LOW LATENCY DATA RETRIEVAL SOLUTIONS FOR BIG DATA

BY

CHAITANYA DATYE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

ABSTRACT

As applications are moving towards peta and exascale data sets, it has become increasingly important to develop more efficient data retrieval and storage mechanisms that will aid in reducing network traffic, server load, as well as minimizing user perceived retrieval delays. We propose an Intelligent Caching technique and a Graph Summarization technique in order to achieve low latency data retrieval for big data based applications.

Our caching approach is developed on top of HDFS to optimize the read latency of HDFS. HDFS is primarily suitable for Write Once Read Many (WORM) applications where the number of reads is significantly more than that of writes. In our Intelligent Caching approach, we analyze real world map reduce traces from Facebook and Yahoo in terms of file size and access pattern distribution. We combine it with the existing analysis from literature to develop a new caching algorithm that builds on top of the HDFS caching API recently released. Based on the findings that a majority of accesses in a map reduce cluster occur within the first 2 hours of file creation, our caching algorithm uses a sliding window approach to ensure that most popular files remain in cache at appropriate time instances. It uses file characteristics for a particular window to determine a file's popularity. File popularity is calculated using file access patterns, file age and workload characteristics. We use a simulator based technique to evaluate our algorithm on various performance metrics by using real world and synthetic traces. We have compared our algorithm with some of the existing variants of LRU/LFU.

Recent rapid growth in real-world social networks has incentivized researchers to explore optimizations which can provide quick insights about the network. Due to this motivation, graph summarization and approximation has been an important research problem. Most of the work in this area has been focused on concise and informative representations of large graph. These large graphs are billion nodes and edges graphs and need a distributed

storage/processing system for any kind of operations on them. Our work primarily focuses on task-based summarization of large graphs that are stored in a distributed fashion and answer queries which are computationally expensive on original graph, but have tolerance with regards to minor errors in exact results. These queries, semantically, provide the same amount of information even with approximate results. Our contribution is a distributed framework which can answer queries probabilistically in a highly efficient way using compact representations of original graph stored in form of summary graphs across a cluster of multiple nodes. These summary graphs are also optimized for space complexity, and only grow in terms of the number of attributes used to answer the query. One can then use a combination of these graphs to answer complex queries in an extremely efficient manner. Our results are promising and show that significant gains in runtime can be achieved using our framework without sacrificing too much on accuracy. In fact, we observe decreasing trend in error as the graph size increases.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my adviser Professor Roy H. Campbell for his invaluable support and advice and for keeping me organized and focussed during my Masters degree. He helped ensure that I had all the resources and guidance I needed to complete my projects.

I would like to thank Read and Jigar who collaborated with me on parts of this project and encouraged me to pursue it further. I would like to acknowledge the guidance provided by Systems Research Group students Cristina, Mayank, Gourav and Shadi and also by my friends Chinmay, Aditya and Naren.

I would like to thank Cinda Heeren and Anna Yershova for their support, friendliness, encouragement and for giving me the wonderful opportunity of being a TA for their courses during my Masters degree.

Finally, I would like to thank my family and friends for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HDFS	Hadoop Distributed File System
WORM	Write Once Read Many
LRU	Least Recently Used
LFU	Least Frequently Used
LLF	Largest File First
ARC	Adaptive Replacement Cache
EQUALGAS	Efficient Query Approximation for Large Graphs using Attributed based Summaries
CDF	Cumulative Distribution Function

CHAPTER 1

INTRODUCTION

As applications are moving towards peta and exascale data sets, it has become increasingly important to develop more efficient data retrieval and storage mechanisms that will aid in reducing network traffic, server load, as well as minimizing user perceived retrieval delays. In this thesis, we aim to solve the problem of minimizing latency incurred in data retrieval using Intelligent Caching and Graph Summarization techniques. Our caching approach primarily focuses on efficient data storage and retrieval based on popularity of data so that popular data items are cached and can be retrieved faster at any particular instant of time. The graph summarization approach deals with providing quick insight into large-scale graph data by using probabilistic estimates obtained from compact representations of original graph.

1.1 Caching

Distributed computing frameworks such as MapReduce [1], Hadoop [2] and Dryad [3], coupled with fault-tolerant distributed data storage like HDFS [4] have become popular for data intensive applications. Such frameworks have been widely deployed in various industries across extensive domains including health care, financial services, energy and utilities, telecoms, social networking and many more [5]. In all these domains, the volume of data that needs to be processed has been growing exponentially causing distributed frameworks to grow increasingly popular as industries attempt to keep up with their data growth. With large volumes of data, there is not only a need for constant data availability but also for minimum delays in data retrieval. Thus, latency has emerged as one of the most important performance metrics when determining if workload constraints can be met.

HDFS is a popular distributed file system that forms the backbone of

Hadoop and all applications built on top of it. It has been designed to be fault-tolerant, highly scalable, robust and is mostly suitable for WORM applications. HDFS is primarily suited for applications that require batch processing rather than those that require interactive usage by users. Historically, the main concentration of HDFS is on data throughput and not on latency. Many of today's applications are interactive and often jobs have time based constraints. Improving latency of HDFS based applications will extend the functionality of HDFS beyond WORM and greatly increase the performance of existing batch style workloads.

Though scalable and robust, HDFS fails to provide efficient support for applications that require random read accesses as its basic storage system is disk. Currently, users will experience extreme latency when reading from HDFS. There have been many approaches to improve the read latency of HDFS. One approach is RAM-HDFS [6] where the authors store the entire data in memory and use the disk for backup and failure recovery. However, this is not an optimal solution since there is very limited RAM available for computations as most of it is occupied by HDFS data. As most of the files are not accessed frequently, keeping the entire data in memory leads to wasted resources.

Another approach is to use basic caching mechanisms like LRU or LFU to maintain recently and frequently accessed data so that data can be fetched faster than that of disk. LRU uses only the last reference time whereas LFU uses only access count of a file to take caching and eviction decisions. Most of the HDFS workloads follow a Zipf distribution [7] i.e. only a small number of files account for majority of accesses. LRU and LFU are not optimal as they use the entire history instead of a particular timespan. Files that were accessed heavily initially will still be in an LFU cache even after not being accessed for a long period of time. Similarly, files that haven't been accessed frequently but have been referenced in the immediate past will remain in an LRU cache irrespective of their access count. Although both these parameters are equally important for caching decisions, for Zipf type distributions, it is important to consider these parameters for a certain time window rather than the entire history.

We propose IDecider - an intelligent caching and eviction algorithm that can achieve significant improvements in latency for HDFS based applications. IDecider uses a sliding window to decide upon file caching and eviction. It

uses file characteristics for a particular window to determine a file's popularity. File popularity is one of the most important metrics is gauging if a file is going to be accessed again in the future and is seldom used in existing caching algorithms for HDFS. File popularity is calculated as a weighted metric using file access patterns, file age and workload characteristics over a particular time window. A time window is maintained on a per file basis and can be dynamically configured. The window size is determined based on a file's popularity at that particular instant of time. IDecider uses the access rate of a file in a particular time window and a threshold value in order to decide whether the file needs to be cached or not. The threshold is a configurable parameter and its value is decided by analyzing various production workloads. Least popular files in the cache are evicted if the cache is full. We use a simulator based technique to evaluate IDecider using various performance metrics by using real world and synthetic traces. These traces were obtained from the Mimesis benchmark [8], which is a synthetic meta-data workload generator.

1.2 Graph Summarization

It has become exceedingly important to provide data driven models in order to make decisions based on the data present. In order to provide quick insight into the data, we have designed a scalable framework to efficiently answer complex queries over large-scale graphs. The framework provides an approximate result for the queries by combining information from underlying attribute-based summary graphs constructed from the original network. The results are obtained using a Naïve Bayes approximation model.

Recent rapid growth in real-world social networks has incentivized researchers to explore optimizations which can provide quick insights about the network. Due to this motivation, graph summarization and approximation has been an important research problem. Most of the work in this area has been focused on concise and informative representations of large graph. It is, however, difficult to come up with a single universal representation optimized for all purposes. Our work primarily focuses on task-based summarization of large graphs and answer queries which are computationally expensive on original graph, but have tolerance with regards to minor errors

in exact results. These queries, semantically, provide the same amount of information even with approximate results. One such example can be that of an agency which wants to estimate the number of people impacted by a certain advertising budget on a social network. The agency is less likely to change it's decision if the result is 4,515,736 or 4,481,973 but is mostly interested if the result is close to 4.5 Million or 8.5 Million.

Calculating the exact result of such queries can possibly take one or multiple passes of the complete original graph, which scales even up to Peta or Exa Bytes. Our contribution is a framework which can answer queries probabilistically in a highly efficient way using compact representations of original graph stored in form of summary graphs. These summary graphs are also optimized for space complexity, and only grow in terms of the number of attributes used to answer the query. One can then use a combination of these graphs to answer complex queries in an extremely efficient manner. While calculating the approximate results, the attributes are assumed to follow Naïve Bayes conditional independence. This assumption allows us to store pairs of attributes, and enables the computation of the joint probability of any combination of attributes. Thus, we can represent the entire summary and still avoid the combinatorial explosion of attribute combinations. We analyze our framework in terms of efficiency and accuracy over varying graph size and query complexity. Our results are promising and show that significant gains in runtime can be achieved using our framework without sacrificing too much on accuracy. In fact, we observe decreasing trend in error as the graph size increases.

The rest of the thesis is organized as follows: In the first part of this thesis, we first talk about caching in HDFS and improvement in read latencies in HDFS using popularity aware caching. In Chapter 2 we discuss related work and compare IDecider with existing caching literature. In Chapter 3, we present a detailed statistical analysis on data access patterns in production clusters that we have obtained by studying file traces from Yahoo and Facebook. In Chapter 4, we discuss the two approaches of IDecider and give a detailed algorithm for both these approaches. We start off with a basic IDecider approach and then go on to explain the more complex popularity based sliding window IDecider approach. In Chapter 5, we give the implementation details for both the IDecider approaches. This chapter also details

out on several experimental evaluations done to evaluate IDecider . The next part of this thesis covers graph summarization and using graph summarization to efficiently probabilistically approximate the result of various data intensive queries. Chapter 6, gives a detailed literature review of various Graph Summarization, Graph Sampling and Query processing techniques used for obtaining information from graph data. In Chapter 7, we propose the graph summarization framework and methodology and also detail out on the dataset used to evaluate this framework. Chapter 8 provides details on the various experiments performed to evaluate the graph summarization framework and also gives a detailed complexity analysis of the algorithm along with various results on real-world data. We discuss directions for future work in Chapter 9. Finally, we conclude the thesis in Chapter 10.

CHAPTER 2

RELATED WORK - CACHING

There has been a large amount of work done on caching in general but limited work has been done on caching specifically in HDFS. Also, it is extremely difficult to obtain real world traces and lately much work targets generating synthetic traces that resemble those used in real production scenarios. In this section we first detail some existing solutions used to obtain petascale workloads characteristics and then restrict ourselves to cache replacement policies used in HDFS.

2.1 Workload Generation

Release of peteascale traces by industry would certainly open new opportunities for research into next generation distributed file systems, but until that happens we are limited to the few model based traces and literature summaries that are available today. Obtaining full traces regarding file system meta-data such as file access patterns, file sizes, and timestamps of important file events for the life time of a file is limited to researchers in industrial settings. This limitation is due in part to the fear of information leaks or business practices. An alternative to full traces has been the usage of modeling tools to statistically model a given workload and then use that model to generate synthetic traces. instead of releasing full traces, corporations have opt-ed to release model traces. One such workload generator is Mimesis [8]. Mimesis captures the statistical model and workload characteristics based on access patterns and then generates synthetic traces that mimic actual workloads. These traces are file level traces which capture the various events carried out on a file at different instances of time during the file's lifespan. IDecider uses traces generated from Mimesis in order to analyze the different patterns in data and then use this analysis to develop caching and eviction

policies.

2.2 Caching

Simple cache replacement policies leverage a single basic property among the numerous file meta-data characteristics. Least Recently Used (LRU) leverages temporal locality - namely, that recently accessed objects are likely to be accessed again in the near future. Least Frequently Used (LFU) leverages the fact that files that were accessed for a large number of times are likely to be popular; this guides the design of the LFU eviction mechanism. One of the other popular caching mechanisms is Largest File First (LLF) that leverages the negative correlation that exists between file sizes and likelihood of accesses - small objects have a higher probability of being referenced in the near future. Another variant of LRU that outperforms the basic LRU was the adaptive replacement cache (ARC) [9]. ARC is a self-tuning, low-overhead algorithm that responds online to changing access patterns. It continually balances between the recency and frequency features of the workload, demonstrating that adaptation eliminates the need for the workload-specific pretuning that plagued many previous proposals to improve LRU.

Early characterizations of file access patterns suggest the presence of strong temporal locality of reference [10]. Also, many of the early characterizations of access patterns suggest a strong preference for small objects. However, more recent studies have concluded that this temporal locality and preference for smaller objects is weakening [11]. Most popular files in HDFS workloads range from MBs to GBs, not quite what has historically been considered "small". Most of the HDFS workloads observed and analyzed led us to conclude that a lot of recently accessed files are unlikely candidates for accesses in the near future [12]. This limits caching algorithms based on LRU and, as we show with our simulation, these algorithms prove to be inadequate. Also, an important observation has been that a large number of objects are popular over short periods of time. These objects later become unpopular even though they have been accessed in a large number, limiting the applicability of caching algorithms based soled on access count.

The above characteristics of HDFS workloads limit the functionality of LFU based caching algorithms as objects which were accessed a large number

of times in a short time frame, (i.e.: right after being loaded into HDFS) will always remain in the cache even though they have a small probabilistic chance of being accessed in near future. These observations have strongly motivated our idea of IDecider as it does not consider the entire history of the file but only considers a particular window of time in which the file is being accessed. As time progresses, the window size per file dynamically changes based on the file's popularity in the previous window and IDecider makes sure that files that have been popular in the recent past remain in cache rather than unnecessary files leading to a strong performance boost over algorithms that use LRU, LFU and LLF as shown in our simulations.

2.3 Improvements in HDFS Latency

A limited amount of work has been done in reducing latency in HDFS, leaving opportunity for researchers to work on improvements. PACMan [13] is one such system for coordinated memory caching for parallel jobs and aims at reducing the job completion times to improve cluster efficiency. However, PACMan uses a variant of LFU called LFU-F as its eviction policy and thus does not provide the most effective solution to today's Zipf HDFS workloads.

A number of in-memory solutions have been proposed for improving latency in HDFS workloads. Memcache [14] has been used widely by industry to achieve faster reads and reduce latency. Facebook leverages Memcached [15] to construct and scale a distributed key-value store for all of its social network data. However, in-memory solutions may not always prove to be the best as they require a majority of RAM usage to go into storing HDFS files, leaving only a small portion available for the actual computational workload, such as mappers and reducers in Hadoop. A similar paper, RAM-Cloud [6], provides an in-memory caching solution and faces the same resource limitations.

IDecider is a novel caching approach that takes file popularity over a certain instance of time for a file and decides whether to cache the file or not. It does not consider any history outside of the time window for that particular file and thus provides a more optimal solution when compared with LRU and LFU for Zipf type workloads. We also propose a feedback looped

based version of IDecider which will decide the file's popularity based on the feedback received from the accesses of the file in the previous time window and then come to a more efficient eviction decision. Since we have file popularity as a decision metric, we can use this metric to characterize files into hot, warm and cold files. We can maintain files in different levels of storage like Memory, SSD and Disk based on the category decided. Based on these characterizations of files, IDecider can also be configured to decide the exact storage location of files in order to improve the latency even further by providing tier based caching.

CHAPTER 3

ANALYSIS OF DATA ACCESS PATTERNS IN REAL-WORLD PRODUCTION CLUSTERS

We analyze statistical information regarding files and access patterns in one of the Yahoo clusters. In one of the Yahoo clusters, grid(dilithium-gold), we used Hadoop command (hadoop dfs -lsr) to collect stats for file and analyze audit log (NameNode logs) to identify access pattern for each file. The data set contains the total number of files, total file size, number of file accesses, number of days between the first access and the most recent access, file distribution, deletion rate of files and directories, creation rate of files and directories in the dilithium-gold cluster. The data set was collected between 01/01/2009 to 03/13/2010.

We also used a Facebook trace that was generated using the Mimesis tool for our simulations. This Facebook trace is a 24 hour data set containing timestamps, file ids and file operations performed on those files. Table 3.1 gives details of this data set.

Table 3.1: Facebook Dataset Features

Total number of files	4,307,299
Total number of unique timestamps	46,442,327
Total number of events or file operations	72,299,640
Total number of create operations	4,307,299
Total number of open operations	66,222,501
Average operations per timestamp	4

The following is our analysis based on the above data sets.

3.1 File Accesses - Yahoo

Figure 3.1 shows a distribution of files accessed and the frequency of file accesses. The data set consists of $41,223,652$ files in total out of which $8,673,686$ files were never accessed. It can be seen that most of the files (73%) were accessed less than 30 times in the cluster. We see that very few files are accessed more than 50 times. We use this information to decide the threshold value for access rate of a file beyond which the file needs to be cached.

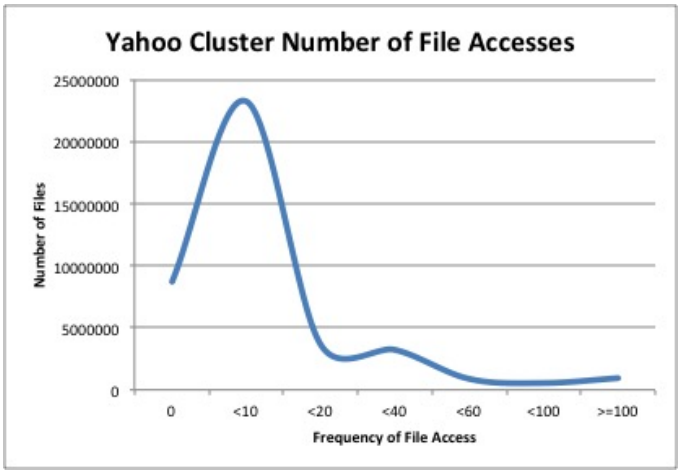


Figure 3.1: File Accesses and Frequency of Files accessed

3.2 File Size - Yahoo

Figure 3.2 shows a distribution of file sizes in the Yahoo Cluster. It is consistent with most of the observations of previous studies which establish the fact that majority of file sizes (58%) lie in the range of MB to GB. There is peak in the graph which clearly depicts this observation. Also, it can be seen that there are a lot of small files (41%) which are below 1MB. These parameters are used in deciding cache size.

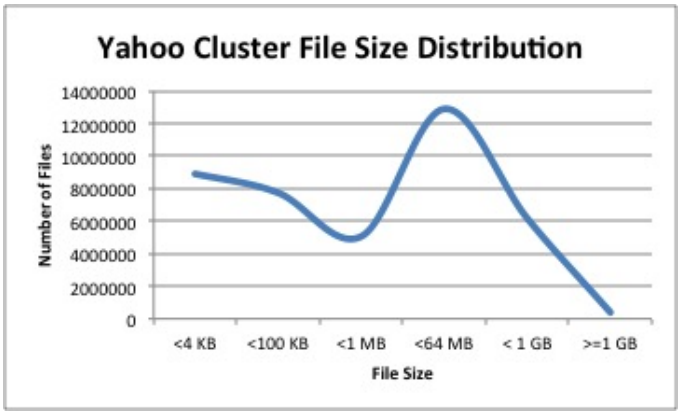


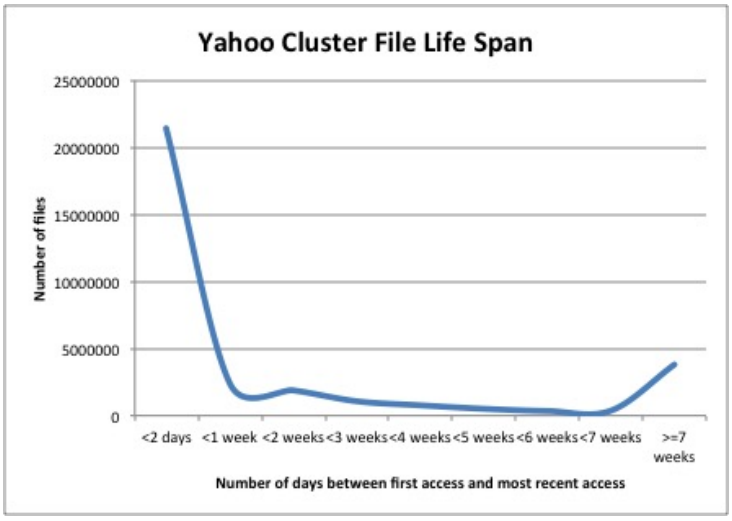
Figure 3.2: File Size Distribution

3.3 File Life Span - Yahoo

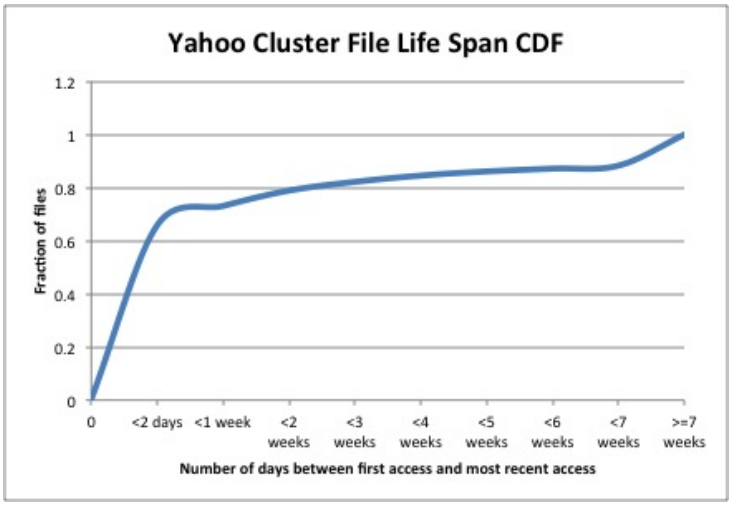
Figure 3.3(a) depicts the life span of files in the Yahoo Cluster. Life of a file is defined as the number of days between the first access of the file and its most recent access. We can see from Figure 3.3(a) that, majority of the files (73%) are accessed within the initial 1 week. As time progresses, the number of accesses for a file decreases exponentially.

Figure 3.3(b) represents a cumulative distribution function of the file age at time of access. This shows a high temporal correlation in accesses. We found that approximately 73% of the accesses of a file occur during 2 days after creation. A similar graph is presented by Abad et al. [16] obtained from a Yahoo 4000 node production cluster from January 2010. Another work by Kaushik et. al. [17] on power aware HDFS conveys similar analysis. The cluster had 2600 servers, and approximately 34 million files comprising of 5PB of data. It was seen that about 90% of the files were accessed within first 2 days of creation. The study also concluded that the majority of the data is hot for less than 10 days after its creation in the system. We use this analysis to decide the initial window size.

Similar industrial workload patterns were also analyzed and we observed a similar distribution of file accesses and life span. Muralidhar et. al. [18] have given similar analysis for Facebook's warm BLOB storage. We can see from the graphs in their study that the file distribution follows Zipf law.



(a) File Life Span



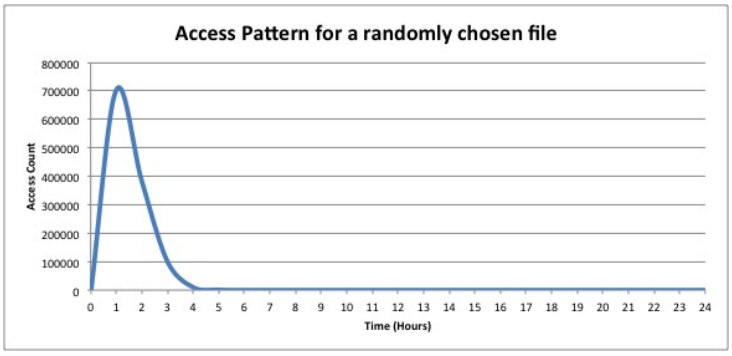
(b) CDF of File Age at time of access

Figure 3.3: File Life Span and Age

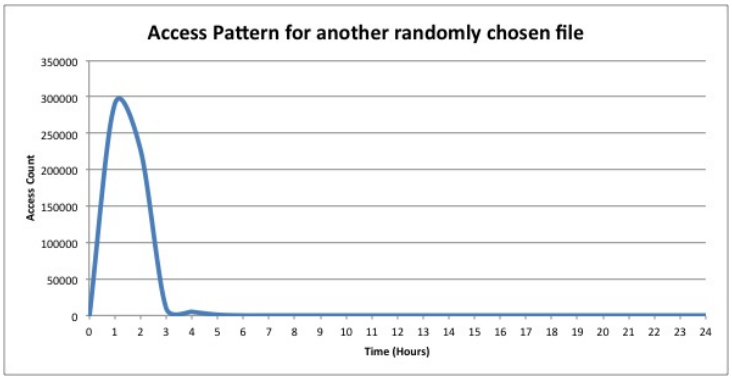
3.4 File Popularity - Facebook

The trace that was used consisted of data for an entire day. We divided it into 1 hour chunks and computed the popularity for two random files $F1$ and $F2$. These were created at the same timestamp and thus, we ignore the file age as it will be same for both the files. Figure 3.4(a) and Figure 3.4(b) show the popularity distribution over 24 hours files $F1$ and $F2$. Total access count for file $F1$ was 1,084,663 and that for file $F2$ was 517,885.

It can be seen from the figures that the computed popularity also follows Zipf distribution and thus, is a good measure to decide caching and eviction.



(a) Popularity over 24 hours for file F1



(b) Popularity over 24 hours for file F2

Figure 3.4: File Popularity Analysis for the Facebook trace

CHAPTER 4

IDECIDER ALGORITHM

We explain the IDecider algorithm in this chapter. We begin with a very basic IDecider algorithm and then move on to a more complex algorithm which is a sliding window based IDecider approach based on file popularity.

4.1 A Very Basic IDecider Approach

Algorithm 1 describes the caching and eviction policies for our basic IDecider approach.

Input: File Name, Timestamp.

Output: Flag to indicate whether to cache the file or not.

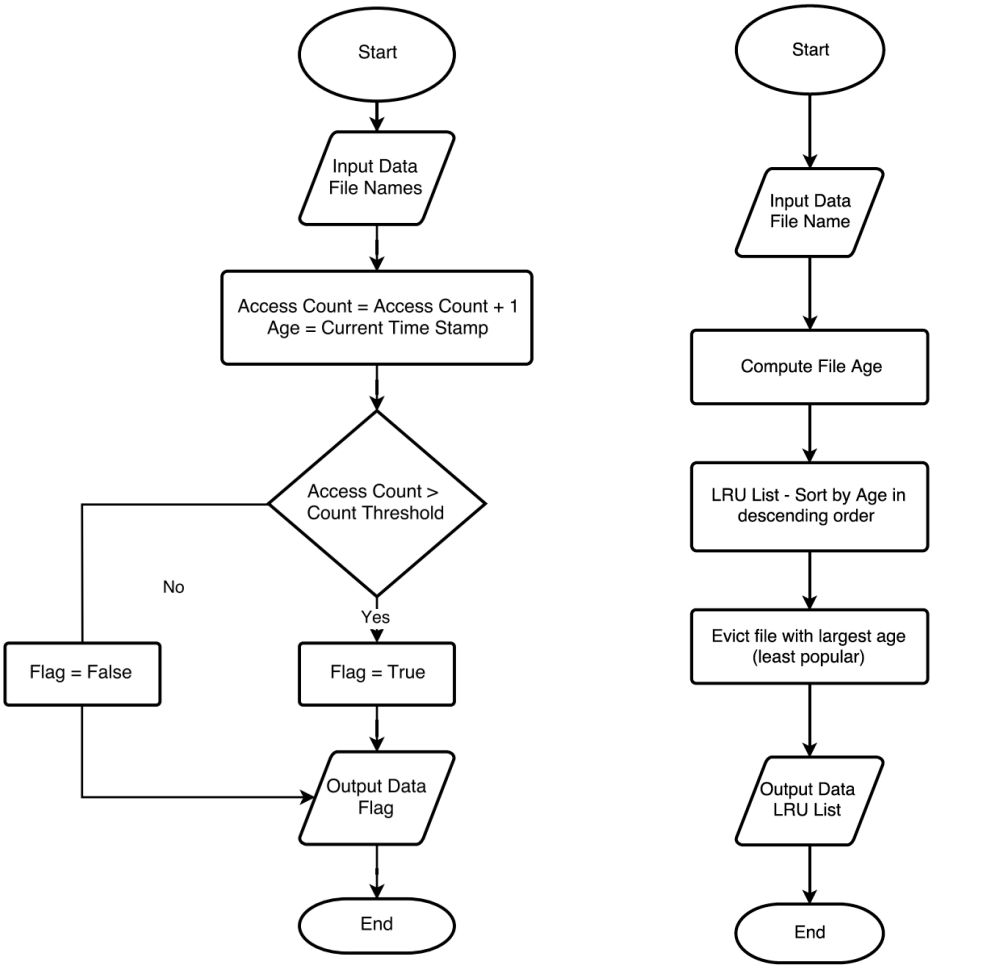
Assumptions: File age here denotes time elapsed between the first access and most recent access of the file. The threshold indicates the number of times a file needs to be accessed before it will be cached. By analyzing the Yahoo and Facebook traces, a threshold value of 3 was decided. An LRU list is maintained based on file age.

Initial: $age = current\ timestamp,\ access\_count = 1,\ threshold = 3$

```
access_count++ ;                                  // For every read operation
if access_count > threshold then
    if isCached == true then
        ;                                          // File is already cached
    else
        isCached = true ;                          // Cache the file
    end
    update LRU list
```

Algorithm 1: Algorithm for Basic IDecider Approach

Figures 4.1(a) and 4.1(b) describe the decision and eviction phases of Algorithm 1 based on access count and file age.



(a) Decision phase of Basic IDecider

(b) LRU based Eviction phase of Basic IDecider

Figure 4.1: IDecider Decision and Eviction Algorithms

4.2 Popularity and Sliding Window based IDecider Approach

Algorithms 2 3 and 4 describe the sliding window based IDecider mechanism using file popularity. As discussed in Chapter 2 in the literature review, the file access patterns follow a Zipf distribution and LRU and LFU both use history in order to make decisions, resulting in inefficient caching decisions. In order to improve the efficiency of caching, we use a sliding window based approach. As per our analysis specified in Chapter 3, we have decided the initial window size for all files should be 24 hours. This is because a file typically remain hot for a day after creation. These observations are clearly

depicted in our analysis of the Yahoo and Facebook data set in Chapter 3. We define *AccessRate* of a particular file as the total number of accesses during its specific window period.

$$Access\ Rate\ =\ \frac{Number\ of\ accesses\ of\ a\ file}{File\ Window\ size}$$

Based on our analysis, we initialize the threshold to

$$Threshold\ =\ \frac{3}{Window\ size}$$

At the end of each window for a file, we calculate the file's popularity using a popularity function given as

$$Popularity(file)\ =\ \frac{AccessRate}{Fileage}$$

We use the above file popularity function, to decide the next window size for that file. Window size is directly proportional to the file's popularity. It can be seen that, as the popularity increases, window size will increase because the access rate for popular files will be higher than that of others. Whenever a window expires for a file, we reinitialize the access count of the file to 0 since so as to not account for any historical meta-data outside of a given window. This makes sure that file popularity is always based on a particular instance of time and not on the entire history. We use a Least Popular Files (LPF) list in order to decide which files need to be evicted when the cache is full. The file at the head of this list is the least popular among all files in the list

Input: File Name, Timestamp, File Operation.

Output: A flag which tells whether a file is cached or not.

Assumptions: We assume that files are hot for 24 hours after their creation. This assumption comes from our analysis of the Yahoo data set.

Initialize: $window\ size\ =\ 24\ hours,\ access\ count\ =\ 0,\ threshold\ =\ \frac{3}{24}$

The basic idea of Algorithm 2 is to check if the cache is full or not and

decide whether we need to evict anything from the cache or just calculate popularity and update the window size for the new file and add it to the cache.

```
Function LPFcache(file) is
    if cache == full then
        evict() Least Popular File from cache
        addToCache(file)
    else
        calculatePopularity(file)
        updateWindowSize()
        addToCache(file)
    end
    Update Cache Size
end
```

Algorithm 2: Skeleton of the LPF Caching IDecider algorithm

Algorithm 3 explains how we update the popularity and window size for a particular file. As defined in the previous equations, popularity is computed using file access rate in the current window and the file age. If the new popularity of the file is greater than the old popularity, we double the window size. If not, we reduce the window size to half of the original.

Algorithm 4 explains the eviction decision based on popularity. We use two popularity thresholds θ_1 and θ_2 to choose whether to evict a file or not. If the file's popularity falls below θ_2, we evict the file and update the cache size. If it's between θ_1 and θ_2, we just mark the file for eviction and evict it on the next run whenever the popularity falls even more.

Function *calculatePopularity(file)* **is**

$$fileAccessRate = \frac{file.AccessCount}{file.getWindowSize()}$$

$$fileAge = file.getStartWindowSize() + file.getWindowSize() - file.lastAccessTime()$$

$$newPopularity = \frac{fileAccessRate}{fileAge}$$

if $newPopularity \geq oldPopularity$ **then**

$$file.windowSize = file.windowSize \times 2$$

else

$$file.windowSize = \frac{file.windowSize}{2}$$

end

updatePopularityThreshold() ; // Set θ_1 to newPopularity and θ_2 to oldPopularity

end

Algorithm 3: Popularity and Window Size update

Function *evict()* **is**
  for *file in cacheList* **do**
    if $file.popularity \leq \theta_2$ **then**
      cacheList.pop(file) cacheSize–
    else
      if $file.popularity \leq \theta_1$ **then**
        ; // mark for eviction
        file.markForEvict = True
    end
    if $cacheSize == cacheCapacity$ **then**
      cacheList.pop()
  end
end

Algorithm 4: Eviction algorithm

CHAPTER 5

IDECIDER IMPLEMENTATION, EVALUATION AND RESULTS

This chapter describes the implementation details of IDecider algorithm. The chapter also covers the experimental setup, the evaluation framework used and the evaluation results of the IDecider algorithm in comparison to some of the other popularly used caching techniques.

5.1 IDecider Implementation Details

This section covers the implementation details of the IDecider algorithm. We begin with the IDecider design and architecture when coupled with HDFS and then talk about the implementation of basic IDecider approach described in Section 4.1 and then later move on to the implementation of the more complex sliding window based approach described in Section 4.2 of Chapter 4.

5.1.1 IDecider Design and Architecture

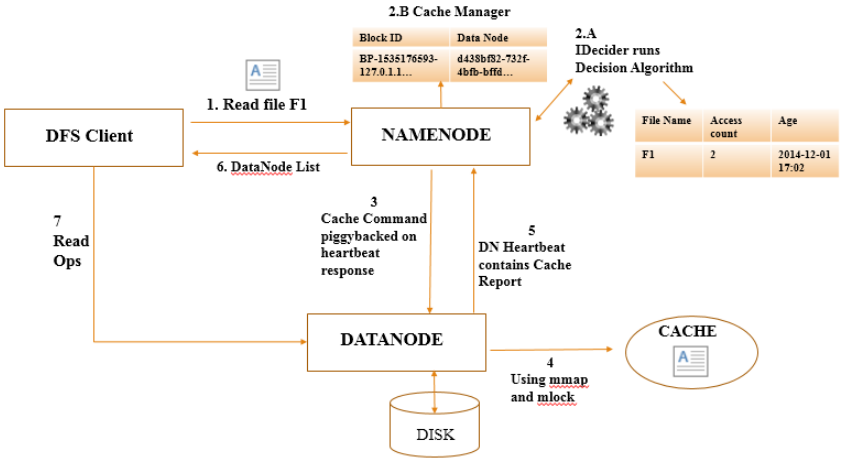


Figure 5.1: IDecider Architecture

Figure 5.1 shows the IDecider architecture with HDFS. IDecider is placed in the Namenode. The advantage of having IDecider in the Namenode is that it can easily interact with the Namespace. Files in HDFS consist of blocks. The information about any particular file is contained in the HDFS Namespace. Each block is replicated and stored on multiple Datanodes. The mapping from block replicas to Datanodes is stored in the INode data structure. The HDFS Namenode has two critical tables managed as HDFS Namespace.

1. filename → block sequence (NameSpace)
2. block replica → machine list (INodes)

The Namespace is persistent while INodes are not. The FsImage file contains all information about the file in an INode data structure. On startup, the FsImage file is loaded and various objects are created based on the INode entries in this file. INodes are created in the event of a Namenode restart with the help of block report sent by Datanodes. This block replica entity contains all the information related to the different blocks of a particular file. The design choice of having IDecider in the Namenode is motivated by the easy access to file information from the Namespace and all the block information from the INodes. The IDecider component keeps track of statistical information about the file.

Here, we introduce two implementations of IDecider . First, we implemented a simple access count based caching policy using LRU as its eviction policy. We evaluated this approach using a pseudo-distributed system environment. Second, we implemented a more complex version of IDecider that uses the sliding window based caching policy and the popularity metrics specified above. We evaluate this approach using a trace driven simulator and compare it's results with some of the existing caching algorithms.

One of the most important design decisions was whether to store file popularity and metadata in a map or whether to store it as part of the INode. Below we highlight the pros and cons of the two methods considered.

1. **Store file meta-data in a map data structure**

   This implementation method requires an entirely new map data structure to be maintained. The map will contain all file popularity informa-

tion and must be explicitly maintained as a separate entity, requiring much error prone maintenance code. There are a few advantages of this implementation, which have been listed below.

(a) Easy to verify whether a file is cached or not.

(b) Adding dynamic information at a later time is easy as it requires only updating the map data structure.

However, there are some disadvantages of using a separate map data structure.

(a) Since a new data structure is being added, it will require a lot of maintenance code to be written to ensure that the map is persistent and up to date.

(b) Persisting the map will increase the size of FsImage file and also the edit logs. Since the FsImage and edit logs are used during startup and checkpointing, both these processes will require more time; this is already a problem for the HDFS community at large.

2. **Storing popularity information in the INode data structure**

In order to avoid the drawbacks of maintaining a separate map, another approach was investigated. This new approach involves storing of all the popularity related information directly in the INode data structure provided by HDFS instead of maintaining a separate map. The following are the advantages of using this method of implementation.

(a) Minimal number of code changes.

(b) Low maintenance cost since the inode data structure is already handled by the existing HDFS implementation.

(c) Persistence is easier and the size will not be affected since these parameters will add very little to the size of the edit logs and FsImage.

One of the major disadvantages of using this method is that if information is to be added dynamically, it will require more code changes in the INode data structure in addition to changes wherever such data is being referenced.

Next, we give the implementation details of the basic and sliding window based IDecider approaches.

5.1.2 Basic IDecider Implementation

This section describes the basic implementation of IDecider and its functionality. The main function of the IDecider component is to consider all the necessary file parameters and dynamically decide whether to cache the file or not. Our basic approach uses access count of the file as a metric in making its decision. Based on a configurable caching threshold parameter, IDecider knows that the file has been accessed frequently and should be cached. Based on decision made by the IDecider , the Namenode will send a cache command to the Datanode. This command is generally piggybacked on the Datanode's heartbeat so as to reduce the network bandwidth utilization. Upon reception of a cache command, the Datanode will cache the respective file. After every cache interval time, the Datanode will send a cache report back to the Namenode. Cache interval time is a configurable parameter. We have programmatically configured this parameter in such a way as to ensure that the cache report is only sent if it has been updated in the last caching interval. This prevents redundant caching information being sent by the Datanode to the Namenode and helps in reducing the network traffic.

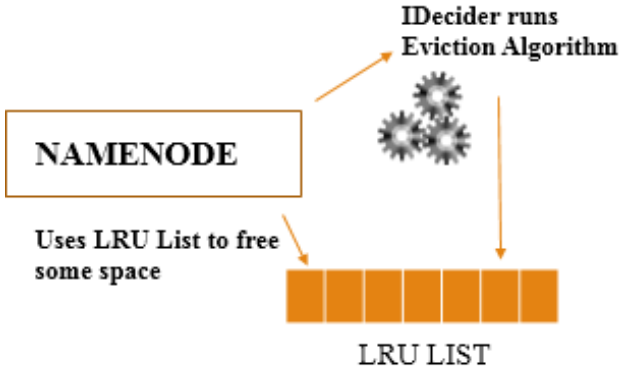


Figure 5.2: Simple IDecider LRU based Eviction Policy

Figure 5.2 shows our current implementation of the cache eviction policy based on LRU. An LRU list is prepared based on the file access times and the latest accessed file is placed at the head of the list. If the cache is full, the

file, which has been least recently used (tail end of the LRU list), is removed and the new file is inserted at the head of the list.

5.1.3 Popularity and Sliding Window based IDecider Implementation

This section describes the sliding window approach for IDecider to perform caching decisions. Using Mimesis [8], statistical information about real world workloads was obtained through the Statistical Analysis Engine (SAE). This information was fed as input to the Workload Generator Engine (WGE) of the Mimesis tool. This outputs a workload trace in the following format $<$ $timestamp$, $file\ id$, $file\ operation$ $>$. A trace driven simulator was developed that replayed events from the traces obtained. We define a transaction as an event in the trace of the form $<$ $timestamp$, $file\ id$, $file\ operation$ $>$. At a particular timestamp, there might be more than one transactions that needs to be processed. These were found by computing the inter-arrival times between consecutive transactions. Based on these inter-arrival times, we group transactions that have the same timestamp into a chunk so that we can fire all these transactions simultaneously using a pool of threads. Each chunk is processed by the simulator and caching and eviction decisions are evaluated based on the algorithm discussed in Section 4.2 of Chapter 4. The results obtained are discussed in Section 5.3.

5.2 Experimental Setup

We evaluate IDecider using a trace driven simulator technique as described in Section 5.1.3. We simulate a distributed cache environment on a cluster of 5 nodes and compare the various caching algorithms with our IDecider approach.

We use a dedicated cluster of 5 nodes to simulate the distributed cache environment. Each server contains 4 Intel Xeon E5620 processors and 32 GB RAM. The servers are connected to each other using a 1 Gbps network. We configure a master server that has 20GB of cache and configure 4 slaves to have 10GB of cache per node. We use the traces that were obtained from the Mimesis Workload Generator Engine. We deploy our generic simulator

on to the master. The trace is stored in the underlying HDFS and is fed to the generic simulator. This generic simulator is basically a plug-and-play engine which takes as input a caching algorithm and the trace and evaluates the caching algorithm in terms of the hit and miss ratios based on the trace. The trace is fed to the simulator in order of file operations occurring as per the timestamp values.

Along with IDecider , we have implemented various existing caching techniques like LRU, LFU and Adaptive Replacement Cache or ARC. We plug each one of these algorithms into the generic simulator and evaluate the hit and miss ratios. We have discussed the results in detail in Section 5.3.

5.3 IDecider Results

This section discusses the results of IDecider and gives a comparison between IDecider and various other caching algorithms like LRU, LFU, LRFU [19] and ARC. We compared the performance of various caching algorithms with our IDecider approach. We use two traces to evaluate the performance. The first set of experiments was performed on a synthetic trace produced by Mimesis and IDecider was compared with LRU, ARC and LFU. The second set of experiments was performed using the Yahoo trace described in Chapter 3 and we compared IDecider with LRU, ARC and LRFU.

Table 5.1 shows the hit ratio percentages of various caching algorithms for different cache sizes on the synthetic trace generated by Mimesis. We can see that on this trace, IDecider performs significantly better than LRU and LFU and has better hit ratios. However, IDecider couldn't beat ARC because ARC uses a multi-level caching and eviction technique which is quite similar to our popularity based technique. Figure 5.3 shows the performance comparison between these algorithms.

Table 5.2 shows the hit ratio percentages of various caching algorithms for different cache sizes on the Yahoo trace described in Chapter 3. We compare the performance of IDecider with ARC, LRU and LRFU which is a combination of LRU and LFU. We can see that on this trace, IDecider performs better than LRU and sometimes also better than ARC but not as good as LRFU. IDecider couldn't beat LRFU because combining LRU and LFU gives the same notion as popularity based eviction. Figure 5.4 shows

Table 5.1: Performance of caching algorithms on Mimesis trace

Cache Size (number of 512-byte pages)	Hit Ratios (%)			
	ARC	LRU	LFU	IDecider
1000	38.93	32.83	27.98	34.76
2000	46.08	42.47	35.21	43.04
5000	55.25	53.65	44.76	55.20
10000	61.87	60.70	52.15	61.66
15000	65.40	64.63	56.22	65.02
20000	66.98	65.2	60.08	66.27

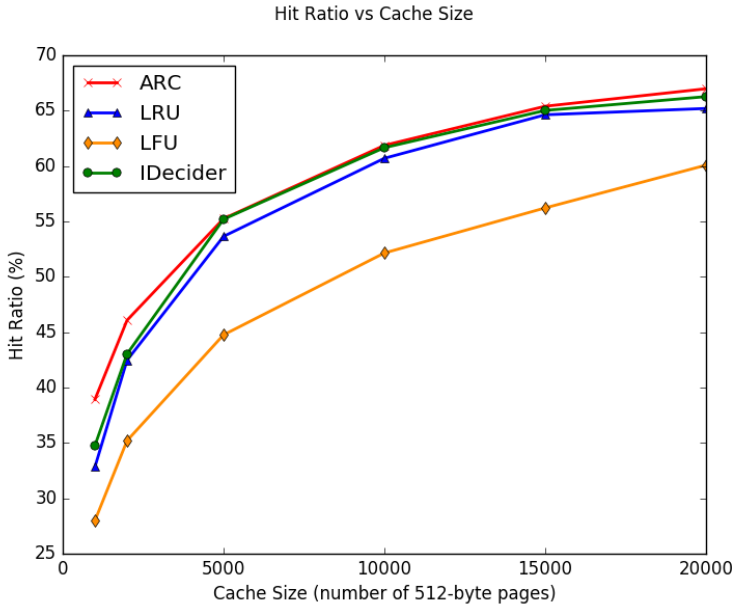


Figure 5.3: Performance of caching techniques - Mimesis trace

the performance comparison between these algorithms.

Table 5.2: Performance of caching algorithms on Yahoo trace

Cache Size (number of 512-byte pages)	Hit Ratios (%)			
	ARC	LRU	LRFU	IDecider
1024	4.16	4.09	4.09	4.10
2048	4.89	4.84	4.84	4.86
4096	5.76	5.61	5.61	5.61
8192	7.14	6.22	7.29	7.09
16384	10.12	7.09	11.01	10.03
32768	15.94	8.93	16.35	14.98
65536	26.09	14.43	25.35	25.98
131072	38.68	29.21	39.78	38.12
262144	53.47	49.11	54.56	53.03
524288	63.56	60.91	63.13	63.33

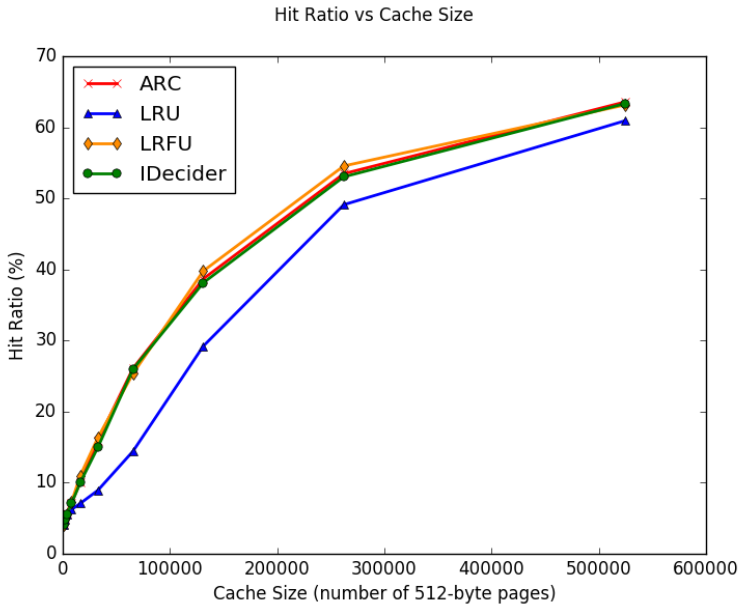


Figure 5.4: Performance of caching techniques - Yahoo trace

CHAPTER 6

LITERATURE REVIEW - GRAPH SUMMARIZATION AND APPROXIMATION

This chapter describes the literature review for the graph summarization and approximation techniques. The chapter is organized to detail out literature review for the four areas namely, graph summarization, graph sampling, query processing on large graphs and bayesian networks.

6.1 Graph Summarization

Graph summarization and network approximation using representative graphs has been an important research problem due to the ever-increasing size of real-world social networks. Since real-world graphs are massive in size and continuously evolving, a summary graph is often constructed using graph sampling techniques [20] in order to represent the entire network and study properties of the entire network. While most studies focus on graph summarization in terms of reducing the description length of graphs and graph compression [21], [22], recent developments in social network analysis give prominence to constructing the representative graph in a way that will allow studying properties of the entire network using just the representative graph. These developments offer new challenges to network summarization and extend the idea of summarization beyond just compression or minimum description length.

One such representative graph technique is constructing the supernodes using an interestingness-driven technique by Leskovec and Faloutsos given in [23]. Their work primarily focuses on computing the summary graph based on the evolving diffusion process over time. The idea is to summarize the entire network of millions of nodes by only capturing few interesting nodes or edges based on an interestingness measure defined for each node. The more interesting a node is, the higher is the probability of that node being a su-

pernode. The work studies dynamic network properties such as diffusion rate over time on this representative graph. One of the most important strengths of this work is that it is an online algorithm which will consider changes in the graph over time. However, a very important challenge in this approach is choosing the interval of time after which the network snapshot needs to be studied. If this interval is too small, the performance of the algorithm will deteriorate. If the interval is too large, some important diffusion properties might be missed. Also, the work fails to address whether the detected summary is actually a good summary and will capture the properties of the underlying graph and if it does, to what extent the properties are captured. Even though this work captures diffusion rate and change in information exchange, it does not describe any static measures such as degree distribution, path length distribution, etc. that can be studied on the representative graph and may represent the same behavior on the underlying network. Our approach involves storing attribute-specific summary graphs which preserve such underlying properties of the main graph.

6.2 Graph Sampling

Ahmed, et. al. proposed different graph sampling techniques [24] which actually capture the properties of the underlying network and preserve the same in the representative graphs. The paper lists different measures that can be evaluated on the sampled graph. Evaluation metrics include degree distribution, path length distribution, clustering coefficient, etc. One of the major strengths of this paper is that these algorithms are defined taking into consideration both static as well as streaming data. However, the paper constructs the sample based on edge connectivity and topology of nodes, but doesn't take into account attribute-based similarities. Answering multi-attribute queries using such sampling techniques will not be possible without sufficient preprocessing of the summary graph which stores attributes for nodes of the summary graphs. Our approach is to store different summary graphs based on the query type and the attributes that this query is dependent on, which cannot be done using generic graph sampling.

6.3 Query Processing Techniques for Large Graphs

Previous research on query processing for large graphs generally involves introducing different methods for indexing these large graphs so that queries can be handled in a manner similar to how they are handled in Relational Database Management System. [25] focus on developing indices over supergraphs and sub-graphs so as to efficiently process queries. The main strength of this paper is that the cost of complex join operations is avoided by storing intermediary metadata about the graph using the indices built on top of the graph. Using this indexing mechanism, they translate graph queries into SQL scripts and thus leverage the computational efficiency of relational operations in SQL in order to achieve fast query processing. The main drawback of this paper is that metadata about specific queries must be recomputed irrespective of the repetitive nature of successive queries. [26] deals with developing an indexing technique called eIndex over the supergraph. The entire graph is considered as a database in their graph query processing algorithm and candidate subgraphs are generated based on the query. The index creation step makes use of all the nodes in the original graph as well as the entrire feature set. The main strength of this paper is the proposal of an efficient solution for the super-graph query problem, such that approximate algorithms are used to reduce the number of sub-graph tests. The authors make use of this result for querying super-graphs efficiently. A major weakness in this paper is that it includes a post-processing step that takes time that is polynomial in the size of the graph. Though this step is carried out only once per run of the algorithm, it will not be practical for large graphs of the order of hundreds of millions of nodes. [27] proposes a high-performance graph indexing mechanism to solve the problem of efficient querying in large graphs. The major strength of this approach as compared to other indexing mechanisms is that a shortest-path approach with respect to vertex neighborhood is taken rather than a per-vertex approach, which makes the indexing highly scalable and much more efficient computationally.

A significant weakness of all index-based approaches to graph querying is the requirement of a large amount of storage. In all cases, to store indexes for all the nodes of a super-graph, without narrowing down the set of nodes based on attributes, will require storage proportional to the size of the entire graph (the number of nodes can be in the order of millions). We thus propose

a solution that generates attribute-based sub-graphs requiring storage that is considerably smaller than the size of the entire graph. These sub-graphs can furthermore be reused for queries that require the same combinations of attributes for their resolution.

6.4 Bayesian Belief Networks and Markov Networks

Bayesian Belief Networks make use of conditional probabilities across attributes to infer joint probability distributions. The exact inference task on these networks is NP-hard. In [28], Paul Dagum et. al. proved that there is no tractable deterministic algorithm that can approximate probabilistic inference to within an absolute error $\epsilon < \frac{1}{2}$. Second, they proved that there is no tractable randomized algorithm that can approximate probabilistic inference to within an absolute error $\epsilon < \frac{1}{2}$ with confidence probability greater than $\frac{1}{2}$. Even in special cases of approximate inference, computation time has been shown to be polynomial in input size. Thus, Bayesian Belief Networks are computationally expensive when it comes to query resolution. Unlike Bayesian Belief Networks which are directed graphs, Markov networks represent an undirected graph with conditional probabilities as edge weights. However, even in this case, above computational limitations hold true.

We have shown that our approach, which uses summary graphs, is computationally efficient and the runtime is independent of the underlying data size, but only depends on number of attributes and their values.

CHAPTER 7

GRAPH SUMMARIZATION FRAMEWORK

In this chapter we explain our graph summarization framework to get quick insight on large scale graphs. We explain the methodology behind constructing summary graphs, the math involved and the query resolution technique using the Naïve Bayes approximation. We also describe the dataset that we have used to evaluate our framework. We give the working of our summary graph technique on an illustrative example in Section 7.3 of this chapter.

7.1 Proposed Methodology

Our model involves a preprocessing step which computes and stores the summary graphs followed by a query resolution step based on the Naïve Bayes approximation model.

7.1.1 Preprocessing

The graph has N nodes. Each node has k attributes, and each attribute can take up to v values.

The preprocessing step involves constructing attribute based summary graphs of the underlying network. Since we want to compute pairwise conditional probabilities of attributes, we construct summary graphs for every pair of attributes. Since there are k attributes, we will have to construct $\binom{k}{2}$ summary graphs. For constructing each of these graphs, we will have to traverse the underlying N nodes. Once these summary graphs are constructed and stored, each query just uses these in the query resolution step. An advantage of using pairwise combinations is that we can compute any larger combination of conditional probabilities using these graphs in an efficient way.

7.1.2 Query Resolution using Naïve Bayes Approximation

In this section, we formally define the query resolution step of our model which uses Naïve Bayes approximation for estimating query results over the large social network graph. Currently, we focus on queries that investigate relationship among a given set of attributes. We can model a general query to have the form - *Find the probability of co-existence of attributes* $a_1, a_2, \cdots, a_q$ *given that the attributes* $b_1, b_2, \cdots, b_r$ *are present.* Thus, we want to estimate the probability

$$\Pr(a_1, a_2, \cdots, a_q | b_1, b_2, \cdots, b_r)$$

We use Naïve Bayes independence assumption for the attributes. Thus, the above expression of probability breaks down to the following

$$\begin{aligned}
&\Pr(a_1, a_2, \cdots, a_q | b_1, b_2, \cdots, b_r) \\
&= \Pr(a_1 | b_1, b_2, \cdots, b_r) \times \\
&\Pr(a_2 | b_1, b_2, \cdots, b_r) \times \\
&\cdots \times \Pr(a_q | b_1, b_2, \cdots, b_r)
\end{aligned}$$

$$\begin{aligned}
&= \frac{\Pr(b_1, b_2, \cdots, b_r | a_1) \Pr(a_1)}{\Pr(b_1, b_2, \cdots, b_r)} \times \\
&\frac{\Pr(b_1, b_2, \cdots, b_r | a_2) \Pr(a_2)}{\Pr(b_1, b_2, \cdots, b_r)} \times \\
&\cdots \times \frac{\Pr(b_1, b_2, \cdots, b_r | a_q) \Pr(a_q)}{\Pr(b_1, b_2, \cdots, b_r)}
\end{aligned}$$

Using Naïve Bayes Independence assumption,

$$\begin{aligned}
&= \frac{\prod_{i=1}^{r} \Pr(b_i | a_1)}{\Pr(b_1, b_2, \cdots, b_r)} \Pr(a_1) \times \\
&\frac{\prod_{i=1}^{r} \Pr(b_i | a_2)}{\Pr(b_1, b_2, \cdots, b_r)} \Pr(a_2) \times \\
&\cdots \times \frac{\prod_{i=1}^{r} \Pr(b_i | a_q)}{\Pr(b_1, b_2, \cdots, b_r)} \Pr(a_q)
\end{aligned}$$

Also, the evidence attributes $b_1, b_2, \cdots, b_r$ are assumed to be independent of each other and so, we have

$$\Pr(b_1, b_2, \cdots, b_r) = \prod_{i=1}^{r} \Pr(b_i)$$

The final expression thus breaks down to

$$\boxed{\frac{\prod_{j=1}^{q} \prod_{i=1}^{r} \Pr(b_i|a_j) \Pr(a_j)}{\prod_{i=1}^{r} \Pr(b_i)^{q}}}$$

7.2 Data

We are using the network of candidates applying to American graduate schools. Each student has multiple attributes such as standardized test scores, undergraduate university, demographic properties, and so on. Also, each student applies to multiple graduate universities, gets accepted by some and gets rejected by others. These interactions can be modeled as a social network with students and universities as entities. This dataset has entities i.e. students with several attributes, and is of considerable size to evaluate difference in performance on original graph versus summarization technique. We chose this dataset because it comes from a public forum, the complete dataset is thus available and there were no restrictions in regards to any privacy concerns or crawling limits. In case of networks like Facebook, Twitter, etc, since there are restrictions on crawling entire data, we only get a sample of data in every snapshot which might not be a representative sample of a complete graph.

There are several online resources where applicants share their admission experiences; Edulix is one commonly used resource [29]. It is an active resource which hosts applicant profiles from all over the world. GRE scores, undergraduate university name, GPA, TOEFL scores and other accomplishments such as work experience and research publications pertinent to the graduate admissions are reported in the profile. In addition, users mention

Table 7.1: Features of Data

General Features	
Total number of users before sanitization	36,207
Total number of users after sanitization	26,148
Features for CS related dataset	
Number of users	10,788
Application year range	[2001 2015]
Median Application Year	2013
Most Frequent Application Term	Fall
Number of universities with reported data	313
Number of applications per student (Mean)	6
Number of applications per university (Mean)	51
Number of undergraduate universities	2353
Degrees sought	[MS, PhD]

the universities that they applied to and the result of each application (*Admit*, *Reject* or *Result Not Available*).

The data used here was collected from this website. Since the data is self reported, it had some erroneous reports, which were identified and removed by completely deleting any such record. Any application that was not classified as either *Admit* or *Reject* was also excluded. It is observed that GRE and TOEFL scores have undergone various changes in grading scale over the years. Also, undergraduate institutions all over the world follow different scales for reporting GPA. As a standardization measure, these fields are mapped linearly to a scale of 0-100. An undergraduate university might be referred to by the differing names due to reasons such as usage of a popular acronym or spelling errors. This problem was mitigated by mapping the university names to their unique website URL. The dataset has students from various undergraduate departments and applying to multitude of graduate departments. Some of the statistics of a sample of this dataset, candidate applying to Computer Science graduate department, are in Table 7.1. Although Computer Science candidates make the plurality of our dataset, it is rich enough to accommodate students applying to various other engineering departments, business, humanities as well as biological sciences.

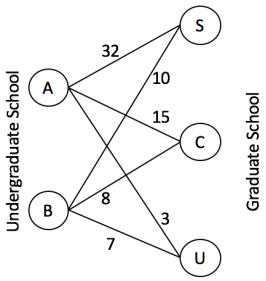
7.3 Illustrative Example

This section details out on an illustrative example to show the working of our framework. The example chosen consists of data sampled randomly from our dataset, which consists of 75 candidates. We ran simple queries on our model to observe the quality of results. We will go through the intuition and explanations of our framework by providing a walk-through of this query example. Based on the sample dataset, we have considered a query in which we only consider four attributes, namely Undergraduate School, Program. Decision and Graduate School. For the Undergraduate School, we have A and B as the possible values. For the Program, we have MS and PhD as the possible values, for the Decision, we have Ad and Rj as possible values to denote an admit or a reject, and for the Graduate School, we have S, C and U as the possible values.

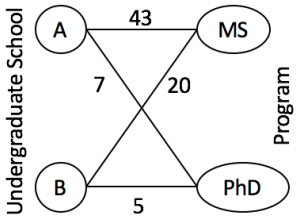
The Fig 7.1(a), 7.1(b), 7.1(c), 7.1(d), 7.1(e) and 7.1(f) show the summary graphs constructed from the preprocessing done on the sample dataset. An edge in a graph from node i to node j corresponds to the number of applications having i and j as their corresponding attribute values.

The sample query which we chose to evaluate our model on, asks - *If a person is from an undergraduate school A, what are her chances of getting admitted to the MS program at school S?* This query is important for students on the forum to get an idea of whether they can expect an admit from a university for a given program. Though the admission decision and graduate school are actually dependent attributes, we observe that assuming conditional independence by applying Naïve Bayes rule gets us approximate results which are very close to the actual probability. This slight compromise in accuracy is worth it since our framework provides query resolution in much less time than the actual approach which runs in time that is linear in the data size.

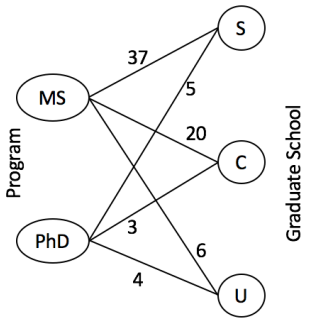
Ideally, if we were to look at the entire graph, we would have looked at all nodes that have *Undergraduate School = A, Program = MS, Decision = Ad, Graduate School = S* as their attribute value tuple. However, if we just consider the above summary graphs, from Fig 7.1(a), 7.1(b), 7.1(c), 7.1(d), 7.1(e) and 7.1(f) we can estimate the probability of the person coming



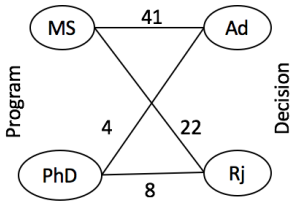
(a) Undergraduate School to Graduate School mapping



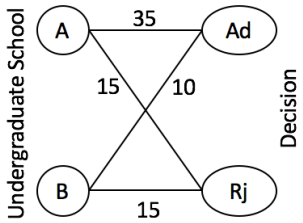
(b) Undergraduate School to Program mapping



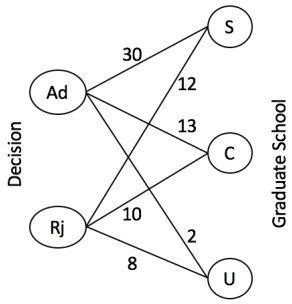
(c) Program to Graduate School mapping



(d) Program to Decision mapping



(e) Undergraduate School to Decision mapping



(f) Graduate School to Decision mapping

Figure 7.1: Summary graphs for the Illustrative Example

from Undergraduate School A and getting admitted to the MS program at Graduate School S.

From our model, we get an approximate result of 0.3853 for the query. A walk-through of this estimated result for the query is given below.

$$\begin{aligned}\Pr(S, Ad, MS|A) &= \Pr(S|A) \times \Pr(Ad|A) \times \Pr(MS|A) \\ &= \frac{32}{50} \times \frac{35}{50} \times \frac{43}{50} \\ &= 0.3853\end{aligned}$$

From this we can say that if a person were to finish her undergraduate education at school A, there is a 38.53% chance that she will get admitted to the MS program at graduate school S. The graphs in corresponding figures are generated for all 75 candidates that are contained in our sample dataset. In general, for a subgraph between attributes a_1 and a_2, we can see the number of candidates out of the total 75 that map to different values of a_1 and a_2.

In order to prove the correctness of our estimated value, we find extreme case bounds for the estimated result.

One extreme case will occur when the maximum number of applicants that get admitted to the MS program to graduate school S are from the undergraduate school A. This means that out of the 32 people that applied to S from A, 30 got admits (since we can see that 30 people got admits to university S overall). We make sure that this is consistent with the number of people that apply for the MS program from undergraduate school A (since $30 < 43$). This is consistent with the number of people that applied for MS to graduate school S (since $30 < 37$) and is also consistent with the number of MS applicants that got admits (since $30 < 35$). We also note that the total number of applicants from undergraduate school A is 50.

Thus, now the probability,

$$\Pr(S, Ad, MS|A) = 30/50 = 0.60$$

The opposite extreme case will occur when as many applicants from undergraduate school A that apply to graduate school S for the MS program get rejected as possible. We have 32 applicants from undergraduate university

A that apply to graduate school S. If we assume the maximum number of these applicants to be PhD applicants, we are left with 27 applicants since graduate school S has received only 5 PhD applications. Of these 27 MS applicants to graduate school S from undergraduate university A, we assume that the maximum possible number of these applicants get rejected. Thus we are left with 15 applicants from undergraduate university A that got admitted to the MS program at graduate school S (there are 12 applications that got rejected by graduate school S). Again, we note that the total number of applicants from undergraduate school A is 50.

Thus, the probability,

$$Pr(S, Ad, MS|A) = 15/50 = 0.30$$

These extreme case scenarios provide the bounds within which the actual answer of query (probability) should lie. It can be seen that our approximation model estimates a result within this range formed by the bounds.

$$0.30 \leq 0.3853 \leq 0.60$$

Calculating the exact result on the original graph will take processing of at least 50 nodes (75 in the worst case), connected to university A. For our estimation, we needed only 3 nodes from Fig 7.1(a), 2 nodes from Fig 7.1(b) and 2 nodes from Fig 7.1(e) resulting in only 7 nodes. Also, if the same graph were to scale to 50 million nodes, our model wouldn't need 5 million nodes, but most likely require nodes in the order of thousands (based on distinct values of undergraduate and graduate universities). Also, as the size of our original graph increases, the number of attributes participating in a query, and the categorical values of each attribute increase, the bounded window keeps on decreasing, and hence, our algorithm provides even better estimates.

CHAPTER 8

COMPLEXITY ANALYSIS, EXPERIMENTS AND RESULTS - GRAPH SUMMARIZATION

This chapter explains the complexity of the graph summarization algorithm in terms of time and space. Further it continues to detail out on the experimental setup, the various experiments performed on the dataset as well as the results for simple and complex queries on the summary graphs.

8.1 Complexity Analysis

In this section, we analyze our summary graph technique both in terms of time complexity and space complexity and compare it with the ordinary querying technique on large graphs.

8.1.1 Time Complexity Analysis

The preprocessing step involves constructing $\binom{k}{2}$ summary graphs. For constructing each of these graphs, we will have to traverse the underlying N nodes. Thus, the complexity involved in the preprocessing step is $\mathcal{O}(k^2N)$ The preprocessing step also involves computing aggregate values for each attribute. This will take additional $\mathcal{O}(kN)$ time. Thus the total preprocessing time is $\mathcal{O}(k^2N + kN) = \mathcal{O}(k^2N)$.

Let's look at the time complexity to process a query once we have summary graphs in place. For each query, of type $\Pr(a_1, a_2, \cdots, a_q|b_1, b_2, \cdots, b_r)$, there can be a maximum of k^2 graphs that we need to look at. In each of these graphs, we need to look at v values at maximum. Thus, the complexity of processing each query is $\mathcal{O}(k^2v)$. Without the summary graphs, if we were to answer the query based on the entire underlying network, we would require

to look at all possible $\binom{k}{2}$ combinations of attributes with v values each over N nodes resulting in a $\mathcal{O}(k^2Nv)$ time complexity. Thus, the preprocessing step reduces the query latency by a factor of N.

Also, with addition of a node to the underlying graph, the complexity doesn't change as we do not have to recompute any summaries but we just need to update the summaries based on the various attributes of the new added node. Let's say that there are N nodes in the underlying graph and we add another node with all k attributes present. So, we will have to update at max the k^2 graphs and each of this update is a $\mathcal{O}1$ time. So, in all the time complexity of adding a node is $\mathcal{O}k^2$ since it just involves updating the summaries that were already computed during the preprocessing step of the algorithm.

8.1.2 Space Complexity Analysis

As far as the space complexity is concerned, we have $\binom{k}{2}$ summary graphs, each consisting of $\mathcal{O}(v)$ nodes and $\mathcal{O}(v^2)$ edges. Thus, the space complexity can be given by $\mathcal{O}(k^2v^2)$.

8.2 Experimental Evaluation

This section describes the various evaluation metrics that we have used to evaluate our summary graph technique and also describes the experimental setup.

8.2.1 Evaluation Metrics

Our evaluation metrics are based on both the efficiency and accuracy of our algorithm in answering queries of varying complexity. The complexity of a query is based on the number of attributes for which we need to store subgraphs. Based on this observation, we evaluate our algorithm on the basis of the following metrics:

1. **Per-Query Runtime**: Per-query runtime is defined as the amount of time required to resolve a query as a function of data size. The resolution time of a query according to our model depends on the square of the number of attributes as we have shown in previous sections and independent of the data size. So, we expect the runtime to remain almost constant even when data size changes. In order to calculate the speed-up that we achieve, we have compared our per-attribute query time to the corresponding time required for retrieving the exact answer by processing the entire graph. The Results section shows that this speed-up is significant (in polynomial order of data size).

2. **Per-Query Error**: Since the resolution of a query depends on the Naïve Bayes assumption, we evaluate the accuracy of our query resolution as compared to the exact solution after having examined the entire graph. We define per-query error as the metric of accuracy. It can be defined as:

$$Error = |Accurate\ result\ -\ Estimation|$$

   As the proposed query approximation framework is efficient, we are able to achieve results that are within some error margin of the actual solution. This error margin is sufficiently small given the use case for our specific queries.

By evaluating the results obtained by our model on the above two metrics, we can observe the effect of query complexity (based on the number of attributes) on the accuracy of our model. Similar readings can be obtained for the speed-up that we achieve when compared to the scenario when the entire graph must be processed and a series of joins be taken in order to derive the exact solution to a query.

8.2.2 Experimental Setup

We evaluate both of the above metrics on simple as well as complex queries. The complexity of a query is defined by the number of attributes involved. As the number of attributes increases, more summary graphs are needed to obtain the estimation and thus the query complexity increases. For both

types of queries, we evaluate per-query runtime and per-query error for different data sizes. The results of our experiments are discussed in the next section.

8.3 Results

This section gives details about the results in terms of both accuracy and running time of simple as well as complex queries using our summary graph techniques. We compare our results with the ordinary technique and show significant improvements.

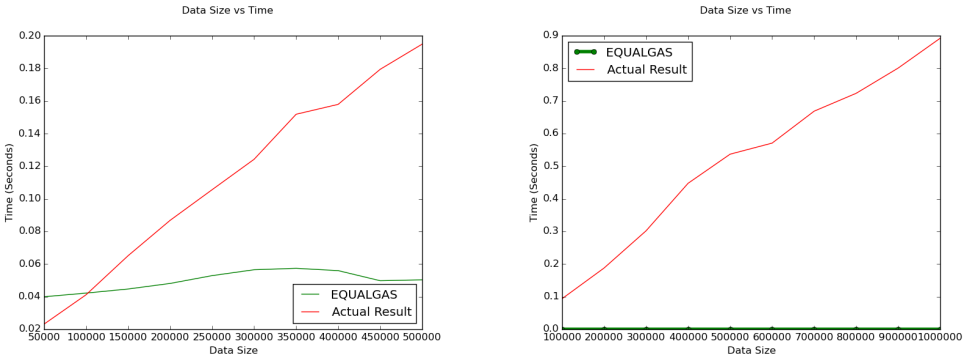
8.3.1 Per-Query Runtime

Fig 8.1(a) and 8.1(b) show the results of our experiments on simple queries. Fig 8.2(a) and 8.2(b) shows the results on a complex query. The plots are per-query runtime as a function of data size. We can see from these figures that the query estimation using summary graph remains almost constant as the data size increases while the time taken for actual query result calculation increases linearly. In Fig 8.1(b), our algorithm performs so efficiently that the time is very close to zero. This is because, the number of categorical values for each attribute i.e. v is very small. Even when this v is significantly large, as is the case in Fig 8.1(a) and 8.2(a), the run time is still constant for our algorithm.

In Fig 8.1(a) and 8.2(b), we can see that initially our algorithm is outperformed by the linear processing of data because of compiler and runtime optimizations. The cost of finding and accessing relevant summary graphs causes the extra time because of hashing operations. However, this cost remains constant irrespective of the graph size.

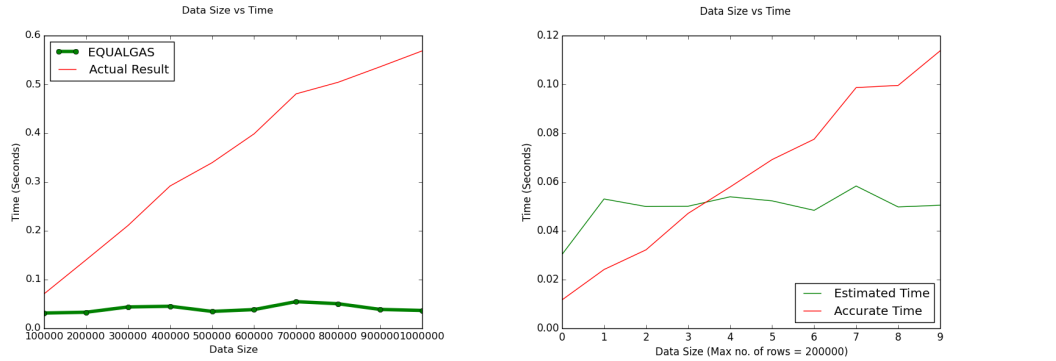
8.3.2 Per-Query Error

Fig 8.3(a) and 8.3(b) show the results of our experiments on simple and complex queries respectively. It can be seen from both these graphs that there is generally a decreasing trend in error as the data size increases. This



(a) $Pr(MS, NCSU|BITS)$ (b) $Pr(MS, CS|Fall)$

Figure 8.1: Data Size vs Runtime for Simple Queries

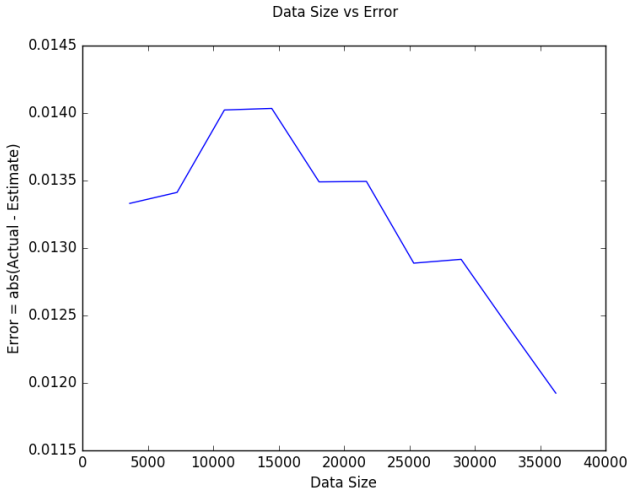


(a) $Pr(MS, CS, UTDallas|Fall, MU)$ (b) $Pr(MS, CS, NCSU, UTDallas|Fall, BITS)$

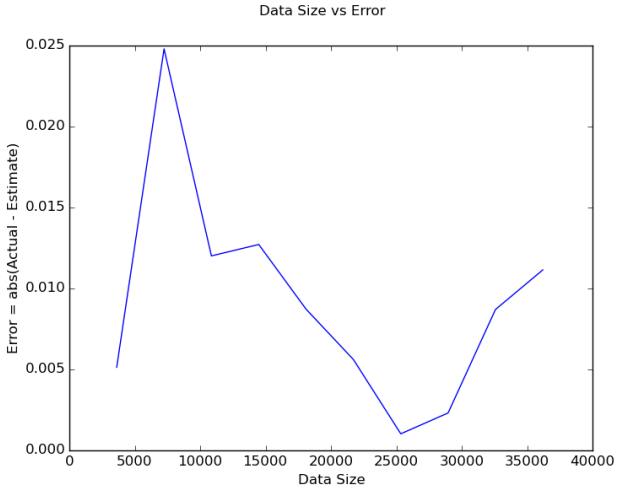
Figure 8.2: Data Size vs Runtime for Complex Queries

is expected since, as the data size grows, the bounds for extremities narrow down and our estimate always lies within these bounds.

The first peak for both the graphs is because the data is not yet large enough for probabilities to make sense. But as data size increases, probabilistic results model the actual data much better. The second peak in Fig 8.3(b) is because data in the last few chunks is extremely skewed in terms of Graduate School attribute distribution. However, over all error has a decreasing trend. In spite of the skewed nature of data, our algorithm always produces results within a 0.025 tolerance window of the actual probability.



(a) Simple Query - $Pr(MS, CS|Fall)$



(b) Complex Query - $Pr(MS, CS, NCSU|Fall, BITS)$

Figure 8.3: Data Size vs Accuracy for Simple and Complex Queries

CHAPTER 9

FUTURE WORK

We discuss possible future directions and extensions to our work both with respect to caching as well as graph summarization.

9.1 Caching - Next Steps

It can be seen that our IDecider approach is more efficient that the basic LRU and LFU approaches since it does not use any kind of historical information. File popularity has been a seldom used metric in making caching decisions. It is one of the most important parameters and can be used to effectively decide on what files are to be cached. Specifically when real world HDFS workloads follow a Zipf distribution, file popularity seems to be of utmost importance. Currently, we have defined file popularity only on the basis of file access rate and file age. However, we can extend the definition of popularity to incorporate other aspects such as file size, underlying workload characteristics and also a feedback loop provided by a dynamic caching algorithm. It can be then modeled as a weighted sum of all these different parameters as:

$$\begin{aligned} File\ Popularity\ =\ & \alpha File\ Size\ +\ \beta Access\ Rate\ + \\ & \gamma File\ Age\ +\ \delta Workload\ Characteristics \\ & +\ \epsilon Feedback \end{aligned}$$

$\alpha,\ \beta,\ \gamma,\ \delta$ and ϵ denote the normalized weight vectors.

Based on the above equation, we can device a feedback driven IDecider approach which will prove to be more efficient than the currently implemented one.

Since we have file popularity as a decision metric, we can even use this

metric to characterize files into hot, warm and cold files. We can maintain files in different levels of storage like Memory, SSD and Disk based on the category decided. At top level it appears to be one logical storage block but physically it will be multiple levels of storage. We can extend IDecider to even decide the storage location for files based on file popularity. We can also leverage the already existing heterogeneous storage API developed as part of the Apache Hadoop Project. This API was released as part of Hadoop 2.3.0 version. There has also been some work done on Heterogeneous storage by HortonWorks which can be leveraged to extend the IDecider functionality.

We can also do a detailed cost analysis and show how the choice of caching algorithm affects the cost of the cluster based on the cache size required to make each algorithm operate at equivalent performance. A comparison between LRU, LFU and our intelligent caching algorithm in terms of cost of cache per throughput increase can be done which will allow us to explore various ways in which we can reduce cache cost through the addition of multi-tiered caching using SSD's. We can thus solve the cost performance trade-off using a knapsack approximation to maximize the cache performance at the lowest cost.

9.2 Graph Summarization - Next Steps

Our graph summarization framework is computationally and spatially efficient and can be applied to large scale social networks. Our results are promising and as part of future work, we wish to apply these to other large scale graphs like Facebook and Twitter. We can then look at social network graphs and find out network characteristics such as diffusion statistics, influence metrics and ways in which you can have information diffusion efficiently using our approach. This will help in targeting information to the super-nodes in the graph so that we achieve maximum information diffusion.

Some of the probable future extensions to our work are lazy initialization of summary graphs using heuristics to predict probable and frequent queries, in order to reduce computations even further. We also plan to extend our framework to capture various social relationships and compute the probabilistic estimates based on these relationships.

CHAPTER 10

CONCLUSION

In this thesis, we have provided two solutions for low latency data retrieval on big data. We presented IDecider - an intelligent caching technique that does popularity aware caching to efficiently perform data accesses and reduce the read latency on HDFS. We also presented a Naïve Bayes based graph summarization framework that efficiently gives data insights without looking at the entire large graph but only looking at the summaries developed on the large graph. Our caching solution aims at faster data retrieval and our graph summarization solution aims at faster data insight.

10.1 Caching and IDecider

We found that file access patterns in HDFS workloads are heavily tailed with some files being more popular than others. For non-uniform file access patterns, current caching mechanisms that use historical file access data such as access count and last access time are inadequate and can result in inefficient caching by evicting popular files thereby leading to a higher job latency. We propose IDecider , an Intelligent caching and eviction mechanism that can significantly improve read latency of HDFS workloads by maintaining most popular files in cache at all times. Exploring both approaches of IDecider has led to a conclusion that file popularity is an extremely important parameter while taking caching and eviction decisions. Although we couldn't beat the results of ARC, it still means that LRU is somewhat an equivalent metric for popularity and there can be popularity incorporated into a basic LRU to make it more effective for Zipf type workloads.

10.2 Graph Summarization

In this paper, we have provided an efficient framework for query resolution over large graphs. Our summary graph approach does not depend on the size of the graph. By computing the summary graphs only once, we provide efficient calculations for subsequent queries. We achieve reductions in run-time in order of input data size. Moreover, even for varying complexity of the query and varying data size, our framework provides probability results within 0.025 of the actual. It is computationally and spatially efficient and can be applied to large scale social networks.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. USENIX Symp. Operating Syst. Design and Implementation (OSDI)*, 2004, pp. 137–150.

[2] "Apache hadoop," June 2011. [Online]. Available: http://hadoop.apache.org

[3] M. Israd, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, 2007, pp. 59–72.

[4] "Apache hdfs architecture," Feb 2014. [Online]. Available: http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[5] "Critical success factors of deploying hadoop across industries," Oct 2014. [Online]. Available: http://www.ibmbigdatahub.com/blog/critical-success-factors-deploying-hadoop-across-industries

[6] Y. Luo, S. Luo, J. Guan, and S. Zhou, "A ramcloud storage system based on hdfs: Architecture, implementation and evaluation," *Journal of Systems and Software*, vol. 86, pp. 744–750, March 2013.

[7] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," in *Proceedings of the VLDB Endowment*, vol. 5, no. 12, August 2012, pp. 1802–1813.

[8] C. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, and R. Campbell, "Metadata traces and workload models for evaluating big storage systems," in *IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, 2012.

[9] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[10] V. Almeida, A. Bestavros, M. Crovella, and A. De Oliveira, "Characterizing reference locality in the www," in *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*. IEEE, 1996, pp. 92–103.

[11] S. Jin and A. Bestavros, "Popularity-aware greedy dual-size web proxy caching algorithms," in *Distributed computing systems, 2000. Proceedings. 20th international conference on*. IEEE, 2000, pp. 254–261.

[12] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of facebook photo caching," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 167–181.

[13] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 20–20.

[14] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 371–384.

[15] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab et al., "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[16] C. L. Abad, Y. Lu, and R. H. Campbell, "Dare: Adaptive data replication for efficient cluster scheduling," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. Ieee, 2011, pp. 159–168.

[17] R. T. Kaushik and M. Bhandarkar, "Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster," in *Proceedings of the USENIX Annual Technical Conference*, 2010, p. 109.

[18] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang et al., "f4: Facebook's warm blob storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 383–398.

[19] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, no. 12, pp. 1352–1361, 2001.

[20] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 631–636.

[21] P. Boldi and S. Vigna, "The webgraph framework i: Compression techniques." in *WWW*, 2004, pp. 595–601.

[22] W. Liu, A. Kan, J. Chan, J. Bailey, C. Leckie, J. Pei, and R. Kotagiri, "On compressing weighted time-evolving graphs," in *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2012, pp. 2319–2322.

[23] Q. Qu, S. Liu, C. S. Jensen, F. Zhu, and C. Faloutsos, "Interestingness-driven diffusion process summarization in dynamic networks." in *Machine Learning and Knowledge Discovery in Databases*. Springer Berlin Heidelberg, 2014, pp. 597–613.

[24] N. K. Ahmed, J. Neville, and R. Kompella, "Network sampling: From static to streaming graphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 8, no. 2, p. 7, 2014.

[25] S.Sakr and G. Al-Naymat, "Efficient relational techniques for processing graph queries," *Journal of Computer Science and Technology*, vol. 25, no. 6, pp. 1237–1255, 2010.

[26] M. Saber, M. Aref, and T. F. Gharib, "An efficient filtering technique for super-graph query processing," in *Digital Information Management (ICDIM), 2010 Fifth International Conference on*. IEEE, 2010, pp. 174–181.

[27] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[28] P. Dagum and R. M. Chavez, "Approximating probabilistic inference in bayesian belief networks," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 15, no. 3, pp. 246–255, 1993.

[29] "Edulix - premier site for scholars - education crowdsourced." [Online]. Available: http://www.edulix.com/