used to encode the message body. One requirement is that message sizes should be minimal. A binary protocol is good for this as the format can be specified to use a minimum of bytes to encode the message body. Another requirement is something that is easily translatable between components. In this case, text formats such as JSON or XML are useful because there are many tools for transforming these formats. Text based formats have another advantage: they are able to be read by humans. This can aid in debugging and testing.

It is possible to use a number of different messaging formats in the system. This would add complication, but would provide an advantage in that each part of the system could use an optimal message type. For example, partial motifs and input sets could use a binary format such as protobuf[15] which will reduce the size of the messages. This is important as there may be tens of thousands of these messages in the system at any one time. Control messages and other less frequently sent messages could be in a text format such as JSON to enable them to be written to relevant log files, making it easy for a human to see what is happening in the system.

If the messaging components are suitably abstract, it is possible to experiment with different protocols by simply changing the message encoder and decoder classes. Messaging formats can then be benchmarked to see if there is a noticeable change in running time or network bandwidth used.

6.6.2 Coordinator

The coordinator is responsible for making sure that the workers are scheduled correctly, keeping track of active jobs and failed tasks, ensuring failed workers are dealt with, and ensuring that a job is correctly processed.

Workers can query the coordinator to find out what task they should work on next, to obtain information about partitions, and to check on job status or to report a failure.

6.6.3 Worker

A worker obtains a task by asking the coordinator, then starts consuming work units for that task from the appropriate work unit queue. E.g. if the worker gets a task for job1, partition1, it would consume from the job1.partition1 work unit queue.

Increasing the processing capacity of the system is as simple as creating a worker on a new node and giving it the address of the message broker. The worker can then register itself with the coordinator and begin receiving tasks. This can be done while the system is actively processing jobs.

Having homogeneous worker hardware yields some benefits. In particular, it allows for better planning of how to schedule job tasks as all nodes have the same capability. It also means that when partitioning graphs, all workers can handle the same partition size so that it is not necessary to account for differences in partition size which may affect which nodes are able to process the partitions.

Each worker will maintain a local cache of graph partition files used. Once the cache exceeds a certain size, the least recently used entry will be deleted. This may help with decreasing the time taken obtaining new graph files over the network.

6.6.4 Partitioner

The partitioner is responsible for partitioning the graphs in such a way that the number of edges between partitions is minimized. It consumes messages containing a graph file location to be partitioned from a queue, retrieves the graph file, and then partitions it. The partition data is then saved somewhere that is accessible to the workers (such as networked storage). A message is then sent back to the coordinator containing a list of the locations of the partition files, which tells the coordinator that the graph is ready for processing.

The actual algorithm used can be changed for different jobs though the use of the strategy design pattern. Each partitioning algorithm implementation needs to implement an interface. This allows the partitioner to be oblivious to which algorithm is actually used.

Each graph gets split into n partitions. A queue for each partition is created. The set of vertices in a partition is divided into subsets of vertices that are sent to a partitions queue. A worker receiving a task to perform motif

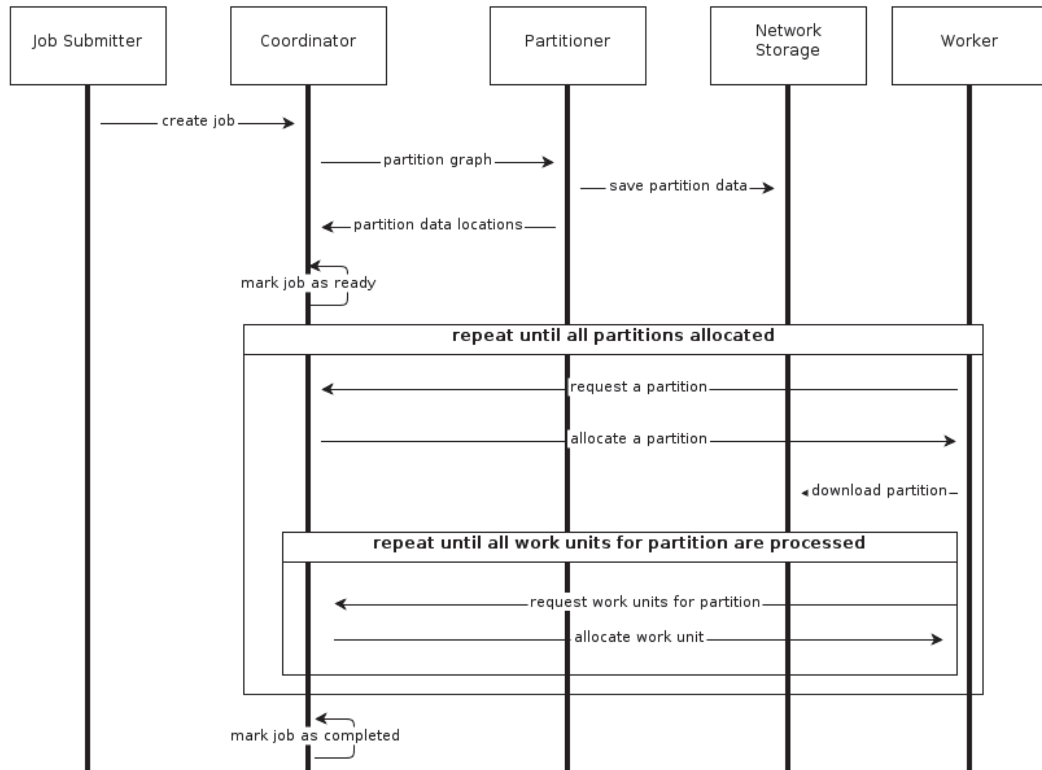

Figure 6.4: *A diagram of how partitions are created and used during job processing.*

discovery on a particular partition, then reads messages from a partitions queue and processes them until there are no more messages. At this point the partitioner will ask the coordinator for a new task. Figure 6.4 shows a sequence diagram of how partitions are used in job processing.

An optimization that could be applied if many queries are using the same graph is to calculate a hash of the graph file as it is uploaded and check that it has not already been partitioned. If it has been partitioned, the existing partition files can be used. This saves having to partition the same graph multiple times for different queries, but at the cost of the space required to store the partition files of the graphs. This may not be practical when dealing with very large graphs (tens of gigabytes).

6.6.5 ResultProcessor

As results for a particular job are produced, they are sent by the worker to a fanout exchange created specifically for that job. Any messages sent to the exchange will be delivered to all consumers. By subscribing to the exchange, any program may register itself as interested in the job results.

Depending on the format that the motif instances are serialised in, a deserialisation step may need to take place to convert the motif instances into a form that is usable by the result listeners.

The GUERY ResultListener interface expects that results are presented as an instance of the MotifInstance class. As such, there needs to be a way to convert the serialised motif instance into a MotifInstance, which requires references to the vertex and edge objects in the original graph. This means that a copy of the original graph is needed in memory on the result processing node. This is a problem when dealing with graphs that do not fit in memory, or a large number of jobs where the total graph memory space required exceeds the available memory. One solution would be to deserialise the motif results and create proxy vertices and edges that only have the basic information about a vertex or edge, for example just the vertices properties, rather than a full set of which other vertices can be reached. Another solution would be to include the vertex and edge data in the serialised motif. This way, there is no need to have the full graph loaded in memory but the downside is that the serialised motif instances are much larger due to all the embedded information. A generic ResultListener interface that can receive result motifs in any format could be added to GUERY. The motif instances can then be passed to a listening program, which can then decide what further processing to perform. This may avoid the need to completely deserialise each motif result received.

A result processor is any node that receives results.

6.6.6 LiveLogViewer (LLV)

One problem identified with HGR was that it was very difficult to get an internal view of the system. There were some Hadoop and HDFS statistics, and counters that could be viewed in the status web page, but it was very difficult to see an overall view of what the system was doing at a given moment in time.

With a message based system using RabbitMQ, there is a simple way

to send messages to a central exchange. A custom log4j[25] Appender was implemented, which allows the automatic logging of messages to a queue. The level of messages that get logged and packages that get logged is configurable. Each worker has local log files, but some messages can also be sent to the logging exchange with a routing key that includes the log message level and the component that sent it, in the format level.component. A LiveLogViewer can then be connected to the logging exchange and view all the log messages in near real-time. Messages can also be filtered to only listen to a particular level of logging message, or by which components generate the log message. For example, a LLV can connect with a filter specifying it is only interested in warning level or above messages from workers. The filter would be ẃarn.worker.

If there are no message receivers connected to the logging exchange, then logging messages are deleted. This is to prevent the exchange from using resources to store a record of all log messages received. Each machine in the system should have its own log file that stores all the log messages it has generated. If a complete interleaved record of the machines in the system is desired, users can listen to the logging exchange with a receiver that will write all the received messages to file. This may not be perfect due to the varying times the messages may take to be processed, but it is a good start. The log entries in the file could then be sorted by date for a truly accurate record.

Using the logging message exchange, it is possible to have a real-time view of what each component in the system is doing. It can be overwhelming with the number of incoming messages, but there are two places where filters can be applied. First is the log4j configuration file that each machine requires, and secondly by using a topic exchange. The real-time overview has proved invaluable in helping track down bugs and errors in the system.

It was relatively simple to create a custom logging appender that connects to a exchange and publishes log messages. Each logger can have multiple appenders, and the configuration can be set in a single log4j.properties file. The configuration file is distributed to all roles during the configuration of the system. This ensures that there is only one configuration file to maintain.

6.7 Scheduling and Task Management

The goals of scheduling and task management in DGPROC are as follows:
1. Minimize the time workers spend idle

2. Minimize job completion time
3. Minimize the time spent preparing for a new task.

Workers receive tasks through RPC calls to the coordinator asking for the next task. The coordinator can then decide which task has highest priority and assign it to the worker. The worker can then begin processing work units from the task's queue.

RabbitMQ uses round-robin scheduling when there are multiple consumers connected to a queue. This is useful as it means that when multiple workers are consuming the same partition work unit queue, work units will be handed out evenly between them, with no further management required.

Once a work unit queue remains empty for a certain period of time, then the worker needs to get a new task. This is one situation where issues with naive scheduling may be encountered. If worker A gets a new task, and worker B adds a partial solution to the work unit queue that worker A was previously reading, a worker will have to load the data associated with the work unit queue that B has written to. There is a trade-off between sitting idle waiting for more work units, and the overhead of switching tasks repeatedly. What trade off is made will depend on the scheduling algorithm in use.

There is also the problem of scheduling work so that the running times of jobs are minimized. If there are two jobs that are queued, each with 5 partitions, and the system has 5 workers, then each worker takes a partition of the first job. When the first worker finishes, it takes a partition from the next job, and the partition it was working on goes back in the queue. This means that it will not be picked up until all the other job partitions have been processed. This might introduce extensive delays in the running time of the job.

Exactly how scheduling should work to meet the goals outlined above requires experimentation with a running system. By implementing different scheduling methods, then running the same experiments with the different methods, it should become clear which, if any, methods are most suited to increasing the speed of processing jobs. It may be that there are different scheduling methods that work better with specific job types, so it could then be left up to the submitter of the job to decide which method is most appropriate, based on the characteristics of their job.

A job is broken into a set of tasks, each of which have associated work units.

The coordinator records which task a worker receives, so that it can keep

track of whether a job is finished or not. In order to tell if a job is finished, the coordinator can check the following conditions:

1. That there are no workers currently working on any task associated with the job.
2. That there are no outstanding failed tasks for the job.
3. That all the partition queues for a job are empty.

If all of these conditions are met, then the job is completed.

6.8 Graph Partitioning and Partial Solutions

To process large graphs, the graph needs to be split into partitions small enough to be loaded into the memory of a worker.

Graph partitioning should be an optional step. If the graph is small enough to fit in the memory of a worker, then there is no need to partition the graph. Doing so would only slow down the workers by introducing unnecessary communication required to deal with these partitions.

There are a number of different ways partitioning can be done. If sequential IDs are assigned to each vertex, a consistent hashing scheme is used. Nodes could then be assigned a set of vertices to process that fall within a given range. This way, each worker could be given a vertex ID ranges of each partition in the partition mapping table for a particular job. This would mean that instead of having to ask a central server which partition contains a given node, the worker can be straight away put it the correct queue for processing.

Partial solutions are used to pass incomplete motifs between workers when a motif crosses a partition boundary. This makes implementing fault tolerance more complicated, as when a node fails the system needs to be able to find all the partial solutions that were generated up to the point the node failed. This is so that when the task is restarted on a new node, the system does not recreate the same partial solutions leading to duplicate results.

If the system can tell what partition a given vertex is in, then it becomes possible to batch partial motifs together for a particular partition. This reduces the number of messages that have to be sent, and allows for a worker to process many partial motifs at once.

The size of the graph that can be processed is restricted by the size of the graph that can be partitioned.

6.9 Job Execution Workflow

A job is registered with the system via an RPC call to the coordinator. This call must include a location of the query to be run and the graph file to be queried. The location of the files needs to be accessible to the nodes in the system. The coordinator will create a job configuration and generate an ID for the job. This job ID will then be returned to the caller so that it knows which queue to subscribe to in order to receive results.

The next step is for the coordinator to send a message to the partitioning queue, specifying the graph file that needs to be partitioned. A partitioning node then downloads the graph and partitions it. The partitions are saved to files and uploaded to storage accessible to the workers. The partitioner is also responsible for splitting each partitions vertex IDs into small sets, so that they can be fed to workers as input. These work unit messages are sent to the active job queue for the specific job and partition. The partitioner then sends a message to the partitioning completed queue that gives the job ID and the location of the partition files.

The coordinator then creates the partition task messages that contain the job and partition pairs. Once these messages are placed in the queue, workers can receive them and start processing the graph.

When a worker receives a task message, the coordinator remembers which task the worker has. This is so that if a worker fails, the coordinator knows which task needs to be placed back in the queue for another worker to complete. When a worker has the correct graph and query file loaded, the worker starts taking messages from the queue for the particular job/partition work unit queue.

There are two types of messages that can be in a job/partition task queue. The first is an input set for the particular partition. In this case, the worker starts the motif search from each of the vertices specified. The second type is a partial solution, upon which the worker will bind the motif to the last vertex in the chain and carry on processing. Workers can use proxy vertex objects for all of the vertices in the partial motif that are not in the current partition.

A job is complete when there are no active tasks for a job and the job/partition queues associated with a job are empty. If there is no more input to process, then there can be no more results, and hence the job is finished.

6.10 Scalability

DGPROC is designed to be horizontally scalable. To add a worker, run the DGPROC worker jar on a new machine and point it to the RabbitMQ host. The worker will then register itself with the coordinator, and request a task.

Scaling is limited by the capacity of the RabbitMQ host to send, receive and store messages. If many large graphs are being processed simultaneously, RabbitMQ might not be able to store all of the input set and result messages in memory. The options are then to limit the number of simultaneous jobs, or allow the messages to be written to disk. If this limit is consistently hit, it may be more effective to store larger messages such as work units in a database. Each message then becomes a reference to the ID of the entry in the database that contains the full text of the message. This will introduce a small overhead, as each work unit will have to be looked up by the worker, but this is what databases are designed to handle. The advantage of this approach is that the RabbitMQ host will have a greatly reduced workload when dealing with the work unit messages.

Throughput of messages is unlikely to be a problem in any case, as RabbitMQ is designed for thousands of messages a second. However, it is not designed to be used as a database. Using persisted queues is significantly slower than non-persisted queues.

If the system outgrows a single host, two or more hosts can be used with each job having all its associated queues on a single server. This should make it possible to balance multiple jobs running simultaneously without needing to keep track of which part of a job is on what server.

6.11 Weaknesses

There are two main points of failure in this system. First, there is the message broker. If this fails, then none of the components can communicate, rendering the system useless. The second point of failure is the coordinator. If this fails, then the workers can continue to process data, but there will be no way to add new jobs, and all job state information will be lost.

Message broker failure can be mitigated by running RabbitMQ in a cluster using High Availability mode. This means that data is replicated across two or more servers so that if one fails, then others can handle the load. There is a risk a small amount of information may be lost if the cluster has not been

synchronized recently.

Using a central message broker, twice the bandwidth is required in order to send the message directly to the recipient. This is unlikely to be a problem due to the ubiquity of gigabit and beyond networks. The largest cause of messages is likely to be cross partition motif discovery or result messages. Both of these are likely to have relatively large message contents, which will increase the load on the RabbitMQ server.

It may not be effective to store the input messages on RabbitMQ, as it is designed for throughput, not as a data store. It might prove better to go with a task management framework that can communicate via messages. This would be relatively easy to do in the current design. A proper database could then be used to store the work unit and partial motif messages.

There is no streaming fault tolerance implemented in DGPROC. How fault tolerance could be applied to DGPROC is discussed in the next section.

6.12 Fault Tolerance in Streaming Systems

There are two common ways of implementing fault tolerance in streaming systems: replication and check-pointing. DGPROC is different to the usual streaming system in that it is concerned with output streams generated from a batch of input data, whereas most systems are based around processing streams of input as it arrives. As such, many of the techniques used to ensure fault tolerance are not applicable to DGPROC, or require modification to be usable.

Fault tolerance in distributed systems has been investigated by many researchers. There is a large body of literature on the classic problems, but less on the hybrid scenarios like DGPROC is trying to achieve. See the following papers for an overview: [50, 65, 61, 78, 76, 38].

How is the system going to keep track of which messages have been processed? One method would be to use a combination of small work units and buffering. As workers successfully finish processing a work unit, they acknowledge the message to the exchange. This will remove it from the queue. Each work unit message can be obtained by only one worker. If the worker fails, then the work unit message is put back into the queue when the connection to a worker is lost.

Buffering could be used to ensure that processing is completed before results are sent, or could be combined with check pointing to record the input

that has been processed. If the outputs are buffered locally, and a worker subsequently fails before the entire input is processed, another worker could resume processing from the last good checkpoint. This way, results are only held back for a short time, so the system may still appear as if it is streaming output. An added bonus is that the system does not have to worry about duplicate partial solutions, as partial solutions are only sent out once a check point is successfully recorded.

In order to detect a failed worker a heartbeat signal can be used. This can be implemented in the form of a daemon running on the worker that sends a heartbeat message to the coordinator at set time intervals. If the worker heartbeat is not received in a specified time, the system could mark the worker as failed and return the workers task to the queue. The worker should then be removed from the cluster. The worker can be re-added to the cluster once the issue has been fixed. Fixing the worker may require human intervention.

Having a progress monitor to ensure that the worker is actually making progress and is not stuck would also be useful. This could be something simple like a count of the number of nodes searched. An actual percentage of completion is unlikely, though one could be estimated based on the number of nodes in the graph and the partition that the worker has. This would indicate that progress was actually being made. The progress could be attached to heartbeat messages.

6.13 Current State of DGPROC

DGPROC has not been completed. There is still a lot of work to be done to get it into a usable state.

The basic partitioning infrastructure is completed, all that is lacking is a partitioning implementation. There are a number of algorithms that may be appropriate (see 2.4) for the system. Although the system is designed to process large graphs, the definition of large is relative. The system should be able to handle graphs with a few million edges, but not graphs with billions of edges. In order to handle the billions of edges graphs, DGPROC would need to further decentralise. Instead of storing the graph work units and partial motifs in RabbitMQ messages, a distributed database should be used. This may slightly slow down processing, as extra lookups will be required to translate the references to database objects stored in the RabbitMQ messages

into actual objects from the database.

Fault tolerance in DGPROC relies on the fact that each worker buffers the results as it runs a partial motif or unit of work. This still leaves a window where duplicate results can be introduced. If a worker fails during the sending of results, the system has no way to tell which results it was able to send out, and hence no way to remove those results from the system before the workers task is re-run. A better system for fault tolerance may be to look at moving to a check pointing system, similar to how BSP systems such as Giraph and GoldenOrb work. Checkpoints must be made in unison across the system or else they will not allow a complete restoration of the system to a particular point. The fault tolerance aspect of DGPROC needs to be re-thought as it is currently not very robust.

It may be worth moving to a distributed database to store the partial motifs and partition data. This would allow for a much looser control structure, as data would not be stored in RabbitMQ queues.

GUERY does not currently support partial motifs. This feature should not be very difficult to add, but is vital for extending the usefulness of DGPROC. Proxy vertices are also required to enable cross partition functionality. Generic graphs are important if GUERY is to be able to process graphs from other domains, eg bioinformatics. The vertex and edge types are hard coded into the application.

A further extension to GUERY that would be useful is to allow the output of a generic MotifInstance type. This would enable delayed deserialisation, rather than having to do it when the result is received. This feature could be useful in situations where the serialisation format is string based, such as JSON.

Scheduling and task management are both areas of DGPROC that need further work. The best method is likely to be decided by experimental evaluation of a variety of implementations. Using a system where nodes vote to halt processing in order to determine when a job is completed, such as in Pregel, may be a superior way of handling job completion than in the current system.

Scripting/automating the infrastructure has not yet been completed. Using an adapter such as JClouds[22] could allow a cluster to be created with a few commands. Using cloud computing providers such as Amazon[1] or Rackspace[32] means that an organisation does not have to own a collection of servers to get the benefit of a system like DGPROC. Having a simple way to get started with a query and graph allows a smoother experience for new

users, and encourages ad hoc use.

6.14 Conclusions

A good theoretical understanding of the problem has been gained and a rudimentary design for DGPROC produced. DGPROC was not completed due to time constraints. Given that several new systems with similar capabilities, such as Storm[33] and GoldenOrb, have been released over the last 12 months it would be worth investigating how these systems work, and whether there are any ideas that are applicable to the objective of this thesis.

A messaging based system is a good fit for dealing with this type of computation. Most other frameworks related to graph processing, such as Pregel, HipG, and Giraph, are based around some sort of message passing. There are many benefits to a messaging system, such as the possibility of aggregating messages, as well as a very loose coupling between components. It also makes adding a new worker very simple, as the worker is simply pointed to a message queue and can obtain a task and the required data.

DGPROC ends up solving two different problems. First, how to create and manage a distributed set of workers. Second, how is GUERY run on top of the workers. These are two very different problems. More investigation into task frameworks could have been done before attempting a new solution.

Chapter 7

Conclusions

Motif detection is important in a number of areas, particularly for software antipattern detection. As the volume of graph data grows, new tools are needed to produce meaningful insights into the data.

In this thesis, two methods of performing motif detection on graphs have been presented. This is a hard problem, and research is advancing rapidly in large graph processing. As such, the results of this thesis are no longer neccessarily the best way to perform the computations, now that there are new systems that were not available when this project was started.

The initial system, HGR, meets the requirements defined in 3 to a high enough degree that it is useful in solving the original problem. However, its lack of fault tolerance and graph partitioning led to a feasibility study of the design of a second system, DGPROC, which was not completed within the timeframe of the project.

It is difficult to compare the two systems, as DGPROC was not completed to a degree where it could be benchmarked against HGR. DGPROC was designed to remedy some of the flaws of HGR, around fault tolerance and lack of partitioning. Supporting these extra features may make the underlying design more complex, but these details should be transparent to the end user. DGPROC is certainly much easier to debug than HGR.

7.1 Contributions

The main contributions of this thesis are as follows:

1. A review of JVM based distributed graph processing methods.

2. A prototype system based on Hadoop that achieves near linear scaling in graph processing. The main limitation is the fact that the graphs must fit in memory on each node, which is fine for the software graphs being analyzed, as they are relatively small. One of the goals for the DGPROC systems was to remove the requirement that the graphs must fit in memory, but this was not fully completed. The system is capable of scaling across multiple cores and multiple machines. Proved that using on-demand infrastructure such as Amazon's EC2 can increase the ease of use of a system, reducing required administration knowledge.

3. Identification of an alternative use case for Hadoop when doing software analysis.

4. The partial implementation of a very flexible system for graph processing based that supports graph partitioning to enable the processing of larger graphs.

5. Demonstrated that cloud computing is a viable platform for distributed graph processing. Using cloud computing also removes much of the need to have in-depth systems administration knowledge to run the software on a cluster.

7.2 Future Work

DGPROC still has significant work required before it is ready for use (see 6.13).

Now that there are several reasonably mature open-source tools for working with large graphs, it would be useful to do a survey of them, and work out relative strengths and weaknesses. GUERY could be integrated with an existing graph processing system. This would likely greatly reduce the amount of work required to get proper fault tolerance and graph partitioning features.

A BSP system has the potential to meet the requirements. GoldenOrb, Giraph or a similar system are worthy of more investigation. The reasons they were not used in this project is that they did not appear to be very mature, and a lack of time to develop prototypes. In particular, a BSP would solve the problem of when the system should halt processing. Each

node can vote to halt processing, so when all nodes vote to halt, the job is done.

.1 Appendix GUERY Query Grammar

```
1
2 grammar guery;
3
4
5 options {
6 output=AST; ASTLabelType=CommonTree;
7 }
8
9 @parser::header{
10   package nz.ac.massey.cs.guery.io.dsl;
11   import nz.ac.massey.cs.guery.*;
12   import java.util.Map;
13   import java.util.HashMap;
14   import java.util.List;
15   import java.util.ArrayList;
16   import java.util.Collection;
17   import nz.ac.massey.cs.guery.mvel.*;
18   import com.google.common.base.Predicate;
19   import com.google.common.collect.Collections2;
20 }
21 @parser::members {
22   private DefaultMotif motif = new DefaultMotif();
23   private List<String> constraintDefs = new ArrayList<String>();
24   private Map<String, Class> typeInfo = new HashMap<String,Class>();
25   private ClassLoader processorClassLoader =
26     this.getClass().getClassLoader();
27
28       public ClassLoader getProcessorClassLoader() {
29     return processorClassLoader;
30   }
31   public void setProcessorClassLoader(ClassLoader
32     processorClassLoader) {
33
34     this.processorClassLoader = processorClassLoader;
35   }
36
37
38   public Motif getMotif () {
```

```
39      return motif;
40    }
41          private void exception(String message,Token token) {
42            StringBuffer b = new StringBuffer();
43            b.append(message);
44            b.append(" Position: row ");
45            b.append(token.getLine());
46            b.append(", column ");
47            b.append(token.getCharPositionInLine());
48            b.append(".");
49            exception(b.toString());
50          }
51          private void exception(String message,Token token,
52            Exception cause) {
53
54            StringBuffer b = new StringBuffer();
55            b.append(message);
56            b.append(" Position: row ");
57            b.append(token.getLine());
58            b.append(", column ");
59            b.append(token.getCharPositionInLine());
60            b.append(".");
61            exception(b.toString(),cause);
62          }
63
64          private void exception(String message) {
65            throw new SemanticException(message);
66          }
67          private void exception(String message,Exception cause) {
68            throw new SemanticException(message,cause);
69          }
70        private int intValue(Token t) {
71          return Integer.parseInt(t.getText());
72        }
73    private void addVertexRole(Token roleT) {
74      String role = roleT.getText();
75      motif.getRoles().add(role);
76      typeInfo.put(role,Vertex.class);
77    }
78    private void addConstraint(Token constraintT) {
```

```
79          PropertyConstraint constraint =
80            buildConstraint(constraintT);
81
82      // try to attach constraint directly to path constraint
83      Collection<PathConstraint> pathConstraints =
84        Collections2.filter(motif.getConstraints(),
85          new Predicate(){@Override public boolean apply(Object arg)
86            {return arg instanceof PathConstraint;}});
87      for (PathConstraint pc:pathConstraints) {
88        if (constraint.getRoles().size()==1 &&
89          constraint.getFirstRole().equals(pc.getRole())) {
90          // attach property constraint to path constraint
91          // this means that it will be evaluated
92          // immediately if paths are explored
93          pc.addConstraint(constraint);
94          return;
95        }
96      }
97      // else, add constraint to list of "global" constraints
98      motif.getConstraints().add(constraint);
99
100   }
101
102       private void addGroupBy(Token groupByT) {
103         String def = removeDoubleQuotes(groupByT.getText());
104         CompiledGroupByClause clause =
105           new CompiledGroupByClause(def);
106     String role = clause.getRole();
107     if (!motif.getRoles().contains(role)) {
108       exception("Group by clause " + def + " does contain
109         an input that has not been declared
110         as a role ",groupByT);
111     }
112     motif.getGroupByClauses().add(clause);
113       }
114
115         private void addProcessor(Token classNameT) {
116           String className = classNameT.getText();
117           Class clazz = null;
118           try {
```

```
119        clazz = processorClassLoader.loadClass(className);
120      }
121      catch (ClassNotFoundException e) {
122        exception("Processor class " + className +
123          " cannot be loaded",e);
124      }
125            if (!Processor.class.isAssignableFrom(clazz)) {
126              exception("Class " + className + " does not implement
127                Processor.class.getName());
128            }
129            try {
130        Processor processor = (Processor) clazz.newInstance();
131        motif.getGraphProcessors().add(processor);
132      } catch (Exception e) {
133          exception("Processor class " + className + " cannot be
134            instantiated",e);
135      }
136          }
137
138    private void addPathRole(Token roleT,Token fromT,Token toT,
139      PathLengthConstraint c,boolean findAll,boolean negated) {
140
141      String source = fromT.getText();
142      String target = toT.getText();
143      String role = roleT.getText();
144      if (motif.getPathRoles().contains(role)) {
145        exception("Path role "+role+" is already defined",roleT);
146      }
147      if (!motif.getRoles().contains(source)) {
148        exception("Source vertex role not defined
149          for path role "+role,fromT);
150      }
151      if (!motif.getRoles().contains(target)) {
152        exception("Target vertex role not defined
153          for path role "+role,toT);
154      }
155      if (negated) motif.getNegatedPathRoles().add(role);
156      else motif.getPathRoles().add(role);
157
158      typeInfo.put(role,Edge.class);
```

```
159
160     // build path constraint
161     PathConstraint constraint = new PathConstraint();
162     constraint.setRole(role);
163     constraint.setSource(source);
164     constraint.setTarget(target);
165     constraint.setNegated(false);
166     constraint.setMinLength(c==null?1:c.getMinLength());
167     constraint.setMaxLength(c==null?-1:c.getMaxLength());
168     constraint.setComputeAll(findAll);
169     constraint.setNegated(negated);
170
171     motif.getConstraints().add(constraint);
172   }
173
174   private String removeDoubleQuotes(String s) {
175         if (s.charAt(0)=='\"' && s.charAt(s.length()-1)=='\"') {
176           return s.substring(1,s.length()-1);
177         }
178         else {
179           return s;
180         }
181       }
182
183
184   private PropertyConstraint buildConstraint(Token constraintDefT) {
185     String constraintDef =
186       removeDoubleQuotes(constraintDefT.getText());
187     CompiledPropertyConstraint constraint = null;
188     try {
189       constraint =
190         new CompiledPropertyConstraint(constraintDef,typeInfo);
191     }
192     catch (Exception x) {
193       exception("Error compiling mvel expression "+constraintDef,
194         constraintDefT,x);
195     }
196     // check whether inputs are in roles
197     for (String role:constraint.getRoles()) {
198       if (!motif.getRoles().contains(role) &&
```

```
199          !motif.getPathRoles().contains(role) &&
200          !motif.getNegatedPathRoles().contains(role)) {
201          exception("The expression "+constraintDef +
202            "contains the variable " + role + " which has not yet
203            been declared as a role",constraintDefT);
204        }
205      }
206
207      return constraint;
208    }
209  }
210  @lexer::header{ package nz.ac.massey.cs.guery.io.dsl; }
211
212  @lexer::members {
213    @Override
214    public void reportError(RecognitionException e) {
215        // Thrower.sneakyThrow(e);
216      }
217  }
218
219
220
221  ID  : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'.')
222  INT : '0'..'9'+;
223  COMMENT
224      :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
225      |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN
226      ;
227
228  NL : '\r'? '\n' ;
229  STRING :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"';
230
231  fragment
232  HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
233
234  fragment
235  ESC_SEQ
236      :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
237      |   UNICODE_ESC
238      |   OCTAL_ESC
```

```
239     ;
240
241 fragment
242 OCTAL_ESC
243     :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
244     |   '\\' ('0'..'7') ('0'..'7')
245     |   '\\' ('0'..'7')
246     ;
247
248 fragment
249 UNICODE_ESC
250     :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
251     ;
252
253 fragment
254 CLASS_NAME
255   :  ID (.ID)*;
256
257 query : declaration? prepare* select+
258   (connected_by|not_connected_by|where)* groupby*;
259
260 declaration   : 'motif' n=ID{motif.setName(n.getText());} NL;
261
262 select  : 'select' vertex=ID{addVertexRole(vertex);} (','
263   vertex=ID{addVertexRole(vertex);})* NL?;
264
265
266 path_length_constraint returns [PathLengthConstraint c]
267   @init { $c = new PathLengthConstraint(); }
268   : '[' min=INT {$c.setMinLength(intValue(min));} ',' max=(INT | '*')
269   {$c.setMaxLength(max.getText().equals("*")?-1:intValue(max));} ']';
270
271 find_all returns [boolean value]
272   @init {$value = false;}
273   : v='find all'?{$value = v!=null;};
274
275 prepare : 'prepare with' scriptClass=ID {addProcessor(scriptClass);} NL
276
277 connected_by_single : path=ID '(' from=ID '>' to=ID ')'
278 lengthConstraint=path_length_constraint?  all=find_all {addPathRole(pat
```

```
279     all.value,false);};
280 not_connected_by_single : path=ID '(' from=ID '>' to=ID ')'
281 lengthConstraint=path_length_constraint?  all=find_all {addPathRole
282     all.value,true);};
283
284 connected_by : 'connected by' connected_by_single
285   ('and' connected_by_single)* NL?;
286 not_connected_by : 'not connected by' not_connected_by_single
287   ('and' not_connected_by_single)* NL?;
288
289 constraint : expr=STRING{addConstraint(expr);} NL?;
290
291 where : 'where' constraint ('and' constraint)* NL?;
292
293 aggregation_clause : expr=STRING{addGroupBy(expr);} NL?;
294
295 groupby : 'group by' aggregation_clause
296   ('and' aggregation_clause)* NL?;
```

Listing 1: The ANTLR grammar file for GUERY queries.

.2 Appendix HGR

To run HGR you need an Amazon AWS account. Sign up for an account at http://aws.amazon.com.

You can download the source of HGR and the scripts used to run it on Ubuntu from https://s3.amazonaws.com/msc-data/hgr-code/hgr-source.zip.

The raw bench mark results are available from https://s3.amazonaws.com/msc-data/hgr-code/benchmark-results.zip.

There are really two parts to the HGR source, the jar that is uploaded to Amazon that contains the code to run the analysis, and the benchmarking tool that handles cluster management and automating benchmark runs.

HGR does not use a build manager such as Maven or Ant, though it really should. There are also a number of hard coded links in the Java source and scripts. Most of these are related to Amazon credential locations and S3 buckets.

HGR has not been run from a Windows machine.

.3 Appendix DGPROC

DGPROC is not complete, and as such there are no benchmarks that can be run. The source is available from `https://github.com/ajesler/dgproc`.

You will need to have RabbitMQ set up on the machine you want to be the host. This can be done by following the instructions on the RabbitMQ website[1]. The instructions are very good, and it should only take 10-15 minutes to get it running.

In order to run the program, you need to ensure that the property files (*.properties in java/src) are updated with your details. Then compile the jars using the included ANT script. Once the RabbitMQ host is started, the jars can be run from a command line.

Setting up the coordinator and workers is not very complicated - just a single command line to start. No installation necessary.

[1] http://www.rabbitmq.com/

Bibliography

[1] Amazon. http://www.amazon.com/.

[2] Amazon elastic cloud compute. http://aws.amazon.com/ec2/.

[3] Amazon elastic mapreduce. http://aws.amazon.com/elasticmapreduce/.

[4] Amazon simple storage service. http://aws.amazon.com/s3/.

[5] Amazon sqs. http://aws.amazon.com/sqs.

[6] Amqp. http://www.amqp.org/.

[7] Apache activemq. http://activemq.apache.org/.

[8] Apache hama. http://incubator.apache.org/hama.

[9] Apache qpid. http://qpid.apache.org/.

[10] Cassandra. http://cassandra.apache.org/.

[11] Corba. http://www.omg.org/spec/CORBA/Current/.

[12] Cypher. http://docs.neo4j.org/chunked/1.7/cypher-query-lang.html.

[13] Giraph. http://incubator.apache.org/giraph/.

[14] Goldenorb. http://goldenorbos.org/.

[15] Google protocol buffers. http://code.google.com/apis/protocolbuffers/docs/overview.html.

[16] Graphml. http://graphml.graphdrawing.org/.

[17] Gremlin. https://github.com/tinkerpop/gremlin/wiki.

[18] Guery framework. http://code.google.com/p/gueryframework/.

[19] Hadoop. http://hadoop.apache.org/.

[20] Infinitegraph. http://objectivity.com/products/infinitegraph/overview.

[21] Java message service. http://www.oracle.com/technetwork/java/jms/index.html.

[22] Jclouds. http://www.jclouds.org/.

[23] Json. http://www.json.org/.

[24] Jsr 914 - java message service (jms) api. http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html.

[25] Log4j. http://logging.apache.org/log4j/index.html.

[26] mfinder. http://www.weizmann.ac.il/mcb/UriAlon/groupNetworkMotifSW.html.

[27] Neo4j. http://www.neo4j.org/.

[28] Objectivity db. http://objectivity.com/products/objectivitydb/overview.

[29] Pagerank. http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=6,285,999.PN.&OS=PN/6,285,999&RS=PN/6,285,999.

[30] Qualitas corpus website. http://www.qualitascorpus.com/.

[31] Rabbitmq. http://www.rabbitmq.com/.

[32] Rackspace. http://www.rackspace.com/.

[33] Storm. https://github.com/nathanmarz/storm.

[34] Terracotta. http://www.terracotta.org/.

[35] Zeromq. http://www.zeromq.org/.

[36] Gene Amdahl. *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, volume 30, pages 483–485. AFIPS Press, 1967.

[37] Timothy L. Bailey, Mikael Boden, Fabian A. Buske, Martin Frith, Charles E. Grant, Luca Clementi, Jingyuan Ren, Wilfred W. Li, and William S. Noble. Meme suite: tools for motif discovery and searching. *Nucleic Acids Research*, 37(suppl 2):W202–W208, 2009.

[38] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. *Fault-Tolerance in the Borealis Distributed Stream Processing System*, page 13. ACM Press, 2005.

[39] R Banos, C Gil, J Ortega, and F G Montoya. Parallel heuristic search in multilevel graph partitioning, 2004.

[40] D Beyer and C Lewerentz. Crocopat: efficient pattern analysis in object-oriented programs, 2003.

[41] Dirk Beyer. Relational programming with crocopat. *International Conference on Software Engineering*, 86(2):807–810, 2006.

[42] Paolo Castagna. Graph algorithms with mapreduce. *Distributed Computing*, 2007:1–16, 2008.

[43] Rishan Chen, Xuetian Weng, Bingsheng He, Mao Yang, Byron Choi, and Xiaoming Li. On the efficiency and programmability of large graph processing in the cloud. (MSR-TR-2010-44), 2010.

[44] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29–41, 2009.

[45] Stephen A Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing STOC 71*, 22(1):151–158, 1971.

[46] Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Generalized louvain method for community detection in large networks. *2011 11th International Conference on Intelligent Systems Design and Applications*, page 6, 2011.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[48] J. Dietrich and C. McCartin. Scalable motif detection and aggregation. In R. Zhang and Y. Zhang, editors, *Australasian Database Conference (ADC 2012)*, volume 124 of *CRPIT*, pages 31–40, Melbourne, Australia, 2012. ACS.

[49] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M Ali Shah. On the detection of high-impact refactoring opportunities in programs. *CoRR*, pages 1–14, 2010.

[50] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[51] Christos Faloutsos. Pegasus : A peta-scale graph mining system - implementation and observations. *World Wide Web Internet And Web Information Systems*, 31132:229–238, 2009.

[52] E Gamma, R Helm, R Johnson, and J Vlissides. *Design Patterns: elements of reusable object-oriented software. 1995*, volume 395. Addison-Wesley Publishing Co., 1995.

[53] M Girvan and M E J Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821–7826, 2002.

[54] D. Benjamin Gordon, Lena Nekludova, Scott McCallum, and Ernest Fraenkel. Tamo: a flexible, object-oriented framework for analyzing transcriptional regulation using dna-sequence motifs. *Bioinformatics*, 21(14):3164–3165.

[55] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars : A mapreduce framework on graphics processors. *October*, 67(2):260–269, 2008.

[56] B Hendrickson and R Leland. A multi-level algorithm for partitioning graphs, 1995.

[57] Mohammad Farhan Husain, Pankil Doshi, and Latifur Khan. Storage and retrieval of large rdf graph using hadoop and mapreduce. *Cloud Computing*, 5931:680–686, 2009.

[58] H V Jagadish. A compression technique to materialise transitive closure. *ACM Trans Database Syst*, 15(4):558–598, 1990.

[59] G Karypis and V Kumar. Analysis of multilevel graph partitioning, 1995.

[60] B W Kernighan and S Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[61] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[62] S Kumar, S K Das, and R Biswas. Graph partitioning for parallel applications in heterogeneous grid environments, 2002.

[63] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. *Genome*, pages 78–85, 2010.

[64] Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel : A system for large-scale graph processing. *Proceedings of the 2010 international conference on Management of data*, pages 135–145, 2010.

[65] Andrew Maloney and Andrzej Goscinski. A survey and review of the current state of rollback-recovery for cluster systems. *Concurrency and Computation: Practice and Experience*, 21(12):1632–1666, 2009.

[66] Peter Mell and Timothy Grance. The nist definition of cloud computing. *Nist Special Publication*, 145(6):7, 2011.

[67] R Milo, S Shen-Orr, S Itzkovitz, N Kashtan, D Chklovskii, and U Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–7, 2002.

[68] Jaeseok Myung, Jongheum Yeon, and Sang goo Lee. Sparql basic graph pattern processing with iterative mapreduce. *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud MDAC 10*, pages 1–6, 2010.

[69] M.E.J. Newman. *Networks: an introduction*. Oxford University Press, 2010.

[70] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *World Wide Web Internet And Web Information Systems*, 54(2):1–17, 1998.

[71] J T Park, M Li, K Nakayama, T L Mao, B Davidson, Z Zhang, R J Kurman, C G Eberhart, M Shih Ie, and T L Wang. Fanmod: a tool for fast network motif detection. *Cancer Research*, 66(9):6312–6318, 2006.

[72] Eric Prudhommeaux and Andy Seaborne. Sparql query language for rdf. *W3C Recommendation*, 2009(January):1–106, 2008.

[73] Padmashree Ravindra, Vikas V Deshpande, and Kemafor Anyanwu. Towards scalable rdf graph analytics on mapreduce. *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud MDAC 10*, pages 1–6, 2010.

[74] A.J. Riel. *Object-oriented design heuristics*. Addison-Wesley Pub. Co., 1996.

[75] David P Rodgers. Improvements in multiprocessor system design. *SIGARCH Comput Archit News*, 13(3):225–231, 1985.

[76] Kassem Saleh, Imtiaz Ahmad, Khaled Al-Saqabi, and Anjali Agarwal. An efficient recovery procedure for fault tolerance in distributed systems. *Journal of Systems and Software*, 25(1):39–50, 1994.

[77] Peter Sanders and Christian Schulz. Multilevel algorithms for multi-constraint graph partitioning. *Proceedings of the IEEEACM SC98 Conference*, 6942(98-019):1–13, 2010.

[78] Fred B Schneider. *Abstractions for Fault Tolerance in Distributed Systems*, pages 727–734. North-Holland, 1986.

[79] Falk Schreiber and Henning Schwöbbermeyer. Mavisto: a tool for the exploration of network motifs. *Bioinformatics*, 21(17):3572–3574, 2005.

[80] Ken Schwaber. *Agile project management with Scrum*, volume 7. Microsoft Press, 2004.

[81] Alan Soper, Christopher Walshaw, and Mark Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.

[82] W P Stevens, G J Myers, and L L Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[83] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[84] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, 2010.

[85] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[86] Shengqi Yang, Bai Wang, Haizhou Zhao, and Bin Wu. Efficient dense structure mining using mapreduce. *2009 IEEE International Conference on Data Mining Workshops*, (v):332–337, 2009.