A Distributed Implementation of the Graph Database System DGraph

A Project Report

submitted by

R ASHWIN

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

April 2016

THESIS CERTIFICATE

This is to certify that the report titled **A Distributed Implementation of the Graph Database System DGraph**, submitted by **R Ashwin**, to the Indian Institute of Technology, Madras, for the award of the degree of **B.Tech**, is a bonafide record of the research and development work done by him under our supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Krishna Sivalingam
Research Guide
Professor
Dept. of CSE
IIT-Madras, 600 036

Place: Chennai

Date: 10th May 2016

ACKNOWLEDGEMENTS

I would like to thank Prof. Krishna Sivalingam and Manish R Jain for helping me with the project.

Manish is the author of DGraph and is an ex-Google, ex-Quora engineer. He was of immense help in getting me up to pace, suggesting project ideas and was very responsive to all my queries. He took a generous amount of time every week to guide me through the project.

Prof. Krishna Sivalingam was very helpful in many ways, first of all, letting me take up this project. He gave many suggestions on how the performance has to be analysed and how to verify the correctness of a large-scale project like this.

I also thank Prashanth Koppula and Pawan Rawal, who helped me refine the report and make it a better one.

Finally, I would like to thank my parents, friends and family members who supported me during this time.

With respect,
Ashwin

ABSTRACT

KEYWORDS: DGraph, Graph Database, Scalable, Distributed, Native, Golang

DGraph is a high throughput, low-latency, open-source, native graph database. The aim of this project is to enable centralised DGraph to run in multiple instances so that large datasets which do not fit in one instance can be accommodated by distribution. This requires a renovation in the storage of the data, how the servers interact with each other and how the number of interactions among them can be reduced.

The approach to attain this is by sharding the data and storing the chunks in different servers. This calls for a different way in which the data is loaded, an efficient way in which the servers serve these data chunks and interact with each other. The basic principle used is that if the same type of edges in a graph are stored closer, the amount of interactions among the servers can be reduced, as all graph nodes that are pointed to by a given graph node and a type of edge (relationship) can be obtained in a single seek in that server's disk. The distribution of data is done through modulo sharding, i.e., the modulo of predicate's (edge/relationship name) fingerprint (Hash) is taken with respect to the total number of shards and all the predicates which produce the same modulo are put in a single shard. Once the query reaches a server, it makes network calls to required servers. On testing the distributed version, it was found that the throughput (number of queries served per second) was better and the latency was lesser than the centralised version as the load on the database increased and reached a considerable value. On using an instance with higher computational capacity, the performance of the system increased proportionally.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

XID	External Identification
UID	Unique Integer Identification
GCE	Google Compute Engine
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
HTTP	Hyper Text Transfer Protocol
RDF	Resource Description Framework
SSD	Solid State Drive
RAM	Random Access Memory
DB	Database
JSON	Java Script Object Notation
IOPS	Input/Output Operations Per Second

CHAPTER 1

INTRODUCTION

DGraph[1] is a native, open source graph database which has low-latency and high throughput in serving real-time queries over huge amounts of structured data and it is written in Go language[2]. It uses GraphQL[5] as the query language and responds in JSON.

The storage medium used is RocksDB[3], which is an open source embedded, persistent key-value store. Distribution of data among different instances is managed by DGraph so that there is control over how data is stored, which is an important factor in reducing the number of network calls and hence the latency of queries.

1.1 Overview

Graph data structures store objects and the relationships between them. In these data structures, the relationship is as important as the object. For example, a social network might want to query all the people someone is friends with or things he or she likes.

It is possible to store graph data sets in relational databases, however querying graph data in relational databases is computationally heavy due to the need to do a large number of table joins to find the relationships. Hence, the compute time required increases as the number of results increases, making traditional databases particularly inefficient for large data sets[10].

Graph databases are setup to store the relationships as first class citizens. Accessing those connections is an efficient, constant-time operation that allows you to quickly traverse millions of connections.

1.2 Motivation

Usage of graph databases has grown significantly in a world where "big data" is now a common term. Apart from social networks, applications include user behaviour analysis, e-commerce recommendations, Internet of things, medical and DNA research, search engines, unstructured text mining, machine learning and artificial intelligence.

1.3 Contribution

Before this project, DGraph did not support distributed operation, i.e., It was centralised and ran on only a single instance. Figure 2.1 provides a rough outline of the centralised version. The storage, computational power available in a single instance is limited as there is only so much one machine can support. Thus, it is a requirement of any modern database to be able to scale horizontally, i.e., across different machines. This is achieved by sharding the data, so that different chunks of the data could be served by different servers, hence increasing the total amount of computational resources used cumulatively. I made the changes required to support the distribution of data among the servers by sharding, splitting the loader into two phases, namely UID assigner and data loader, and supporting communication over the network by the servers. This Distributed version was released as a part of the v0.2 release of DGraph and is being run on the website demo[18].

1.4 Organisation

In this report, first, we will get a background on the language, tools and frameworks that are used by DGraph in chapter 2, why they were used compared to other tools or frameworks that are available and discuss some of the components or terminologies involved in DGraph. Chapter 3 will contain the changes made and an example of how the query would be handled by the distributed version. It will be followed by the performance and evaluation study in chapter 4 and then we conclude in chapter 5.

CHAPTER 2

Background and Related Work

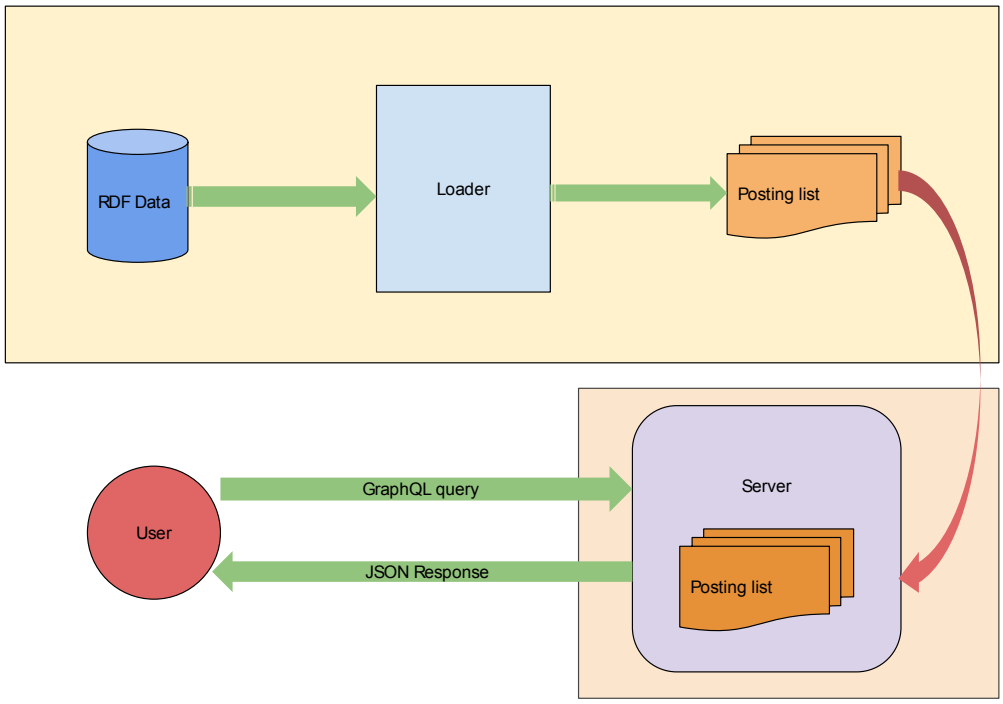

Figure 2.1: A diagram showing the relationship among Loader, RDF Data, Posting lists and user-server interaction. This roughly outlines what happens in the centralised version of DGraph.

Graph databases are optimised for connections, traversing them, rather than aggregation and simple lookups which are the case with relational DB, key-value store respectively.

2.1 Go-lang

Go-lang[2] is an open-source language created by Google. It is compiled, statically typed with garbage collection. One of the most important features of Go is that it has built-in facilities for concurrency. Concurrency refers to not just CPU parallelism

but also letting event-based services like network calls and database reads to be run in an asynchronous fashion. The main concurrency construct in Go is the go-routine, which is a lightweight process (A single core can run hundreds of go-routines). This is highly used in DGraph and one of the main reasons for high performance as independent predicates can be fetched in parallel.

Go-lang enables communication among go-routines through channels which are like linux pipes. Channels are typed so that only messages of a given type can be sent to channels. Using go-routines and channels, we can build constructs like worker pools, network connection pools, background calls with timeouts, fanned-out parallel calls to set of servers and more, all of which are used in DGraph to achieve the highest performance by utilising maximum CPU. We will see how all the described features are used, one by one in the subsequent report.

2.2 RDF

RDF stands for Resource Description Framework. It is a meta-data model and is a family of W3C specifications. DGraph takes the dataset in RDF format as input. It is an effective format to represent entity-relationship or subject-predicate-object relationships. Since graphs are full of such relationships, RDF is the ideal format to represent large graph datasets. A single line in an RDF file would look like this:

<Alice> <friendof> <Bob>.

This represents one relationship and any real world dataset would have millions if not billions of such lines which represent the edges of the graph.

2.3 RocksDB

RocksDB[3] is an embedded, persistent key-value store that stores the keys in a sorted fashion. It is used for fast storage. It is a C++ library that persistently stores keys and values which are arbitrary byte streams. It can fully use the IOPS offered by flash

storage, making it perform fast. It allows range scan, prefix search in keys, batch writes which heavily benefit in better performance of DGraph.

One of the main reasons RocksDB is used in DGraph is that it allows locking over a single key. This is extremely crucial in any database that aims to achieve high throughput as otherwise all the queries that a DB receives will be serialised if a global mutex lock for the entire store is acquired.

2.4 GraphQL

GraphQL[5] is a query language and execution engine that was developed by engineers at Facebook. To put it simply, it allows one to specify exactly the fields that are required in response. It has other features like limiting the result set, sorting based on some attribute. DGraph has its own parser which converts the GraphQL query into the internal representation. The support for GraphQL features is still limited in DGraph and the list of supported features can be viewed in the product road-map[17].

2.5 Flatbuffer

Flatbuffer[4] is an efficient cross-platform serialisation library. It was created by Google for performance critical applications. Since DGraph is being designed ground-up for extreme performance, it uses Flatbuffer for all the communication among the servers.

Some of the characteristics of Flatbuffer, which make it very useful are: access to serialised data without processing or unpacking, memory efficiency and speed (It just needs the byte array that holds the Flatbuffer object) and strongly typed nature of Faltbuffer.

2.6 Sharding

Sharding refers to the partitioning of a dataset into multiple data chunks based on some mathematical functions which are many and have to be chosen to fit the needs of the system that is being developed. Some examples of sharding are Range-based sharding

where the different ranges of the dataset are mapped to different shards. Another technique is Modulo sharding where the key name is passed to a Hash function to get an integer and this integer's Modulo with the number of shards (say N) is obtained, say it is K, this key would belong to the K^{th} Shard among the N shards. We use modulo sharding in DGraph so that all the edges with the same name belong to the same shard.

2.7 Cayley: A Graph Layer

Cayley[11] is a hybrid document-graph engine. It acts as a graph layer on top of other document databases and has no control over data distribution across machines, snapshotting, fault-tolerance. The performance will not be as good due to these factors as data-distribution is an important aspect in reducing the number of network calls and hence the latency of the query itself.

For DGraph, low latency for query execution is the prime goal. In a distributed system, this equates to minimising the number of network calls. For graph processing systems, doing that is hard. If data distribution across machines is done in a standard key-based sharding way, a single graph query could end up hitting a lot, if not all the machines, when the intermediate/final result set gets large. DGraph tackles this problem by dividing up the triple data (subject S, predicate P, object O) in a way so as to co-locate all the (S, O) for P on the same machine (possibly further sharding it if P is too big). Furthermore, it stores the data in sorted lists (O1 ... Oi), to allow for really cheap list intersections.

This allows keeping the total number of network calls required to process a query, linearly proportional to the number of predicates in the query and not the number of results. Besides, all the entities (S, O) are converted to uint64 numbers because they are a lot more efficient to work on and pass around.

2.8 Posting List

Posting list contains a sorted list of UIDs (unsigned 64-bit integers) for a given entity and attribute. It is stored on RocksDB with [Attribute : entity-UID] as the key. Since

RocksDB stores the keys in a sorted fashion, entities for a given attribute will be stored contiguously, typically in a single instance. Key-value pair that is stored in RocksDB is represented as follows:

$$[Attribute : Entity] -> \ Sorted\ list\ of\ Entity\ IDs\ /\ Value$$

2.9 Commit logs

When a mutation hits a database, it is not directly put into the rocksDB store as this would be expensive if the posting lists are generated too often as they have to be kept sorted. Every mutation is logged into an append-only commit log in the disc. This log is played at certain intervals of time and posting lists are updated accordingly. In case of a system crash, all the acknowledged mutations can be recovered from this log by replaying them.

2.10 Mutation layer

Apart from the commit logs, a mutation layer is also stored in memory over the immutable posting list. This allows us to scan the posting list as if they are sorted, without actually changing the posting lists. There are two types of mutation layers: Replace and append or delete.

A posting list is considered dirty if it has a mutation in memory. These layers are merged periodically to re-generate the immutable version of posting list and written back to the disc on RocksDB. Every time this is done, the max commit time-stamp is also written as this would help us figure out how long to seek back in case it has to be regenerated.

2.11 Network calls

A network call is the slowest among RAM, SSD, disc access. To have a low-latency system, it is important to minimise the number of network calls that are made. Sharding

is done based on predicate and not an entity, so even if there are a large number of entities, the payload of the network call might increase but not the number of network calls. Thus, the number of network calls will be proportional to the number of predicates in the query and not the size of the result that is returned.

2.12 Gotomic map

Gotomic map is provided by a library and allows lock less concurrent access to in-memory hash maps. This improves the performance, as it allows multiple go-routines to access the map which in turn improves the throughput and reduces the latency of queries.

2.13 Fine grained locks

In DGraph, the locks are obtained over a posting list and not over the entire database. Simultaneous access to different posting lists could be made which increases the throughput that can be obtained. This is one of the major differences compared to other databases and the reason RocksDB was chosen as the database store at a single instance level.

CHAPTER 3

Distributed DGraph: Implementation Details

This chapter discusses what changes were made to the previous version of DGraph, how it was made distributed, what aspects did we look at in the process.

3.1 DGraph: Previous version

In the previous version of DGraph, loading the data from RDF to RocksDB was done in a single step, i.e., the entire RDF data would be stored in one RocksDB object and then a single server would serve the request it receives using this object which contains all the relationships in the graph. In the next section, we will look at how this architecture was changed as a part of this project to enable DGraph to run in a distributed manner.

3.2 DGraph: Distributed Version

In this version, major changes were made to enable distribution. The changes were made in the loader and the server. The loader was split into three parts: UID assigner, UID merger and data loader. The server was incorporated with network functions to enable it to communicate with other server instances using RPC. The outline of the distributed version can be seen in Figure 3.1. Let us look at the changes made to different components in some detail.

3.2.1 UID Assigner

UID assigner is responsible for assigning each node in the graph a unique 64-bit integer which acts as a unique ID for all the future references to the node within DGraph servers.

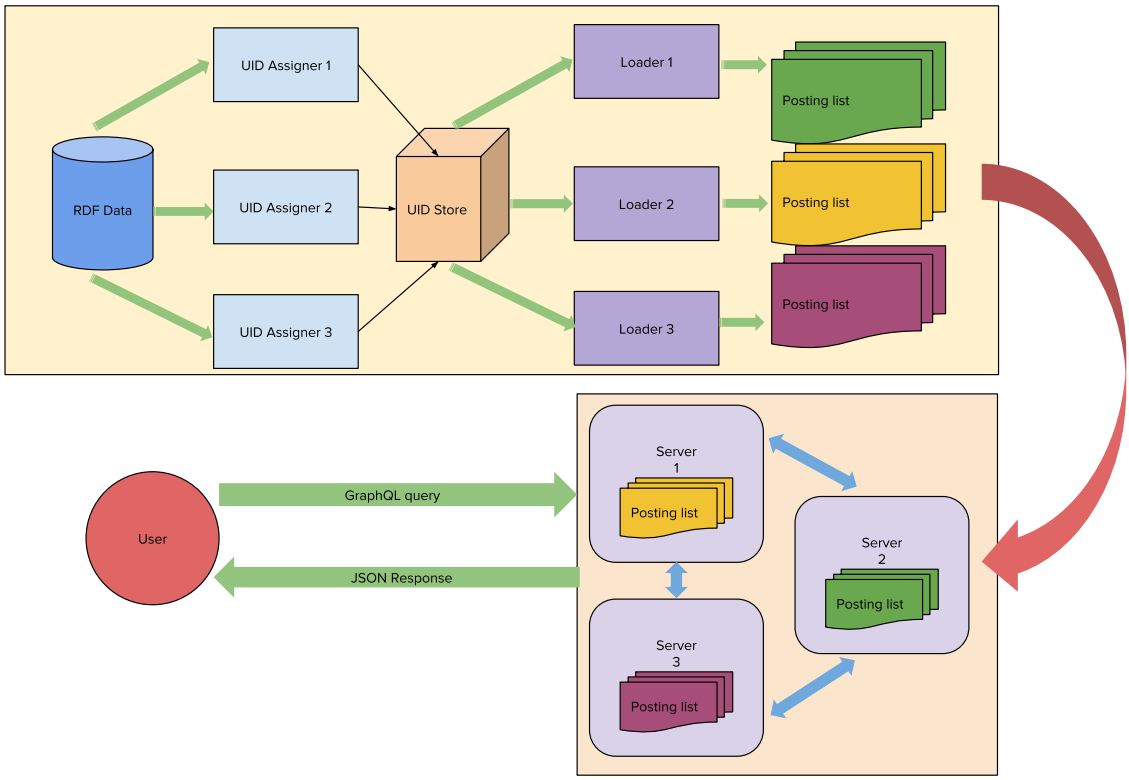

Figure 3.1: This Diagram outlines the steps followed in the Distributed version of DGraph. RDF data is parsed and UIDs are assigned by the UID assigner. Each Loader then converts a part of the data into corresponding posting lists which are then served by multiple servers.

UID assigner takes as input the dataset that needs to be served, total number of shards that are required and ID. It works as follows.

UID assigner reads all the relationships in the RDF data file and extracts the objects (Graph nodes) from it. It then computes the modulo of fingerprint (Integer hash) of that object string and if that value is equivalent to the ID of the instance, it is allotted a unique integer which has not yet been used and is in the allowed range for this given instance. This is done for all the objects in the dataset. Note that this process can be executed on N machines. Each of these machines will generate a RocksDB object which contains the UID for some subset of nodes in the graph.

Before data can be loaded, it is required that we have a global map of UIDs, so they need to be merged to a single RocksDB object.

3.2.2 RocksDB merger

RocksDB merger takes in a list of RocksDB objects and merges them into a single object. Since RocksDB stores all the keys in a sorted manner, we could apply the logic of merge sort and do a K-way merge[8] where K is the number of shards that we generated in the UID assignment phase. This is done using a heap data structure. The complexity of this step would be $N * (logK)$ where K is the number of shards and N is the total number of nodes/objects in the Graph dataset. This method would also help us identify if any duplicate IDs were assigned in the assignment step as two same keys would be consecutive while processing them.

3.2.3 Data Loader

Data loader is the one that loads the relationships into RocksDB object. It works similar to the UID assigner, but the difference is that, instead of taking finger print modulo on object strings, it is done on the predicate string. This ensures that all the predicates end up in the same shard, which is what we want in order to minimise the communication among the servers when queries are processed.

There are N Loaders that are run in N different machines. Each one produces a RocksDB object and represents a shard. These shards are then served by N servers.

This is how data is distributed.

Batch Writes

In the new version, we switched to making batch writes in RocksDB and opening some RocksDB objects which would only be read and not written to as "Read Only". This lead to a significant improvement in the performance of the Loader. Previously, it used to take 6 hours to load the Freebase[7] film data which has 21 million edges, whereas after using batch writes, it only takes 25 minutes for it to complete. Thus, a significant improvement was achieved in the load time.

3.2.4 Server

Server is the one that receives a GraphQL query in the form of a HTTP request. It parses this GraphQL query into a sub-graph object and makes a request to other servers based on the predicates that are required. The decision on which server to connect to is made by finding the modulo of the fingerprint of the predicate, just as it is done during the data loading phase above. We will have a look at what happens when a query is received in the Flow of query section.

Each server creates a pool of TCP connections to every other server in the cluster. This way, the overhead in establishing a new TCP connection by following the handshake protocol for every transfer between the servers is avoided and adds to the efficiency and speed of DGraph.

3.2.5 Novelty of Approach

There are very few graph databases and even fewer that are distributed. There are graph layers which operate on some distributed databas, but that case, since we do not have the control over the data distribution, it leads to more network communications and hence they are not as efficient.

3.3 Flow of Query

Let us now see what happens when a query hits the DGraph servers. First, a HTTP POST request is made by the client which contains the query in GraphQL format. Example :

```
{
    me(xid: Ashwin) {
        Name
        friends {
            followers {
                Name
            }
        }
    }
}
```

The above query requests the name of the node "Ashwin" and the names of all the followers of his friends.

The GraphQL request is received by the server which listens for queries. Once the query is received, it is parsed and a subgraph object is formed using it. Subgraph is a tree like structure which contains the attribute name required at any given level.

```
type SubGraph struct {
        Attr      string
        Children  []*SubGraph

        query   []byte
        result  []byte
}
```

The node from which the query begins is specified in the query. If the id is specified as UID, the processing can start immediately. Otherwise, the XID (external ID) is first converted to UID by sending a request to the machine which serves the UID attribute.

In case of the example query, the starting node is one with XID Alice. First, the

UID of Alice is obtained through a request to server 1. Then the list of her friends is obtained by a network call to server 1 which returns a list of UIDs, which are those of her friends. We want the followers of all these friends, so a request is made to server 2 and a list of list is returned to server 3 which is converted to a list by merging. This way, subgraph object is recursively processed by sending out requests to corresponding machines and receiving the list of UIDs as the result. It has to be noted that all the communication is between the server which receives the query and the server which has the required predicates. Remaining servers do not interact with each other. Once all the required UIDs are obtained in this fashion, the names of the nodes can be obtained. Once this is done, the subgraph is converted to JSON object by parsing it, and this is returned to the user as the response to the HTTP query. Figure 3.2 helps in visualising the flow of query. This is a high-level view of what happens in DGraph.

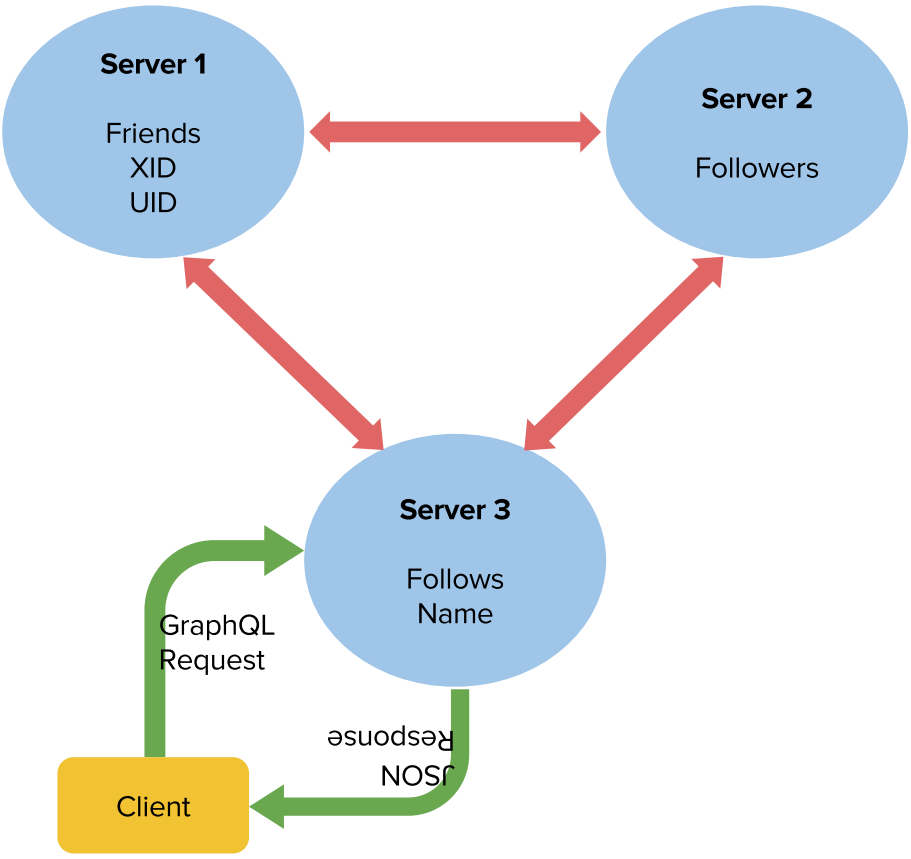

Figure 3.2: A sample topology.

QUERY:

Let us take this GraphQL query as example:

```
{
    me {
        id
        firstName
        lastName
        birthday {
            month
            day
        }
        friends {
            name
        }
    }
}
```

REPRESENTATION:

This would be represented in SubGraph format internally, as such:

```
SubGraph [result uid = me]
    |
  Children
    |
    --> SubGraph [Attr = "xid"]
    --> SubGraph [Attr = "firstName"]
    --> SubGraph [Attr = "lastName"]
    --> SubGraph [Attr = "birthday"]
          |
        Children
          |
          --> SubGraph [Attr = "month"]
          --> SubGraph [Attr = "day"]
```

```
--> SubGraph [Attr = "friends"]
      |
    Children
      |
      --> SubGraph [Attr = "name"]
```

This is a rough and simple algorithm of how to process this SubGraph query and populate the results:

For a given entity, a new SubGraph can be started off with NewGraph(id). Given a SubGraph, is the Query field empty? [Step a]

- If no, run (or send it to server serving the attribute) query and populate result. Iterate over children and copy Result UIDs to child Query UIDs. Set Attribute. Then for each child, use go-routine to run [Step a]. Wait for go-routines to finish. Return errors, if any.

3.4 Consensus : RAFT

RAFT[9] is a consensus algorithm which is equivalent to Paxos in fault-tolerance and performance. Consensus is a fundamental problem in fault-tolerant distributed system. It involves multiple servers agreeing on some values. As long as a majority of servers are fine, the cluster can make progress.

DGraph uses RAFT for two purposes:

1. Maintain a global shard map consistent across the cluster.
2. Replicate the shard among a few machines and keep them consistent to be fault-tolerant. (Implementation in Future)

Overview of RAFT

Raft works by using replicated state machine, i.e., all servers execute the commands in the same order. Consensus is used to maintain this state machine. Hence, as long as majority of the nodes in the cluster are working, the system makes progress. There are three main components in RAFT:

1. Leader election

(a) Heartbeats: These are sent by the master to the followers every few milliseconds.

(b) Randomized timeouts: Timeouts are randomised to prevent multiple nodes starting the election process at the same time.

(c) Majority voting: A leader is elected only if the majority of the nodes in the cluster accept the proposal.

2. Log replication
   (a) Leader receives commands from clients.
   (b) Leader replicates its logs to other servers.

3. Safety
   (a) Only nodes with all committed logs are elected as leaders.
   (b) New leader does not commit entries from previous terms.

The raft library used in DGraph is the one written by the CoreOS etcd group[12].

3.4.1 Global Shard Map

For the distributed database to work, the nodes need to communicate and know which nodes serve which predicates. This requires a globally consistent map that keeps track of this information. When a new node joins the cluster, the information on which predicates it servers is shared among the existing nodes and this node gets the information of the cluster. This happens implicitly when the node joins the RAFT cluster and obtains the map information from the RAFT master at that time and then proposes the values that it serves (which are the predicates it controls).

A peer list has to be maintained by each node which has the information about all the nodes in the cluster. When a node wants to join the cluster, it sends a Hello message to one of the nodes in the cluster and gets the nodeID and IP of the master. It then connects to the master and proposes a configuration change to the cluster. The master tries to get consensus on this proposal and once that is done, the node has joined the cluster and it gets the peer list from the master and makes a connection to all the nodes in the list. It then proposes the change to the shard map, adding the predicates that this node serves. This is spread across all the nodes in the cluster through RAFT.

In case a node goes down, the master tries to connect to it for some time and removes it after a timeout. The shard map has to be updated accordingly. When the node comes

up again it joins the cluster with a new ID if it came up after the time out, else it can join back with the same ID.

3.4.2 Shard Replication

Each shard would be replicated across multiple machines so that some amount of failure can be handled by the cluster. Each predicate would be served by few machines and they would form a RAFT group among themselves. Having heart beats for each group would be very costly as the number of predicates is very large. Hence, we condense the heartbeats of different RAFT groups running on the same machine before transferring it. This is called Muli-RAFT where messages from different groups are combined into one before transferring across the network. The challenge lies in grouping the heartbeats into one, while not letting it affect the timeout of different raft groups and other factors. Hence, it is something that we would like to tackle in future.

CHAPTER 4

Performance Evaluation and Results

In this chapter we will look at the data that was used for testing, the performance of the centralised version and the distributed version of the database on a range of queries and how does varying the computational resources available to the database affect its performance.

4.1 Freebase Film Data

Freebase[14] is an online collection of structured data which includes contributions from many sources including individual and user-generated contributions. Freebase data is available under Creative Commons Attribution License.

Freebase films data[15] is used to test the working and performance of DGraph. It includes information about 478,936 actors, 98,063 directors, films produced worldwide, producers, genres, ratings of the film, countries in which they were made. This is a comprehensive movie database that is publicly available. There are 2 million nodes, which represent directors, actors, films and all the other objects in the database and 21 million edges which are the relationships between actors, films, directors and all the other nodes in the database.

Some examples of the predicates in Freebase film data are :

- type.object.name.en : Connects the object to its English name
- film.actor.film : Connects an actor to the film objects he has acted in

Some example of entries in the dataset are :

- <m.0102j2vq> <film.actor.film> <m.011kyqsq> .
- <m.0102xz6t> <film.performance.film> <m.0kv00q> .
- <m.050llt> <type.object.name> "Aishwarya Rai Bachchan"@hr .
- <m.0bxtg> <type.object.name> "Tom Hanks"@es .

4.2 Performance

In this section, we will see how DGraph performs under various configurations and loads. All the code[16] is available publicly.

4.2.1 Parameters

The parameters that were varied include number of parallel connections to the database(from 1 to 1000 in steps of 100), computational power of the machines [2, 4, 8, 16 cores], number of nodes in the cluster [1, 2, 5]. This would give us an idea of what to expect from the system and help in predicting the configuration required to handle a given load. All the testing was done on GCE instances.

4.2.2 Metrics

The tests were run for 1-minute intervals during which all the parallel connections kept making requests to the database. This was repeated ten times and **Throughput**, **mean latency**, **95th percentile latency**, **50th percentile latency** were measured. For user-facing systems, measuring percentile latency is better than mean latency[13] as the average can be skewed by outliers. Throughput is defined as the "Number of queries served by the server per second and received by the client" and latency is defined as the "Difference between the time when the server received the request and the time it finished processing the request". 95 percentile latency refers to the worst case latency which 95 percentage of users that query the database face. Similarly, 50 percentile latency refers to the worst case latency which half the users that query the database face.

4.2.3 Queries

The queries used to test the performance were about 478,936 Actors and 98,063 Directors. For each request, a query is randomly chosen from these queries.

The query about an actor obtains the list of all the movies that the actor has acted in and looks like this:

```
{
    me(_xid_: XID) {
        type.object.name.en
        film.actor.film {
            film.performance.film {
                type.object.name.en
            }
        }
    }
}
```

The query about a director obtains the genre of all the movies directed by that person and looks like this:

```
{
    me(_xid_: XID) {
        type.object.name.en
        film.director.film  {
            film.film.genre {
                type.object.name.en
            }
        }
    }
}
```

The modulos of the fingerprint of predicates used in the queries are reported in Table 4.1. It can be seen that on using 2 and 5 instance clusters, all the nodes will be utilised while processing the query.

4.2.4 Varying Number of Nodes

In this part, we vary the number of nodes in the cluster to 1, 2 and 5. We measure the metrics specified in the metrics section. Throughput is reported in Table 4.2, Mean latency in Table 4.3, 50 percentile latency in Table 4.4, 95 percentile latency in Table 4.5. The corresponding graphs are in Figure 4.1, Figure 4.2, Figure 4.3, Figure 4.4.

Table 4.1: Modulo of Predicates

Attribute	modulo 2	modulo 4	modulo 5
type.object.name.en	1	1	3
film.actor.film	0	0	4
film.performance.film	1	1	0
film.director.film	0	2	1
film.film.genre	0	0	2

For a given configuration, when we increase the number of connections, the throughput as well as the latency (mean, 50 percentile, 95 percentile) increase. Throughput increases till some point and then flattens out, i.e., ceases increasing. This is the point where the computational capacity is being utilised almost fully. The latency increases almost linearly with the number of connections.

Now, on comparing across the one, two and five node clusters, we can see that the latency (mean, 50 percentile, 95 percentile), as well as the throughput, are better for configurations with higher number of nodes, i.e., when there is more computational capacity at disposal. The throughput increases as we have greater computational power and can handle more queries.

The comparison of the mean, 50th and 95th percentile for 5 node cluster is shown in Figure 4.5. 50th percentile latency is slightly lesser than the mean latency and the 95th percentile latency is greater than the mean latency.

Table 4.2: Throughput comparison on varying the number of nodes

No. of connections	1 node	2 nodes	5 nodes
1	285.83	301.03	298.3
10	463.20	600.16	647.20
50	472.76	644.30	846.30
100	507.06	629.36	959.50
200	523.25	640.75	986.63
300	528.33	663.56	1105.66
400	536.20	648.96	1078.96
500	530.40	674.70	1099.30
600	540.50	680.67	1108.96
700	525.33	657.66	1057.25
800	560.76	683.68	1006.53
900	540.25	679.16	1112.36
1000	560.16	696.50	1072.90

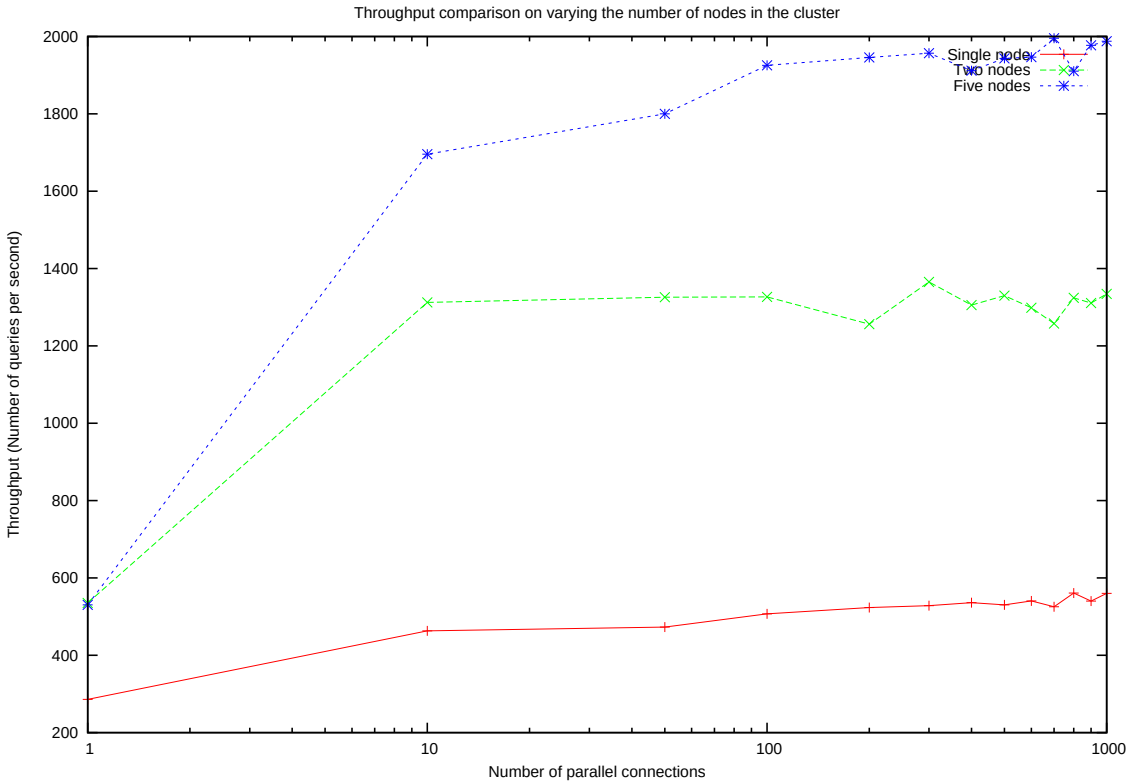

Figure 4.1: Throughput comparison on varying the number of nodes

Table 4.3: Mean Latency comparison on varying the number of nodes

No. of connections	1 node (ms)	2 nodes (ms)	5 nodes (ms)
1	1.77	2.21	3.12
10	14.73	13.50	12.45
50	99.53	71.76	51.65
100	185.16	146.32	95.03
200	361.33	295.66	176.63
300	520.36	412.50	271.33
400	720.25	560.25	380.63
500	986.21	728.59	444.38
600	1080.33	901.66	520.68
700	1250.25	1050.63	626.93
800	1460.30	1157.93	714.55
900	1600.50	1328.50	850.60
1000	1730.52	1468.94	921.18

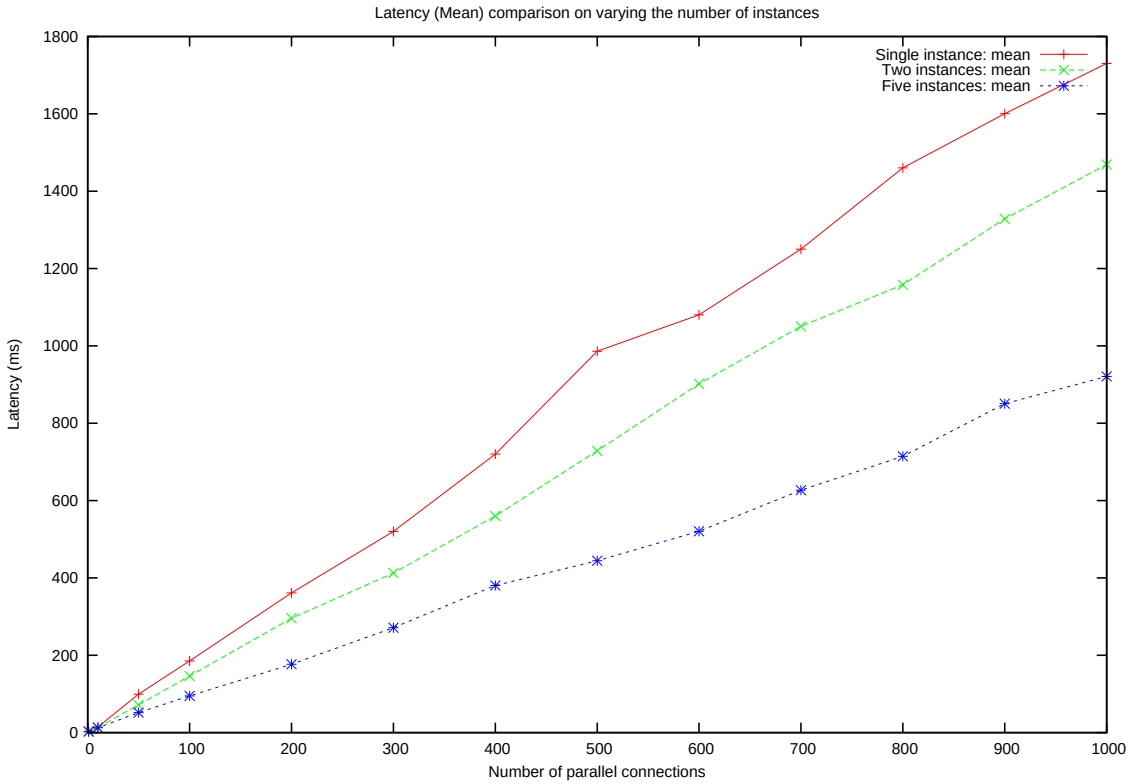

Figure 4.2: Mean Latency comparison on varying the number of nodes

Table 4.4: 50th Percentile Latency comparison on varying the number of nodes

No. of connections	1 node (ms)	2 nodes (ms)	5 nodes (ms)
1	0.99	1.93	2.03
10	9.62	9.41	6.95
50	62.34	52.03	27.97
100	131.31	108.87	55.42
200	250.50	216.33	101.66
300	385.66	334.83	168.93
400	530.75	460.96	232.50
500	696.65	588.79	289.65
600	775.33	731.35	345.66
700	901.50	846.16	415.35
800	1047.13	995.63	461.33
900	1213.76	1190.50	536.93
1000	1327.99	1334.96	590.64

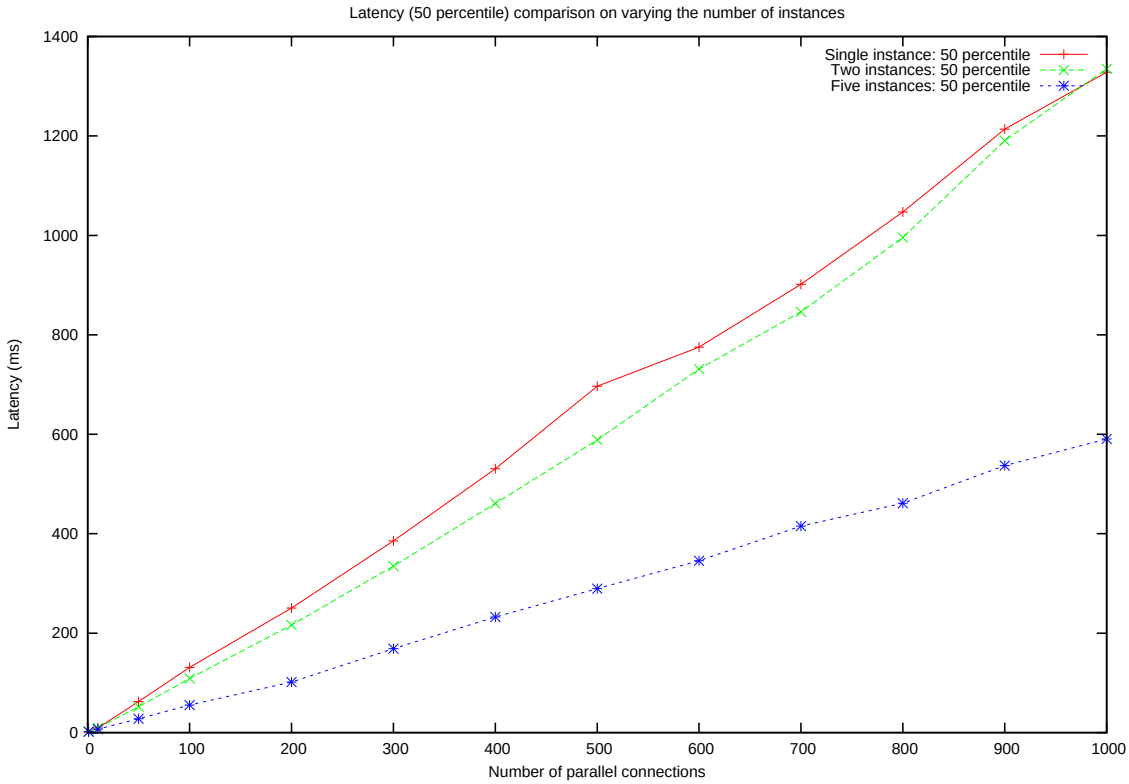

Figure 4.3: 50th Percentile Latency comparison on varying the number of nodes

Table 4.5: 95th Percentile Latency comparison on varying the number of nodes

No. of connections	1 node (ms)	2 nodes (ms)	5 nodes (ms)
1	5.04	2.71	3.91
10	32.13	20.82	30.46
50	186.01	123.63	144.23
100	393.11	473.17	292.48
200	796.33	624.33	463.50
300	1450.15	907.83	742.33
400	2049.33	1131.33	952.35
500	2889.42	1640.34	1419.37
600	3101.50	1910.50	1653.66
700	3521.35	2156.33	1874.63
800	3723.05	2486.93	2018.66
900	4006.55	2758.33	2311.50
1000	4394.97	3038.74	2549.10

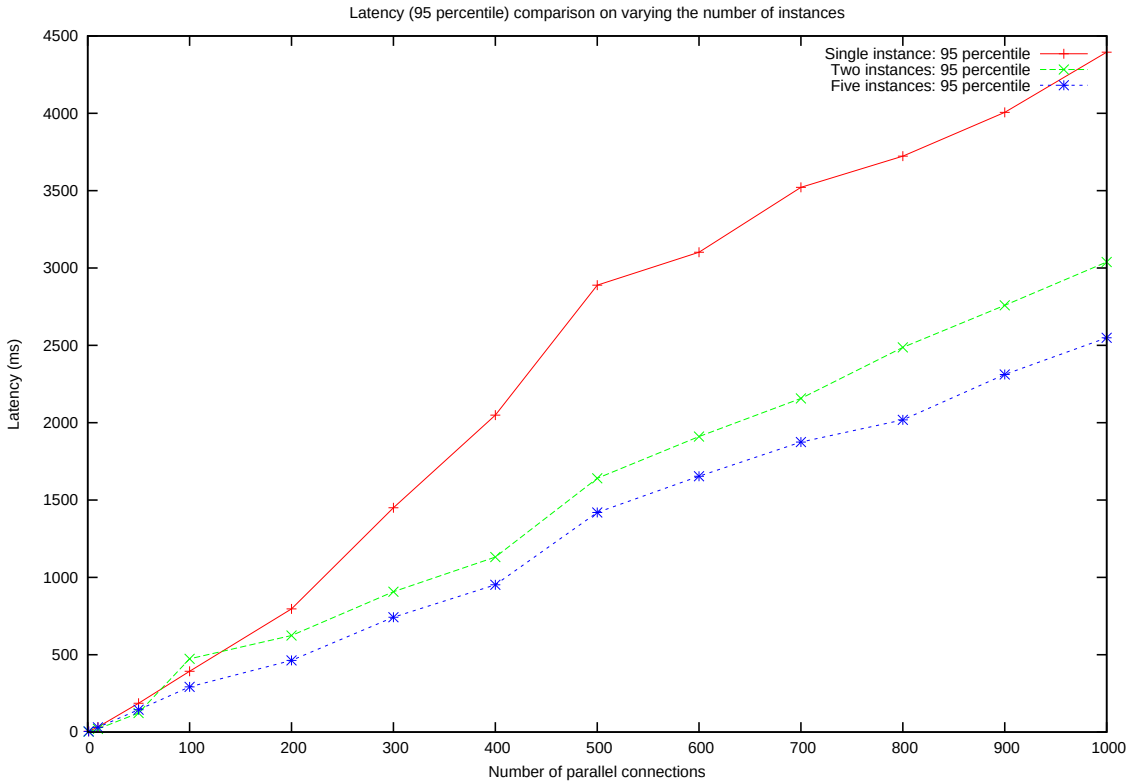

Figure 4.4: 95th Percentile Latency comparison on varying the number of nodes

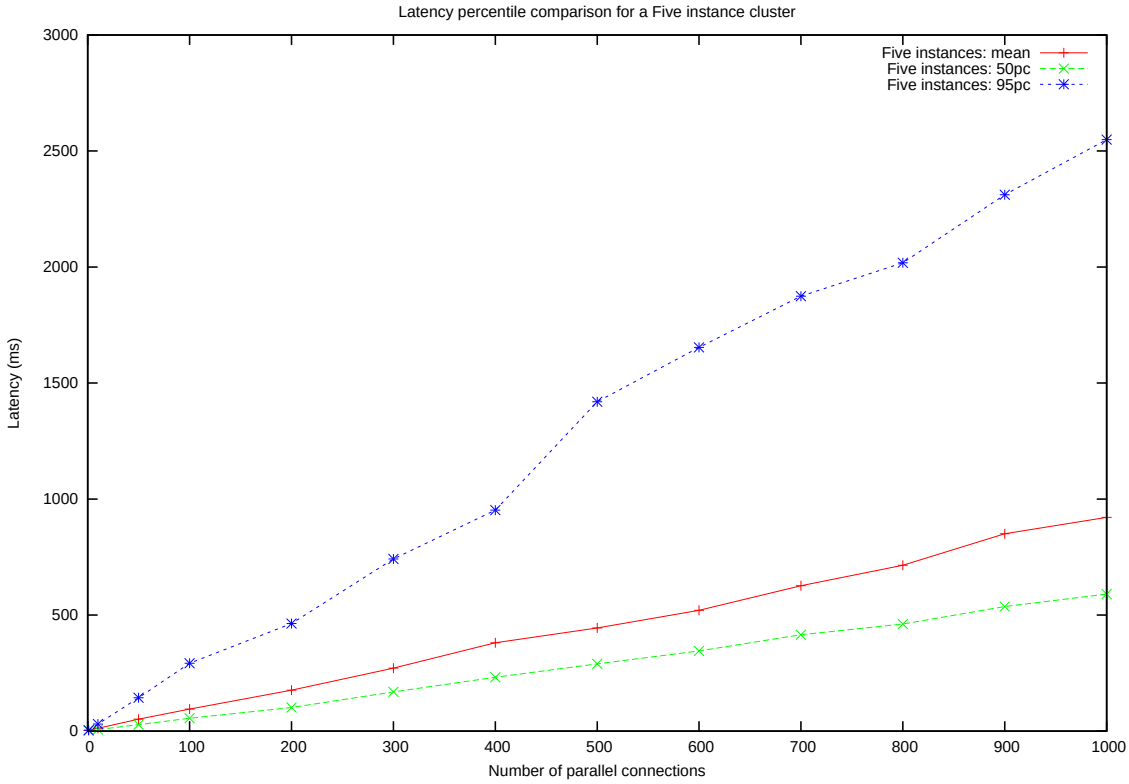

Figure 4.5: Mean, 50th, 95th Percentile Latency comparison on a 5 node cluster

4.2.5 Varying the Computational Power

In this part, we vary the number of cores in an instance to 2, 4, 8 and 16. We measure the metrics specified in the metrics section. Throughput is reported in Table 4.6, Mean latency in Table 4.7, 50 percentile latency in Table 4.8, 95 percentile latency in Table 4.9. The corresponding graphs are in Figure 4.6, Figure 4.7, Figure 4.8, Figure 4.9.

For a node with given number of cores, when we increase the number of connections, the throughput as well as the latency (mean, 50 percentile, 95 percentile) increase. Throughput increases till some point and then flattens out, i.e., ceases increasing. This is the point where the computational capacity is being utilised almost fully. The latency increases almost linearly with the number of connections.

Now, on comparing across two, four, eight and sixteen core nodes, we can see that the latency (mean, 50 percentile, 95 percentile) decreases and the throughput increases. The throughput increases as we have greater computational power at disposal and can handle more queries at the same time.

The comparison of the mean, 50th and 95th percentile for the 16 core machine is shown in Figure 4.10. 50th percentile latency is slightly lesser than the mean latency and the 95th percentile latency is greater than the mean latency.

Table 4.6: Throughput comparison in instances with 2, 4, 8, 16 cores

No. of connections	2 cores	4 cores	8 cores	16 cores
1	285.83	535.43	530.23	623.83
10	463.20	1312.46	1695.76	2254.80
50	472.76	1325.93	1800.03	2320.00
100	507.06	1326.66	1925.53	2327.00
200	523.25	1256.35	1945.66	2380.66
300	528.33	1365.25	1956.93	2355.80
400	536.20	1305.67	1911.65	2269.66
500	530.40	1330.13	1943.40	2311.60
600	540.50	1298.63	1946.83	2295.35
700	525.33	1257.68	1995.36	2431.33
800	560.76	1324.36	1910.54	2455.15
900	540.25	1310.83	1977.25	2411.66
1000	560.16	1334.53	1987.73	2423.70

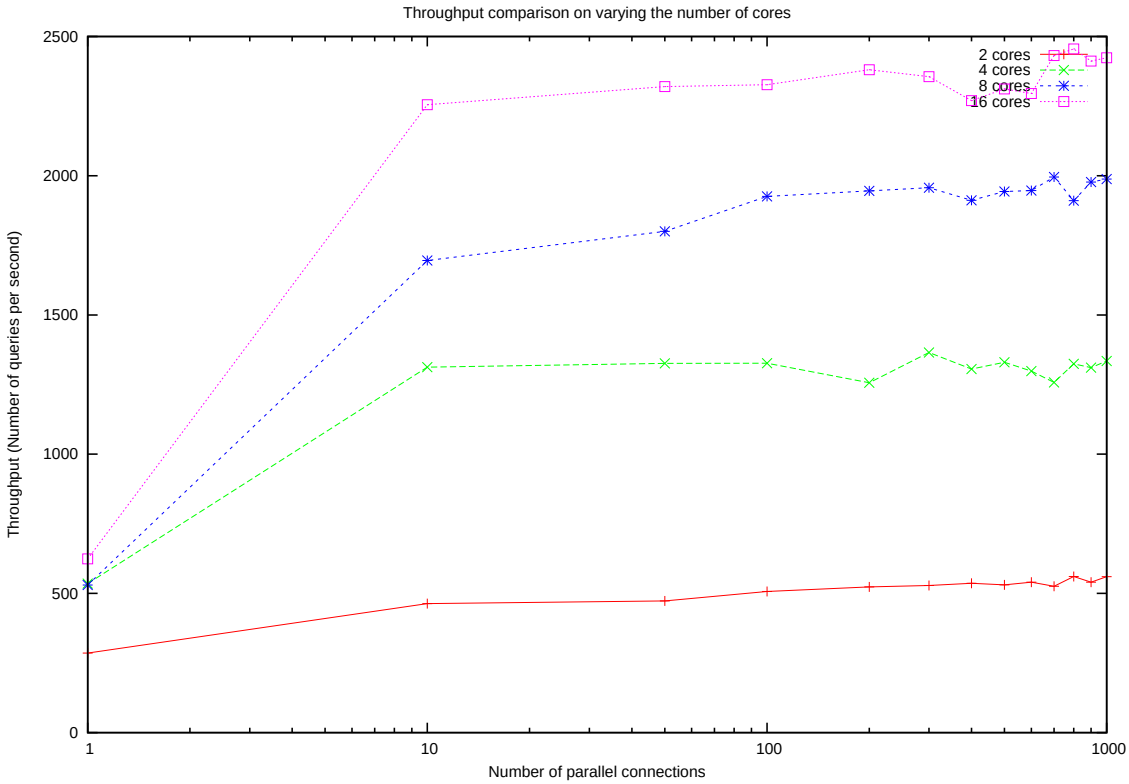

Figure 4.6: Throughput comparison in instances with 2, 4, 8, 16 cores

Table 4.7: Mean Latency comparison in instances with 2, 4, 8, 16 cores

No. of connections	2 cores (ms)	4 cores (ms)	8 cores (ms)	16 cores (ms)
1	1.77	0.82	1.00	0.66
10	14.73	4.77	3.49	2.21
50	99.53	30.28	21.14	15.35
100	185.16	62.54	40.12	25.35
200	361.33	115.66	76.33	54.66
300	520.36	183.50	135.68	76.83
400	720.25	257.67	156.20	105.63
500	986.21	330.42	186.97	141.15
600	1080.33	394.25	221.56	160.55
700	1250.25	460.38	254.66	175.50
800	1460.30	499.66	293.93	183.56
900	1600.50	521.33	325.83	220.86
1000	1730.52	545.53	356.78	285.31

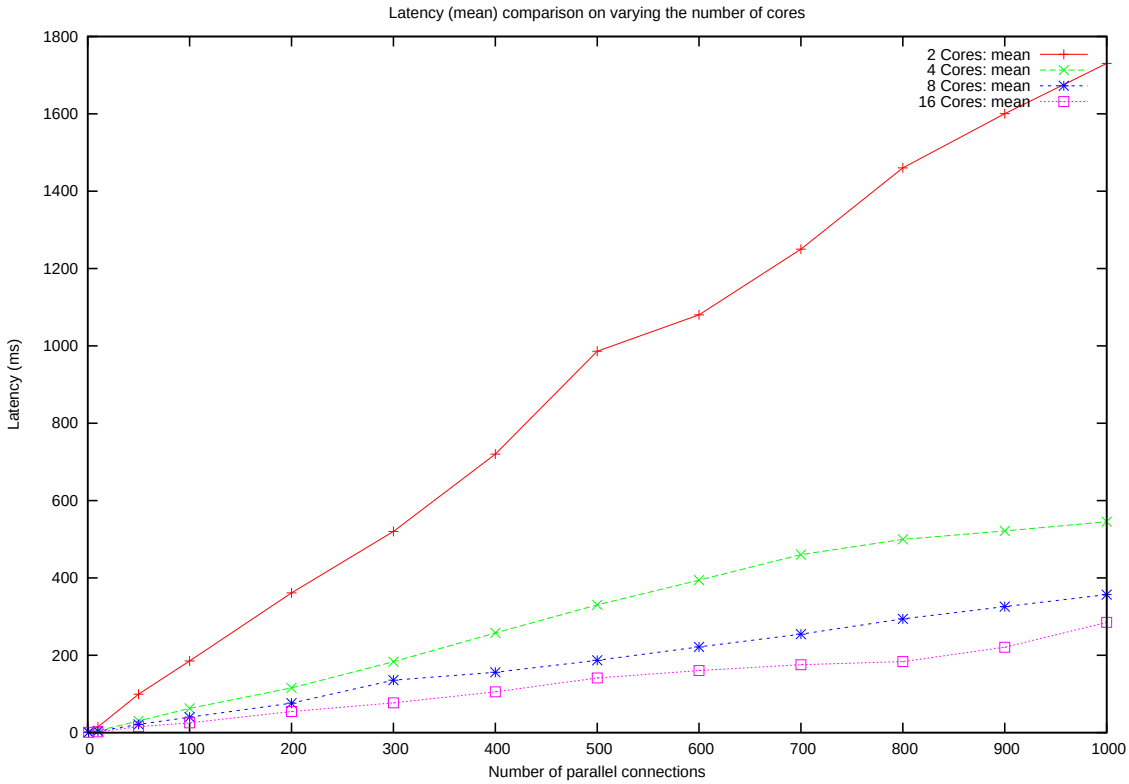

Figure 4.7: Mean Latency comparison in instances with 2, 4, 8, 16 cores

Table 4.8: 50th Percentile Latency comparison in instances with 2, 4, 8, 16 cores

No. of connections	2 cores (ms)	4 cores (ms)	8 cores (ms)	16 cores (ms)
1	0.99	0.74	0.83	0.59
10	9.62	3.54	2.68	1.72
50	62.34	21.58	16.14	11.78
100	131.31	44.83	30.60	24.79
200	250.50	80.50	65.83	53.50
300	385.66	125.66	104.50	70.66
400	530.75	175.63	124.66	101.50
500	696.65	229.74	148.97	115.21
600	775.33	275.83	161.33	150.15
700	901.50	315.25	185.40	170.50
800	1047.13	338.65	243.50	181.66
900	1213.76	365.96	276.93	205.93
1000	1327.99	392.06	315.82	223.52

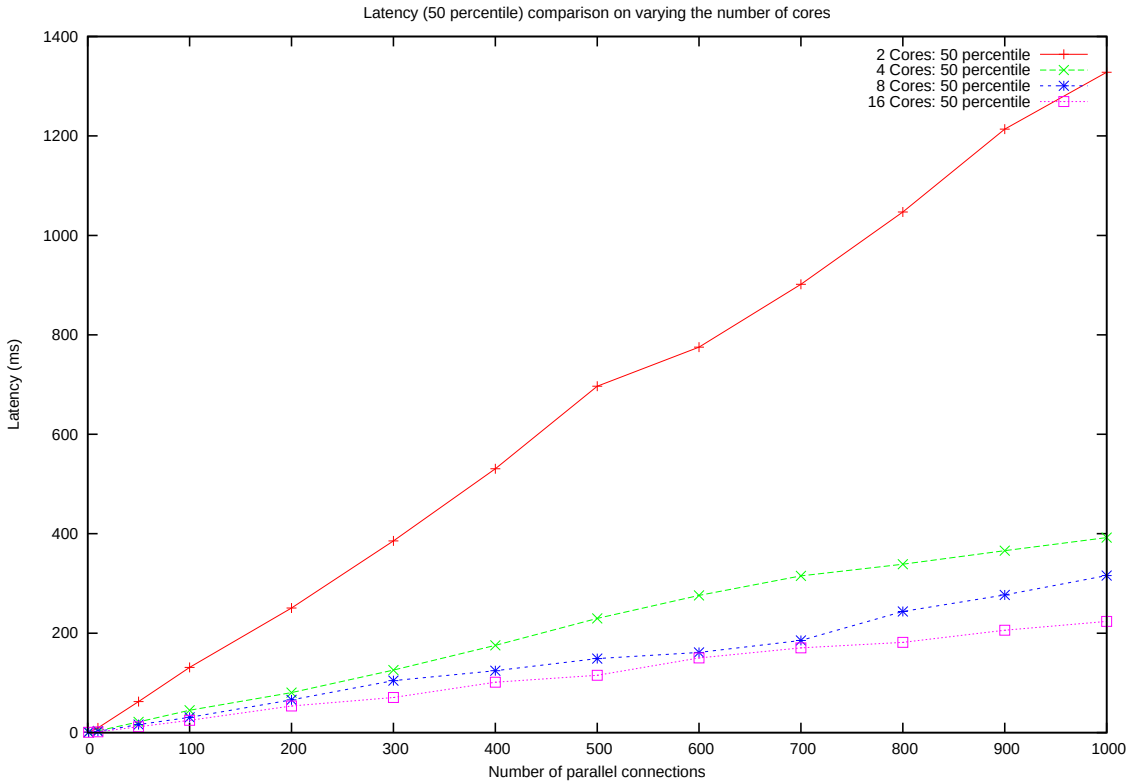

Figure 4.8: 50th Percentile Latency comparison in instances with 2, 4, 8, 16 cores

Table 4.9: 95th Percentile Latency comparison in instances with 2, 4, 8, 16 cores

No. of connections	2 cores (ms)	4 cores (ms)	8 cores (ms)	16 cores (ms)
1	5.04	1.07	1.53	0.87
10	32.13	9.48	6.78	4.25
50	186.01	61.38	42.59	31.37
100	393.11	127.74	85.96	66.50
200	796.33	253.25	147.33	135.65
300	1450.15	380.66	219.63	190.50
400	2049.33	495.63	326.93	250.93
500	2889.42	904.14	485.62	339.16
600	3101.50	910.50	560.50	437.93
700	3521.35	1014.93	602.93	507.80
800	3723.05	1245.83	658.67	584.33
900	4006.55	1438.67	712.63	668.50
1000	4394.97	1613.20	783.20	734.80

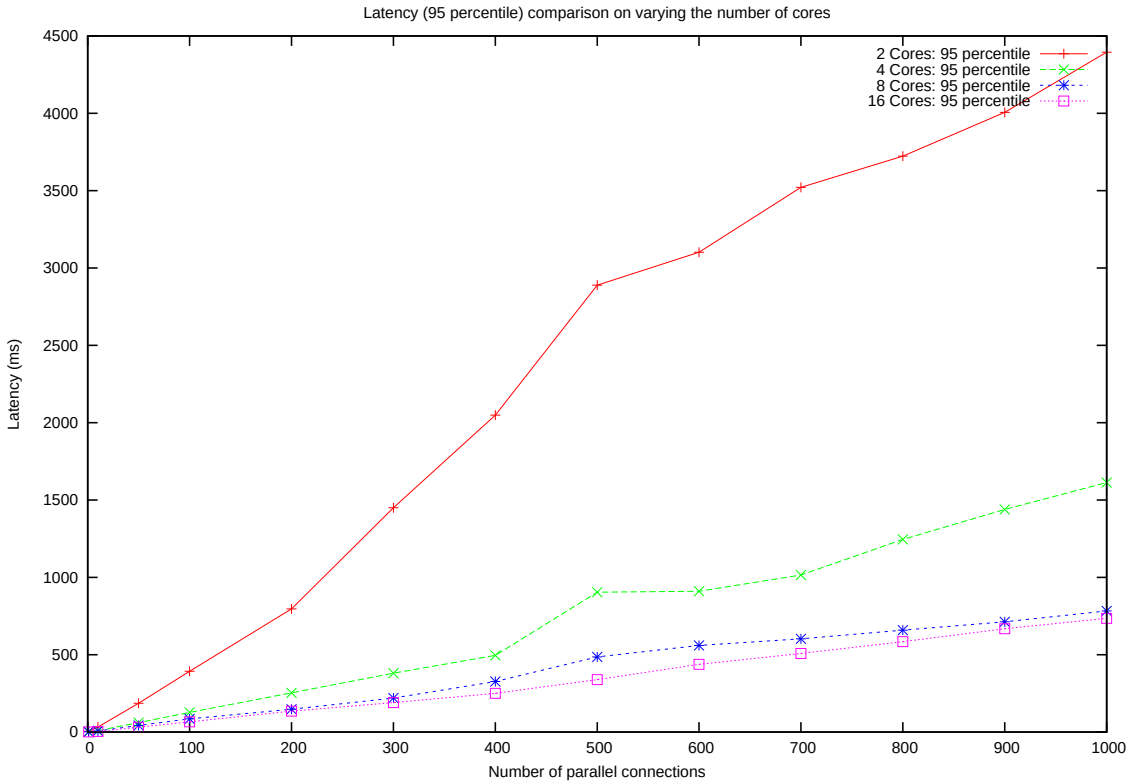

Figure 4.9: 95th Percentile Latency comparison in instances with 2, 4, 8, 16 cores

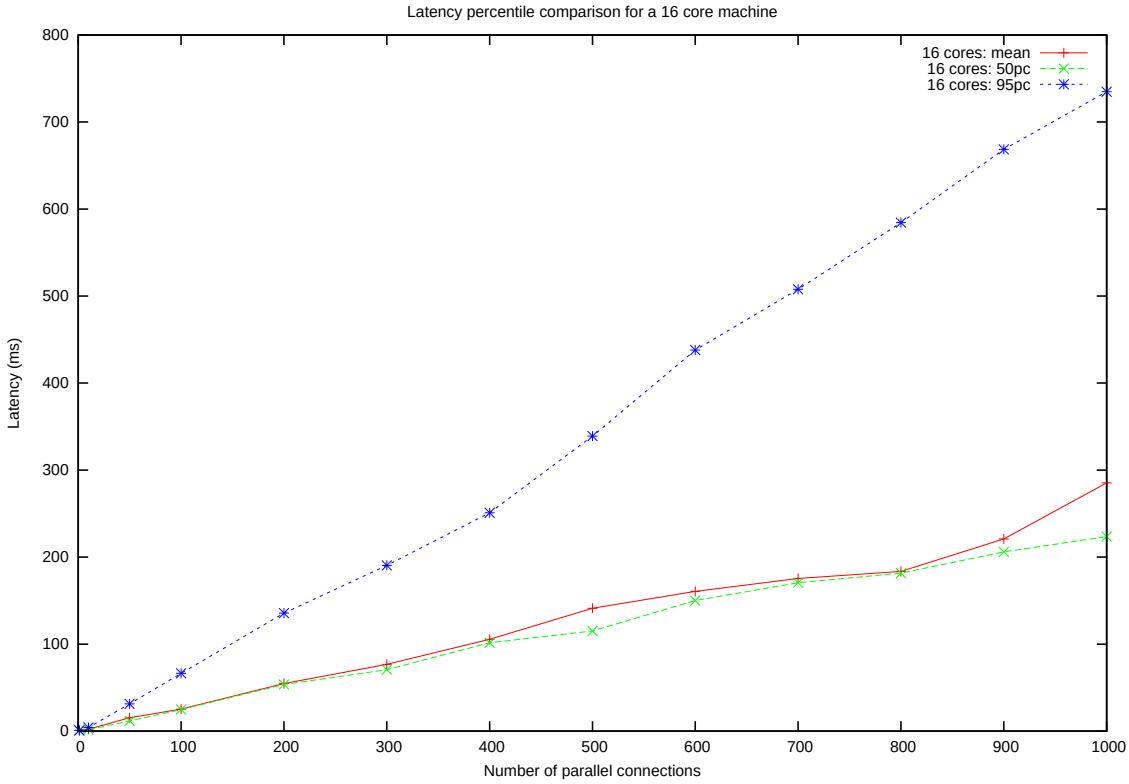

Figure 4.10: Mean, 50th, 95th Percentile Latency comparison in a 16 core machine

4.2.6 Varying the Number of Queries

Let us look at what happens if there is lesser variety in the queries. Suppose, only one query is used, all the queries will block on the same key in the database, the access is serialised and hence, throughput reaches the maximum threshold early. This is validated by the fact that when we increase the number of queries to 25, we see an increase in the throughput and further when the number of queries is increased to larger number [using all actors/directors], it further increased as each query now accesses different key in the store and these can be parallelised and hence can happen concurrently. This is ideal in the real world as a single user will not repeat the same query 1000s of times in a second. There is a spread of users and queries, and DGraph will handle it well.

As the number of simultaneous users increases, the latency that each user faces increases on an average. This is expected as the amount of CPU available for each query reduces on average.

4.2.7 Summary

From the above experiments, we can see a relationship between the throughput, latency and other parameters. The throughput increases as the computational power increases, as the number of nodes in the cluster increases and as the load on the database increases until the point where the computational power is being utilised almost fully. The latency increases as the amount of load on the database increases. This is roughly shown in the following equation:

$$Throughput \propto computational\ capacity * number\ of\ nodes * load/latency$$

$$load: \ Number\ of\ parallel\ connections$$

This is an empirical relationship and roughly shows how throughput varies as the different parameters are changed.

We can see that, as the cumulative computational power that is available increases, the performance of the database is better, which is as expected. There is a limit on

how much computational power a single node can have and once we reach that limit, scaling horizontally is the option. The above-performed experiments prove that scaling horizontally increases the performance. Hence, having more replicas, distributing the dataset optimally across machines are some factors which help in improving the throughput and reducing the latency that the users face.

The RAM usage was not very high in the server process and was 800MB at the maximum during all the experiments when queried.

The recommendation for running DGraph would be:

- Use as many cores as possible
- Have the servers geographically close-by so that network latency is reduced
- Larger sized RAMs are not a necessity in server but are important in case of loader
- Distribute the data among servers and query them in a round-robin fashion for greater throughput

4.3 Processing time

Here we will look at the different components of total time, which are:

1. Server latency: The amount of time the cluster takes to process the query by making network calls to required nodes and creating the appropriate resultant flatbuffer object.
2. JSON conversion latency: Once the flatbuffer is formed, it has to be parsed to create the JSON object that can be returned to the client. This is a time and memory-consuming operation.
3. Parsing latency: The amount of time the server takes to parse the GraphQL query and convert it into a subgraph object.

Let us look at the time when we run a query and see which sub-parts take what amount of times.

Consider this query which tries to retrieve the name of an Actor and the list of all the movies he has acted in.

```
{
```

```
    me(xid: m.0bxtg) {
        type.object.name.en
        film.actor.film {
            film.performance.film {
                type.object.name.en
            }
        }
    }
}
```

On one run, 6.6KB of data is returned with 103 entities in the response. The total server latency is 9.73ms out of which :

- Processing takes up 7.09ms
- JSON conversion takes up 1.17ms
- Parsing takes up 1.47ms

As the number of entities in the result increases the percentage of time consumed in JSON conversion increases proportionally.

CHAPTER 5

Conclusion

The goal of this project was to enable distributed operation of DGraph. DGraph from the ground up is being built to reduce the number of network calls and use as few resources as possible in the distributed version. As we saw in the performance section, the distributed version performs better than the centralised version and this becomes eminent as the amount of load on the database increases. Latency and throughput are better for the distributed version, so, we can infer that as the computational power of the cluster increases, performance gets better. Considering the advantages of the distributed version like horizontal scalability which is the cornerstone of today's modern day databases, it is necessary that we support it and this is a step in that direction.

Further work includes supporting GraphQL features, moving shards across machines, fault tolerance, replication. There is a long way to go before this becomes a production level software and I am sure it will be useful to many people who want to do some interesting projects which involve huge datasets that can be represented as graphs.

REFERENCES

[1] "DGraph code repository" github.com/dgraph-io/dgraph, May 2016.

[2] "Go Language" https://golang.org/, April 2016.

[3] "RocksDB" http://rocksdb.org/, May 2016.

[4] "Flatbuffers While paper" https://google.github.io/flatbuffers/flatbuffers_white_paper.html, May 2016.

[5] He, H., and A. K. Singh. GraphQL: Query language and access methods for graph databases. Technical Report, University of California, Santa Barbara, 2007.

[6] Renzo Angles and Claudio Gutierrez. Survey of graph database models. ACM Comput. Surv. 40, 1, Article 1 (2008). pages : 1:1–1:39

[7] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In Proceedings (SIGMOD '08). 1247-1250.

[8] "K-way Merge sort" http://stackoverflow.com/questions/5055909/algorithm-for-n-way-merge, April 2016.

[9] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm." 2014 USENIX Annual Technical Conference : 305–319.

[10] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A comparison of a graph database and a relational database: a data provenance perspective. In Proceedings, Article 42 , 6 pages.

[11] "Cayley," https://github.com/google/cayley, May 2016.

[12] "etcd RAFT library" https://godoc.org/github.com/coreos/etcd/raft, May 2016.

[13] "Averages can dangerous use percentile," https://www.elastic.co/blog/averages-can-dangerous-use-percentile, May 2016.

[14] "FreeBase Dataset" https://developers.google.com/freebase/#freebase-rdf-dumps, April 2016.

[15] "DGraph Dataset" https://github.com/dgraph-io/benchmarks/tree/master/data, April 2016.

[16] "DGraph Repository" github.com/dgraph-io, May 2016.

[17] "Supported GraphQL features" https://github.com/dgraph-io/dgraph/issues/1, May 2016.

[18] "DGraph website" http://dgraph.io/, May 2016.