Contents

Chapter 1

Introduction

With the growing amount of generated data, the size of files is increasing as well. To deal with this problem efficient loading and importing of these files is an important topic to deal with. Especially in times of bigdata where file sizes of serval gigabytes are common and each second of high-performance computing is very valuable. In many cases algorithms can't proceed with an incomplete dataset, which means that everything else has to wait on the loading task.
The data is often stored in graph like networks, where data is represented by vertices and edges. These graphs can consist of billions of small objects, in which case the metadata-overhead must be very small to work efficiently. There are many kinds of various graph file formats but this thesis will only focus on the most common and relevant ones, which serve the purpose of exchange the information between two systems.

This works tries to resolve the problems, which comes with trying to use the resources of a distributed system to accelerate the loading of graphs into memory and manage the chunk distribution in the network. To accomplish this the file format must be parallelizable. This needs to be determined first. The graph files formats, this work deals with, will be evaluated to their parallelizability, their relevance and current state of use. Over the course of time various graph file formats got introduced with many different properties and purposes. They vary from simple edge lists to complex formats in binary or xml format.

The current state-of-the-art standard is that a parallelizable file format gets used and mapped on a map-reduce framework, often simple formats are used, like connection list, where the overhead to deal with is very small and one line describes exactly one node and its edges. It's common practice that complex formats get parsed into simpler ones and then read in. This work will try to accomplish a map-reduce-like approach with many more formats, while trying to preserve the performance gained by the resources of the distributed system.

Chapter 2

Preliminaries

2.1 Graph File Formats

A graph file format is a specification for describing a graph in a predefined syntax, which then can be read in by an application to access the data. Over the course of time many graph file formats have been established. Most of these formats were developed, with a special use cases in mind. For example GraphML was designed to support as many features as possible for graph drawing [**?**]. In the recent time the trend to invent new graph formats is decreasing. It can be seen in [**?**] that most of the formats, which get listed in their work, aren't developed any further, also there is an significant reduction of xml type formats in the last couple of years and the number of json graph file formats is rising. Based on the collection provided in [**?**], a table with formats was created to identify their up-to-dateness.

Id	**Graph File Format**	**Reference Time Frame**	**Structure**
1	GraphML	2000 - present	XML
2	JSON Graph	2014 - present	JSON
3	KrackPlot	1993 - present	simple
5	Ordered Graph Data Language	2002 - present	BNF
6	Open Graph Markup Language	2012 - present	XML
7	Simple Interaction Format	2003 - present	simple
8	Stanford Network Analysis Platform	2005 - present	simple
9	Trival Graph Format	unknow - present	simple

Table 2.1: Selection of up-to-date Graph File Formats based on [**?**] and their specifications.

There are several properties, which an graph file format has to fulfill. For example, not all formats are a good option for big data sets. XML and even JSON formats have an bigger overhead then just a simple edge list [**?**]. This overhead scales with the size of the data and can have an impact on the performance of the loading. Our file formats have to be scalable,

otherwise just small datasets are supported and this results in probably faster loading times on a single peer. Most big data sets are provided in simple formats and resolve around saving space, but keeping the format as simple as possible. Also there file formats that don't provide any clear specifications the development has been stopped. This is a problem for more complex formats, because this could lead to inconsistencies while loading or reading these format files. For more simple formats, which follow a trivial approach, is often no specification needed, because their simplicity makes them self-explanatory.

2.1.1 Division of Formats

To enable distributed loading, the graph file format needs to be split into multiple chunks [**?**], which then can be distributed inside the distributed system. The goal is to gain performance, while parallel processing the data on multiple nodes. To be able to split the file, an section must be identified, so that the information inside this chunks doesn't lose it current context. A chunk without its context, can't be interpreted correctly and will unavoidable lead to an wrong graph. If the context isn't keep-able, then loading the format on an single node is probably the best workaround.
It isn't surprising that most formats are divisible in some way, but the main factor to consider at this point, is performance. All peers must wait for the chunks to be created, thats why is in this step no interpretation of the file should be done. This isn't avoidable for all formats, but for the most simple formats there is a way split files without or minimal reading of the file itself.
To determine if a format is divisibility the structure of the file needs to be specified and analyzed. Most important data inside the file shouldn't be dependent on other sections of the same file. In other words the chunks should contain all needed data for reading. There will be formats that specify information about vertices in multiple sections of the file, but this information needs to be independent. If the following data can only be processed after processing all lines before, then this section can't be split. Sections that can't be split for example are edge-tuples, this sections will be referred to as indivisible sections.
These indivisible sections are often surrounded or closed by separators. An separator could be anything often newline characters, tabulations and semicolons are used, but also tags (XML), brackets (JSON) or even the position itself in binary sequences can be used to divide these sections. Some formats provide meta data, that will help split up files by defining the position of the information and their format.
A format is fully divisible if it could be split at the any separator, this will mostly be true for simple formats.

2.1.2 Simple Formats

Simple formats are often a trivial approach of creating an low level but highly function graph file format. These formats are often self-explanatory, but have no official specification. This leads to various problems, beacuse it isn't clear where the boundaries of this formats are. For example which character set is used, which separator is used or if the format contains

meta data/comments [**?**].
Many of these formats are fully divisible, duo to the fact that they often only contain small indivisible sections, so that these sections can be grouped and stored in chunks. Also leads their simplicity to an plain hierarchy inside the file.

Trivial Graph Formats

As trivial graph file formats(TGF) are referred to a list of formats, that follow an simple implementation, but have no specification. The most common formats are standard approaches like edge lists, adjacency matrices, neighbor lists and path list.

Edge List
The Edge list is the most common and trivial approach , of storing a graph. This format is just an list of all edges of the graph. In most cases one line equals one edge, so lines form an indivisible section. After each line the file could be split. One line is made of two edges, which a separated by an separator. Often tabulations are used here, but also spaces and other characters could be used. This format has a relatively low overhead and scales well.

Binary Edge List
This is an compressed version of the Edge List, where no lines are used instead all edges have an fixed size for example the first 8 bytes indicate the start vertex and the follow 8 bytes the target vertex. This compression leads to the advantage that no separator is used, so there is no overhead, but the human readability is lost.

Adjacency Matrix
The adjacency matrix is an matrix, where each entry represents an edge connection between the according vertices. The vertices are identified via row and column number, but often these numbers are left out, to safe space and the rows and lines read in previously need to be counted. The biggest flaw of this format is that it doesn't scale well for big data sets, duo to the fact that every vertex got an entry for every other vertices. So its size increases squarely to the amount of vertices.

Neighbor List
The Neighbor List is is a combination of an Adjacency Matrix and an Edge List, but it removes the unnecessary entires for each vertex. Each row describes all connections for a vertex, so we end up with one line per vertex. This format scales better then the Edge List, duo to the fact that vertices are identified through lines.

SNAP Format
The SNAP format just makes an simple addition to the edge list, by allowing comments inside the file. These are identified through a hashtag at the beginning of the line.

Edge/Vertex-List with Properties

This Format consists out of two files, one file contains just list of vertices. The other file contains an Weighted Edge List with an weight value for each edge at the end of the line. This format is used by the LDBC-Graphalytics Benchmark. [**?**]

[Graphic]

adjacency matrices	neighbor list	edge list	path list
$A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$	$1,2$ $1,3$ $3,1$	$2,3$ 1	$3,1,2$ $1,3$

Figure 2.1: An example graph with three vertices in different trivial graph formats [**?**]

SIF Format

The SIF Format is a combination of an edge list with a neighbor list. Additionally, the type of connection can be specified with strings. It allows multiple edges between the same nodes, if the connection type varies, otherwise it is specified to ignore duplicates. One downside is that the format allows tabulations or spaces as separators, it is stated that if no tabulations are used in the whole file spaces are used as separators, but because all lines have an equal format, it can be stated, if the first line doesn't contain any tab character, spaces will be used as separators. Based of the fact that SIF is combination of two TGFs, it can be said that SIF is also fully divisable, only one separator is used per file and the lines are indivisible sections between which can be split. [**?**]

KrackPlot 3.0

KrackPlot 3.0 follows for simple formats a more complex syntax then the TGFs. The first line contains the number of nodes specified in the following data. This information helps with various problems, like splitting nodes appropriately into chunks. The next line can contain two options "!nc" or "!nl". The first option specifies that the following data contains no coordinates, the other one declares that no labels will be specified. If labels and/or coordinates are defined they start on the second line until line (node-amount)+1. After the second line or the labels/coordinates an adjacency matrix specifies the connections of the nodes. This format doesn't scale well for big data sets duo to its adjacency matrix, but splitting this format would be rather easy duo to the fact that the number of nodes is known and only two lines need to be read in, to know the whole nature of the file. [**?**]

2.1.3 XML Formats

There are various implementations of graph file formats using XML. The XML format is based around tags, which define the object it is describing. It is a structure descriptive language for hierarchically structured data. Reading its information is as result not line based

rather it is tag based. This format is hard to divide without reading it completely, because the XML format consists of multiple layers. This results in the problem to determine the layer on which the object is located, this problem can only be solved by counting the opened and closed tags or using a flat hierarchy. [**?**, **?**] The XML language provides an much higher overhead as the simple graph file formats, which results in an increased file size and as a consequence increased loading time.

GraphML

GraphML is an XML based graph file format. It consists out of one graph element which can contain unordered node and edge elements. GraphML supports hyperedges and nested networks and much more graph features. This results in the problem of deep hierarchies, which can only be solved by tag counting do determine the layer. GraphML support many kinds of graphs, this variety causes many different cases to consider. GraphML doesn't specify the positions of vertices or edges inside the format, so that chunks could result in an unequal distribution of different objects. This results in extracting information while reading the file, which will result in a much more time consumption. [**?**, **?**]

Open Graph Markup Language

The Open Graph Markup Language is part of the Open Graph Drawing Framework. The OPML got some similarities to GraphML, duo to the fact that both formats resolve around XML. Also, this format implements many graph style and drawing operations. Also the OPML doesn't support nested graphs unlike GraphML, which results in a flatter hierarchy. [**?**]

Resource Description Framework

The Resource Description Framwork/XML Format isn't meant to be an graph format, but is commonly used for smaller graphs. Its defines objects and attributes via namespaces. The main feature of this format is its portability duo its model of three information types, which can be mapped as network. [**?**, **?**]

2.1.4 JSON Graph

JSON Graph is based on the JSON-syntax-specification, which allows this fomrat to be read in by normal JSON parsers. This format defines first an graph object, which contains an array of mutiple other object. JSON Graph specifies that every graph object contains an array of nodes and an array of edges. Edges alaways contain the fields source node and target node. Nodes always contain a unique key, which identifies them. All field are JSON Strings.
To load this format it can be assumed, that our graphs do not contain any meta data. This leads to lower oberhead then the XML formats, but still more overhead then the simple formats. [**?**, **?**]

2.1.5 Other Formats

Ordered Graph Data Language

2.2 Distributed Loading of Graph File Formats

2.2.1 Single-Pass-Step

In a single pass step the input data gets only read from memory once. this leads to the problem, that all data of the file is temporally stored on the peers. And for files, that only consist out of edges this gets tedious, duo to the fact that all edges have to be extracted and then also the edge has to be extracted.

2.2.2 Two-Pass-Step

In a two pass step the input data gets read twice, this leads to fact that IO gets performed to times, but not all edges need to be stored temporarily This suits espically graph formats, which consist out of more then one file, it would be nonsense to load data that isn't needed at that point of time. With two-pass-step, the vertices can be read first an stored and then the edges can be inserted easily.
There are multiple ways of loading a graphs into a distributed system. There are several problems bound to the fact that a distributed system consists out of multiple nodes. This problem is very similar to parallel graph loading except, that additionally synchronization between nodes needed is, but the basics from parallel graph loading can be applied. As mentioned to use all nodes for computing the input files need to be split into chunks, which can be distributed to the peers.
All peers must be able to identify, were a key belongs. This leads to fact that

Chapter 3

Architecture

This chapter will give a brief introduction into DXRAM. Then the schema according to which the graphs are going to be loaded will be explained. After that problems that will occure will be discussed.

3.1 DXRAM

DXRAM is a distrubuted in-memory key/value-store. It is implemented in Java and optimized to manage billions of small data objects. For low latency data access DXRAM keeps 100% of the data in RAM. DXRAM provides a low data overhead, which suits hrapg based application very well. Nodes of DXRAM can take the role of a ”normal” peer or a superpeer. Superpeers are arranged in a chord like ring structure.

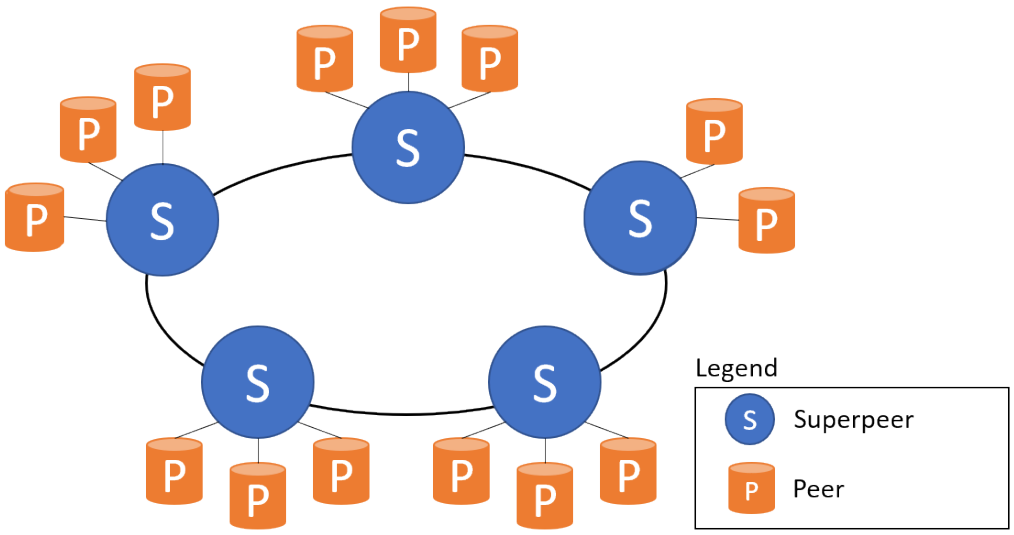

Figure 3.1: Topology of DXRAM

Peers are always assigned to one superpeer. Superpeer take over adminstativ tasks within the distrubuted system while peers serve the role of a storage or backup nodes. DXRAM provides its own application programming interface through services.DXRAM uses a distrubuted file system, so every node can access every file. They can be loaded by applications to get access to storage, computing power and more operations. Custom applications can be loaded as jar-files, but must implement the `AbstractApplication` class and register thereself. All nodes in the system are accessible through the `BootService`, which can list all online peers and theire role inside the system. The key/value-store can be accessed by the `ChunkService`, who provides several operations for creating, getting and storing data structures in chunks. All data structurs, which should be storable, must implement the `AbstractChunk` class.
The computing power of the network can be used in serveral ways. One way is the `JobService` which manages lightweigth jobs. Every job has a unique id and inherits from `AbstractJob`. Every job has its own thread and can run on a local nodes or be pushed to a remote node. Job will be pushed into a job queue and wait for their execution.. Another way to compute something is the `MasterSlaveService` enables to run parallel computations on groups of peers. In this group peers can takes different rolee, one peers takes the role of the master, which coordinates the task. The other peers are either workers(slaves) or take no role. One master and several workers make up one computation group. [**?**, **?**]

3.1.1 Integration in DXRAM

There are several ways of integrating foreign code in DXRAM. One goal is to be able to load data via a terminal command, but also to be able to easily integrate the loader as a library. Therefore the main part of the application will be accessible through a custom Job, which can be used as library. To provide terminal access, without the current possibility to execute jobs from the terminal, the job will be wrapped by an DxApp.

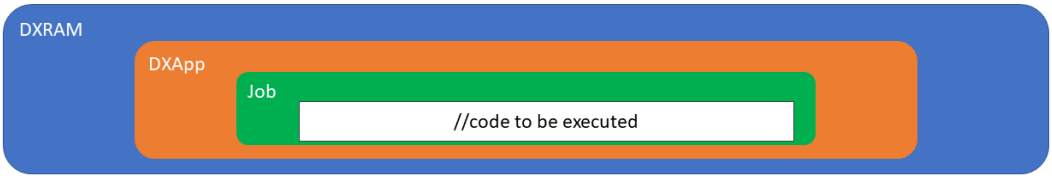

Figure 3.2: Schema of code integeration in DXRAM

One problem that needs to be solving is that at the current state of development, DXRAM isn't able to register jobs on remote peers. This problem occurs to dependency issues, another problem is that jobs can only be registered on the current peer, but local pushing of jobs is by default disabled. To solve this issue The main application invokes the call of a `JobRegistrationApp` Application, which are loaded dynamically at runtime. This helper application takes in a list of jobs (function would be possible as well) and registers them on every peer in the network. This offers this application the possibility to execute remote jobs.

3.2 Architecture of the Application

The architecture of the application consist out of multiple steps. The first steps is to setup our environment, this includes calling the `JobRegistrationApp` to register all needed jobs on all peers and also validating the passed in parameters. Some parameters are required like the file(s) to load and the format, other parameters are optional like the amount of peers or an list of peers to run this application on, by default this application will try to use every peer available for maximizing the loading speed.

3.2.1 Deployment of File data on nodes

The deployment can be done in multiple ways, which have to be evaluated. One possibility would splitting the original file into multiple smaller files and access them via the NFS. This approach would be straight forward, but results in one read and write operation of the whole original file size. Also the file would need to be loaded again into memory for extracting its data. This would result in $O(n)$ on the initiating peer and $O(n/chunks)$. Another approach would be to load the data into memory and send chunk structures to the according peers. This reduces the writing operation and another reading operation on the peer, because the data lays already in memory. The last method would be to access the original file via the NFS and extract the chunk directly on the remote peer. This would result in distriubution of the loading of the original file in one run, but could result in the bottleneck of the one storing node with the original file. This would probably be slower duo to the fact that all nodes accessing one file, would cause a random read sequence, which is often slower then sequential reads.

```
1   ChunkService chunkService = getService(Chunkservice.class);
2   String filePath = ''examplefile.txt'';
3   int chunkSize = 4096 * 1028; //in bytes
4   List<short> peers = getService(BootService.class).getOnlinePeers();
5
6   FileChunkCreator chunkCreator = new FileChunkCreator(filepath, chunkSize);
7
8   while(chunkCreator.hasRemaining()){
9       for(short p:peers){
10          FileChunk fileChunk = chunkCreator.getNextChunk();
11          if(fileChunk != null){
12              chunkService.create().create(p, fileChunk);
13              chunkService.put().put(fileChunk);
14          }
15      }
16  }
```

While the chunks are created from the file, the chunks could already be processed on the remote peers, so that the concurrency reduces the loading time significantly.

Efficient Loading of File into Memory

One key ability of the application is that the loading of the file system should be fast. Often the reading/writing from Storage Devices is a bottleneck. The first problem is that most Storage Devices are designed for sequential read and write operations and often support only one operation at a time. This leads to the problem that multiple threads reading the storage device is a waste of performance and won't speed up the input rate of the file. Another problem is that accessing a storage device will cause a context switch. This results in the application stopping, while the kernel executes sensitive functions. This problem is often address with buffering the files and by requesting more data then needed in one step. Buffer sizes are one optimization problem to consider. [5]
The traditional way of accessing the data of a file is a common routine. First the file gets opened after that the data can be read, written in sequential or random order. This method causes many context switches, which force the application to stop, while the kernel executes sensitive functions. That's why in nearly every case IO gets buffered, so the context switches occur less often. [6]
To reduce the amount of context switches to a minimum. This problem is addressed by memory mapping and suits large files where we can reduce the context switches drastically. A virtual memory mapping between the filesystem and the application address space is created. So expansive system calls can be avoided. The setup of this method is more expensive then the setup of file IO. [5,6]**??**

3.2.2 Processing/Parsing of the Data

The processing/ parsing of the data is unique for each format. Some formats like the TGF need a a different parser in contrast to XML or JSON sytle formats. The best option would be to create as many threads as possible to read in the data.

3.2.3 Storing

The extracted vertices, edges and optional metadata needs to be stored. For this task the `ChunkService` will be used to push the created structs into the key/value-store. ne problem of the distrubuted loading is that peers cannot wait on other peers to get information about vertices, because this would result in too much messages and network traffic for large datasets. This still leaves the problem of not synchronized vertex objects on different peers, which cant be written to the key/value-store because their information could be incomplete.

Duplicate node objects

Some formats (especially in edge lists) nodes just get described by their edges. One problem is if two different peers create the same nodes by different edges, so they don't know if other peers already created a node object for specific nodes. As result they create a local node object with the information they got. In that case after reading the file in its entirety some merge of all hash maps and objects would need to be done. This would result in the

problem that one assumption of this work is that one peer can't keep all nodes and edges in memory, because there are too many objects.
Hash-Distribution of the Nodes while reading
As result of the fact of reorganizing the nodes after they have been loaded is very inefficient and doesn't suit our needs, the next approach would be to organize the nodes at loading/parsing-time. One way could be to hash the labels/ids used in the graph. Note that the hashes created have no security constrains and don't need to be unique. This hash could be used to divide the nodes onto the peers. As result every peer would buffer all nodes, that don't belong to its range of hash values, and send the information he gathered to the according peers. This way we don't have to deal with duplicate nodes or merge. One problem of this approach is, that peers could end up with a chunk, that contains no nodes that hash to itself.
Hash-Distribution after reading
The peers could finish reading the nodes of its chunk and create their own versions in their key-value store. After reading and parsing is finished, they could like in first solution send the key-value-objects to the according nodes based on the hashes of their ids. The target peer then merges all versions of that node and stores the copy that has all information, the other duplicates get deleted. This could result in many objects for one node for each peer

3.3 Issues

A requirement is that the file is located inside one of our peer's storage devices and isn't part of our distributed system in any way. Loading files via an internet connection, will not be featured. Some start parameters are indispensable for this type of application, like the absolute file path, the format of the file or rather the parser that should be used and the number of peers to load the file. Frist one peer should deal with the file, to distribute it to our others peers. The best thing would be, if the peer, who deals with file, would be the same peer where the file is located on. Otherwise the file transfer to the according peer would be a waste of time, especially for file sizes of several gigabytes. After the file chunks got deployed on the network, they could be parsed and loaded by the prior assigned peers into our DXRAM key-value store. [4]

First the file data gets loaded into small chunks which can be distrubted tFirst the file gets partially mapped into the memory, where it can be read by the master slave. The master slave first deals with the format specifications of the file, providing the division schema for splitting the file into chunks. chunks and will be dealing with the file format. Duo to the fact that the data, is in memory, multiple thread can access the data parallel to accelerate processing. The processed data will be collected in chunks which will be deployed to the assigned slaves. The support of infiniband would be increasing the performance drastically, due to spreading the data in the distributed system would be faster. When the chunks arrive on the according slaves, they could immediately start processing their chunks.

Duo to fact that some formats specify data of nodes in two or multiple places inside the file, it would be an interesting approach of using for example two memory maps of the same

file in different sections. So, the chunk (buffer) can be filled with both information at the same time and the node information lie on all on one slave.

3.4 Application Programming Interface

The loader application provides an simple API. To add own formats, the class needs to be added to the `SupportedFormats` class and extend the `GraphFormat` class, which provides information of the required splitting type of the format and, how it should be read.

3.4.1 SupportedFormats

The `SupportedFormats` class, is the lookup table for all formats and which loader and splitter to use. It provides three simple functionalities.

addFormat(String key, Class<? extends GraphFormat> format)
This function adds a format to the supported formats by assigning a key to a class type. For example ”edgellist” is the key for the class `EdgeListFormat`.

getFormat(final String key, final String[] files)
This function returns the class file for a given format key.

isSupported(String key)
This methods returns if a given key is supported.

3.4.2 GraphFormat

To add a custom format, the class needs to extend the `GraphFormat` class. A custom format specifies a Splitter and a Parser.

Chapter 4

Evaluation

Appendix A

Mein Anhang

Klassendiagramme und weitere Anhänge sind hier einzufügen.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Gasterstädt, Sven

Düsseldorf, 15.03.2019