ABSTRACT

Title of dissertation: MINIMIZATION OF RESOURCE CONSUMPTION THROUGH WORKLOAD CONSOLIDATION IN LARGE-SCALE DISTRIBUTED DATA PLATFORMS

Ashwin Kumar Kayyoor, Doctor of Philosophy, 2014

Dissertation directed by: Associate Professor Amol Deshpande,
Associate Professor Jimmy Lin
Department of Computer Science

The rapid increase in the data volumes encountered in many application domains has led to widespread adoption of parallel and distributed data management systems like parallel databases and MapReduce-based frameworks (e.g., Hadoop) in recent years. Use of such parallel and distributed frameworks is expected to accelerate in the coming years, putting further strain on already-scarce resources like compute power, network bandwidth, and energy. To reduce total execution times, there is a trend towards increasing execution parallelism by spreading out data across a large number of machines. However, this often increases the total resource consumption, and especially energy consumption, significantly because of process startup costs and other overheads (e.g., communication overheads). In this disserta-

tion, we develop several data management techniques to minimize resource consumption through workload consolidation.

In this dissertation, we introduce a key metric called *query span*, i.e., number of machines involved in the execution of a query or a job. In order to minimize the per query resource consumption we propose to minimize *query span*. To that end, we develop several workload-driven data partitioning and replica selection algorithms that attempt to minimize the average query span by exploiting the fact that most distributed environments need to use replication for fault tolerance. Extensive experiments on various datasets show that judicious data placement and replication can dramatically reduce the average query spans resulting in significant reductions in resource consumption. We show our results primarily on two applications, distributed data warehouse system and distributed information retrieval. In the first case, we show that minimizing average query spans can minimize overall resource consumption for a given workload and can also improve the performance of complex analytical queries. In the second case, our approach minimizes the overall search cost as well as effectively trades off search cost with load imbalance.

The best case of resource efficiency for any underlying data processing system is achieved when the job or the query can be run efficiently on a single machine (i.e., *query span*=1). In the final part of dissertation, we discuss an in-memory MapReduce system optimized for performing complex analytics

tasks on input data sizes that fit in a single machine's memory. We argue that systems like Hadoop that are designed to operate across a large number of machines are not optimal in performance for small and medium sized complex analytics tasks because of high startup costs, heavy disk activity, and wasteful checkpointing. We have developed a prototype runtime called HONE that is API compatible with standard (distributed) Hadoop. In other words, we can take existing Hadoop code and run it, without modification, on a multi-core shared memory machine. This allows us to take existing Hadoop algorithms and find the most suitable runtime environment for execution on datasets of varying sizes.

Overall, in this dissertation, our key contributions in this work include identification of key metric *query span* and its relationship with overall resource consumption in scale-out architectures. We introduce several workload-aware techniques to optimize this key metric. We go on to demonstrate the effectiveness of query span minimization on different application scenarios. In order to take advantage of scale-up architectures effectively we develop novel in-memory MapReduce system HONE for single machine. Our thorough experiments on real and synthetic datasets demonstrate the efficacy of our proposed approaches.

MINIMIZATION OF RESOURCE CONSUMPTION THROUGH WORKLOAD CONSOLIDATION IN LARGE-SCALE DISTRIBUTED DATA PLATFORMS

by

Ashwin Kumar Kayyoor

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Advisory Committee:
Associate Professor Amol Deshpande, Chair/Advisor
Associate Professor Jimmy Lin, Co-Advisor
Professor Alan Sussman
Professor Atif Memon
Professor Richard Marciano

Dedication

To

my gurus.

my mother K. Shakuntala Bhat.

my father K. A. Muralidhar Bhat.

my brother K. Vinay Kumar.

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to thank my advisor, Amol Deshpande for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past five years. I am also very grateful to my advisor for giving me enough freedom to pursue my own ideas.

I would also like to thank my co-advisor, Jimmy Lin. Without his financial support, his extraordinary ideas and computational expertise, this thesis would have been a distant dream. He has always made himself available for help and advice and there has never been an occasion when I've wanted to meet him and he hasn't given me time. It has been a pleasure to work with and learn from such an extraordinary individual. Thanks are due to Professor Alan Sussman, Professor Atif Memon and Professor Richard Marciano for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript. I thank Professor Samir Khuller for his guidance during my initial years of my PhD.

At UMD, I met many selfless professors, I would like to specially mention Professor Alan Sussman, Professor Clyde Kruskal and Professor Atif Memon who have inspired me in their own unique ways. Some have inspired me through their genuine suggestions, some have touched me through their brilliant sense of humor and friendly nature. Whereas some others have inspired me by simply being around.

My colleagues at the database lab have enriched my graduate life in many ways and deserve a special mention. My interaction with Abdul Quamar,

Udayan Khurana, Vikas Shivshankar, Arijit Biswas, Jayanta Mondal, Souvik Bhattacharjee, Amit Chavan, and Theodoras Rekatsinas has been very fruitful. I have been lucky to have these wonderful set of friendly colleagues.

I am grateful to Develop Empower Synergize India (DESI) an on-campus student organization for giving me a purpose during my time at UMD. I made great friends here and learnt a lot from them. I am extremely thankful to Ashutosh Gupta and Umang Agarwal for being with me and helping me become a better being.

I owe my deepest thanks to my family - my mother and father who have always stood by me and had faith in me. Words cannot express the gratitude I owe them. I thank my brother Vinay for supporting me and looking after my parents in my absence. My special thanks to my sister Kiranmayi Acharya for helping me numerous times during my foreign internships and while I was preparing my PhD application.

My housemates at my place of residence have been a crucial factor in my finishing smoothly. I'd like to express my gratitude to Saurabh Chandla, Rohan Kapoor, Bandaru Varaprasad, Ram Krishna and Aniruddha Gaekwad for their friendship and support.

Above all, I bow to my gurus Shri Krishna, Shri Nagaraja, Shri Nisargadatta Maharaj, Bhagawan Ramana Maharshi, Shri Siddharameshwar Maharaj, Lahiri Mahasaya and Shri Mahavatar Babaji for spiritually guiding me and taking care of me.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Lastly, thank you all and thank God!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Big Data Architectures

Large-scale data management and analysis is rapidly gaining importance because of an exponential increase in the data volumes being generated in a wide range of application domains. The deluge of data (popularly called "Big Data") creates many challenges in storage, processing, and querying of such data. There is also an overwhelming variety in the types of applications and services that we are witnessing today. There is growing consensus that a single system cannot cater to the variety of workloads, and different solutions are being researched and developed for different application needs. For example, *column-stores* are optimized specifically for data warehousing applications, whereas *row-stores* are better suited for transactional workloads. There are also hybrid systems for applications that need support for both

transactional workloads and data analytics. Other varied systems are being developed to store different types of data, such as document data stores for storing XML or JSON documents, and graph databases for graph-structured or RDF data.

To handle the increasing volumes of data, two solutions are commonly considered. One approach is to use a sufficiently powerful machine that can handle the workload (called the *scale-up* approach), whereas the other approach is to use a cluster of commodity machines to parallelize the compute tasks (*scale-out* approach). The scale-up approach is attractive because it is significantly easier to code for, whereas the scale-out approach requires one to deal with the distributed nature of computation as well as distributed fault-tolerance issues. However, the scale-up approach is limited in its ability to scale to large volumes of data, and also is typically more expensive. There has been much work on the scale-out approach over the last decade – several high-level programming frameworks and abstractions (e.g., MapReduce) have been proposed, and numerous systems have been developed for supporting those frameworks over a large number of machines.

Scale-out is typically achieved by partitioning the data across multiple machines. Machine failures present an important problem for scale-out architectures resulting in data unavailability. In order to tolerate machine failures and to improve data availability traditionally data replication is employed.

1.2 Resource Inefficiency in Scale-out Architectures

Although large-scale systems deployed over scale-out architectures enable us to handle the data volumes, velocity and variety efficiently, we note that these architectures are prone to resource inefficiencies. Also, the issue of *minimizing resource consumption* in executing large-scale data analysis tasks is not a focus of many data systems that are developed to date. In fact, it is easy to see that many of the design decisions made, especially in scale-out architectures, can typically reduce overall execution times, but can lead to inefficient use of resources. As the field matures and the demands on computing infrastructure grow, many of those design decisions need to be revisited with the goal of minimizing resource consumption. Another impetus is provided by the increasing awareness that the energy needs of the computing infrastructure, typically proportional to the resource consumption, are growing rapidly and are accounting for a large fraction of the total cost of providing the computing services.

Let us take a look at an example of resource inefficiency in scale-out architectures. Consider a query that takes 100 seconds to execute on a single machine and consumes 100 joules of energy. Now consider a situation where the data corresponding to the same query is spread equally on to two machines and query is allowed to execute parallelly on these two machines.

Ideally, this query should finish its execution in exactly half of the time (i.e., 50 seconds) on each machine and should consume 50 joules energy on each machine. But in practice as a result of several overheads and process startup costs the query executes in > 50 seconds time and consumes > 50 joules of energy on each machine. In summary, a job or query executing on multiple machines can consume more energy when compared to the same query executing on relatively fewer number of machines. In other words, in the absence of super-linear speedups, *more the number of machines a job or a query touches, more energy it consumes.* In order to minimize resource consumption of given job or a query, we should minimize the number of machines required for a job or a query for its execution.

1.3 Query Span: A Key Metric to Optimize

For a given query or analysis task, its *span* is defined to be the minimum number of machines that contain the data needed to execute that query or task. Minimizing query span has significant advantages that make it an important metric for which to optimize.

Minimize the communication overhead: Query span directly impacts the total communication that must be performed to execute a query. This is clearly a concern in distributed setups (e.g., grid systems [83] or multi-datacenter deployments); however even within a datacenter, communication network is

oversubscribed, and especially cross-rack communication bandwidth can be a bottleneck [19, 37]. In cloud computing, the total communication directly impacts the total dollar cost of executing a query.

Minimize the total amount of resources consumed: It is well-known that parallelism comes with significant startup and coordination overheads, and we typically see sub-linear speedups as a result of these overheads and data skew [68]. Although the response time of a query usually decreases in a parallel setting, the total amount of resources consumed typically increases with increased parallelism. Even in scenarios where we obtain *super-linear* speedups due to higher aggregate memory across the machines, we expect the total resource consumption to increase with the degree of parallelism.

Reduce the energy footprint: Computing equipment in US costs datacenter operators millions of dollars annually for energy, and also impacts the environment. Energy costs are ever increasing and hardware costs are decreasing – as a result soon the energy costs to operate and cool a datacenter may exceed the cost of the hardware itself. Minimizing the total amount of resources consumed directly reduces the total energy consumption of a task.

Illustrative Experiments: To support these claims and to motivate query span as a key metric to optimize, we conducted a set of experiments analyzing the effect of query span on the total resource and energy consumption under a

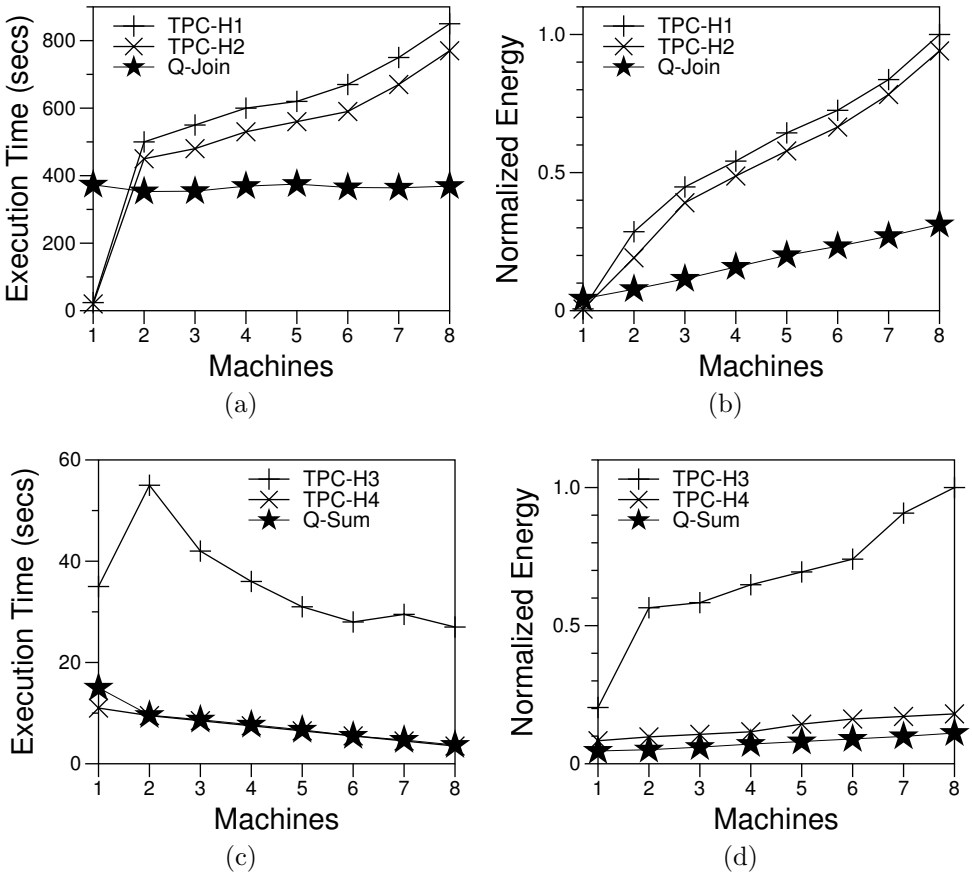


Figure 1.1: **Increasing the degree of parallelism may result in lower execution times, but leads to higher resource and energy consumption.**

variety of settings. First setting is a horizontally partitioned MySQL cluster, on which we execute four SQL queries against a TPC-H database. Two of the queries are complex analytical join queries (TPC-H1, TPC-H2 in Figure 1.1), whereas the other two are simple aggregation queries on a single table (TPC-H3, TPC-H4). In the second setting, we implemented our own distributed query processor on the top of multiple MySQL instances running on a cluster where predicate evaluations are pushed on to the individual nodes and data is

shipped to a single node for perform the final steps. On this setup we evaluate two queries: a complex join query (Q-Join) and a simple aggregate query on a single table (Q-Sum). In Figures 1.1a and 1.1b, we plot the execution times and the energy consumed as the number of machines across which the tables are partitioned (and hence query span) increases. The energy consumption is estimated by using an Itanium server power model constructed by using the Mantis full-system power modeling technique [29]. We use the *dstat* tool to collect various system performance counters such as CPU utilization, network reads and writes, I/O, and memory footprint, and then use the power model to estimate the total energy consumed.

As we can see, the execution times of the TPC-H queries run on MySQL cluster actually increased with parallelism, which may be because of nested loop join implementation in MySQL cluster (a known problem that is being fixed). In our implementation, the execution time remains constant. But in all cases, energy consumption increased with query span. In the second experiment with simpler queries (Figures 1.1c and 1.1d), though execution times decrease as the query span increases, energy consumption increases in all cases. From this simple set of experiments it is evident that, as the number of machines involved in processing a query increases, total resources consumed to process the query also rise. So in order to minimize overall resource consumption of a given query/job, one should minimize number of

machines involved in a processing of a query, that is *query span*.

Best case: Given a query or a job, when the data required by a job fits in the memory of a single machine, then a job can be executed on a single machine as efficiently as possible (*query span* = 1). This results in the highest resource efficiency of the underlying system.

1.4 Central Technical Insight

In scale-out settings, an effective way to minimize the *query span* is to co-locate the data items required by the queries on fewer machines. With colocation, data items needed for a particular query or a job can be found in lesser number of machines, consuming lesser resources often also improving execution times. In order to perform data colocation certain information about data access patterns is required. For example, if we know that the data items d_1 and d_2 are being frequently accessed by the queries, then these data items can be colocated and placed in single partition. In other words, we want to understand the history of data access and partition the data items such that frequently co-accessed data items are placed together and when queries access these data items then *query span* is minimized.

In this dissertation, we develop a workload-driven approach that aims to reduce the average query span in distributed data platforms by co-locating data items that are frequently accessed together by queries. We observe that,

for fault tolerance, load balancing, and availability, those systems typically maintain several copies of each data item (e.g., Hadoop file system (HDFS) maintains at least 3 copies of each data item by default [89]), and we propose exploiting this inherent replication to achieve higher colocation by judicious replica creation and placement. Our approach is workload-driven in that, we propose capturing a historical query workload over a period of time, and optimizing data placement and replication for that workload. Our techniques work on an abstract representation of the query workload, and are applicable to both multi-site data warehouses and general purpose data centers. we represent the query workload as a *hypergraph*, where the nodes are the data items and each query is translated into a *hyperedge* over the nodes. The data items could be database relations, parts of database relations (e.g., tuples or columns), or arbitrary files. The goal is to store each data item (node in the graph) onto a subset of machines/sites (also called *partitions*), obeying the storage capacity requirements for the partitions. Note that the partitions do not have to be machines, but could instead represent racks or even datacenters. The span of a query is defined to be the smallest number of partitions that contain all the data that the query needs. Our goal is to find a layout that minimizes the average span over all queries in the workload. Further, our algorithms can optimize for load or storage constraints, or both.

1.5 Thesis Contributions

In the first part of the dissertation, we focus on the problem of minimizing resource consumption through workload-aware data partitioning and replication in the context of distributed data warehouses. Our key contributions include:

- Formulating and analyzing this problem, drawing connections to several problems studied in the graph algorithms literature, and developing efficient algorithms for data placement.
- Developing theoretical bounds for special classes of graphs that gives an understanding of the trade-off between resource consumption and storage.
- Identification of query span as having a direct and significant impact on total resource consumption, and as being an important metric to optimize for.
- A suite of novel algorithms for making data replication and placement decisions that minimize average query span for a given workload.
- Building a trace-driven simulation framework that enables one to systematically compare different algorithms, by automatically generating varying types of query workloads and by calculating the total energy

cost of a query trace.

- Conducting an extensive experimental evaluation using this framework, and results show that our techniques can result in high reductions in query spans and resource consumption compared to baseline or random data placement approaches.

In the second part of the dissertation, we solve the problem of assigning topical document clusters to physical partitions with the goal to decrease *query span* as much as possible, while minimizing load imbalance. Following are the contributions of the work presented in the second part of the dissertation:

- We propose highly effective workload-aware data partitioning techniques to minimize search cost significantly. Our approach reduces search cost up to 75% when compared to state-of-the-art distributed exhaustive and selective search.
- We propose a novel scalable techniques for partitioning extremely large and dense hypergraphs that represent query workload. Our novel hypergraph partitioning technique achieved at least 40× times speed improvement over state-of-the-art (hMETIS-based) partitioning technique in our experimental study.
- We develop load- and search cost-aware replication and routing tech-

niques to minimize load imbalance and to opportunistically optimize search cost. Our novel load-aware set cover routing technique gracefully scales even when the partitions are subjected to tight load constraints.

- We implement an incremental repartitioning technique that repartitions small subset of data to decrease in load variations caused by change in workload while carefully trading the search cost. In the experimental study, our workload-aware incremental repartitioning technique achieved at max 171× times more effectiveness over the smart baseline (discussed in Section 4.3.3).

Lastly, in the final part of the dissertation, we present a prototype runtime called HONE that is intended to be API compatible with standard (distributed) Hadoop, that allows us to take an algorithm implemented in Hadoop and find the most suitable runtime environment for execution on datasets of varying sizes—if the data fits into RAM, we can avoid network latency and significantly decrease execution time in a shared-memory environment. Contributions of our work are:

- HONE is a scalable MapReduce implementation for multi-core, shared-memory machines. To our knowledge it is the first MapReduce implementation that is both Hadoop API compatible and optimized for scale-up architectures.

- We propose and evaluate different approaches to implementing the data shuffling stage in MapReduce, which is critical for achieving high performance.
- We discuss key challenges in implementing HONE on the JVM, how we addressed them, and lessons we learned along the way.
- We evaluate HONE on a number of real-world applications, comparing it to Hadoop pseudo-distributed mode, a 16-node Hadoop cluster, and several systems proposed in prior work for single-machine analytics.
- We propose a synthetic workload generator for evaluating HONE that may be of independent interest for evaluating other systems.

Chapter 2

Related Work

2.1 Data Colocation for Resource Minimization

First part of this thesis deals with data colocation and resource minimization issues in various data domains (Chapter 3 and Chapter 4). There are a vast number of papers that relate to these issues separately. In the following subsections we discuss several related papers in the various subareas that are related to our work.

2.1.1 Data Partitioning and Replication

Data partitioning and replication plays an increasingly important role in large scale distributed networks such as content delivery networks (CDN), distributed databases and distributed systems such as peer-to-peer networks. Recent work [3, 26] has shown that judicious placement of data and replica-

tion improves the efficiency of query processing algorithms. There has been some recent interest in improving data co-location in large scale processing systems like Hadoop. Recent work by Eltabakh et al. [30] on CoHadoop is very close to our work, where they provide an extension for Hadoop with a lightweight mechanism that allows applications to control where data is stored. They focus on data colocation to improve the efficiency of many operations, including indexing, grouping, aggregation, columnar storage, joins, and sessionization. Our techniques are complementary to their work.

Hadoop++ [26] is another closely related work that exploits data pre-partitioning and co-location. There is substantial work on replica placement that focuses on minimization of network latency and bandwidth. Neves et al. [65] propose a technique for replication in CDN where they replicate data onto a subset of servers to handle requests so that the traffic cost in the network is minimized. There has been a lot of work on dynamic/adaptive replica management (e.g., [34, 72, 73, 79, 90, 94]), where replicas are dynamically placed, moved, or deleted based on the read/write access frequencies of the data items again with the goal of minimizing bandwidth and access latency.

Our work is different from several other works on data placement [46, 59, 66] where the database query workload is also modeled as a hypergraph and partitioning techniques are used to drive data placement decisions. Tosun et

al. [86, 87] and Ferhatosmanoglu et al. [32] propose using replication along with declustering for achieving optimal parallel I/O for spatial range queries. The goal of most prior work is typically minimization of query latencies by spreading out the work over a large number of partitions or devices. For us, that is exactly the wrong optimization goal – we would like to cluster data required for each query on as few partitions as possible.

Graphs have been used as a tool to model various distributed storage problems and to come up with replication strategies to achieve a specific objective. Du et al. [28] study Quality-of-Service (QoS)-aware replica placement problem in a general graph model. In their model, vertices are the servers with various weights representing node characteristics and edges representing the communication costs. Other work has modeled network topology as a graph and developed replication strategies or approximations (replica placement in general graphs is NP-complete) [91]. In contrast, we model query workload as a hypergraph, and assume a uniform network topology (i.e., identical communication costs between any pair of nodes); we believe this better approximates current networks.

2.1.2 Graph Algorithms

The problems we study are closely related to several well-studied problems in graph theory and can be considered generalizations of those problems.

A basic special case of our main problem is the *minimum graph bisection* problem (which is NP-Hard), where the goal is to partition the input graph into two equal sized partitions, while minimizing the number of edges that are cut [12]. There is much work on both that problem and its generalization to hypergraphs and to k-way partitioning [40,43,63]. Another closely related problem is that of finding *dense subgraphs* in a graph, where the goal is to find a group of vertices where the number of edges in the induced subgraph is maximized [31]. Finally, there is much work on finding *small* separators in graphs. Several theoretical results are known about this problem. We discuss these connections in more detail later when we describe our proposed algorithms.

In the context of hypergraph partitioning, there is a vast literature of work proposing variety of techniques and approaches to perform k-way balanced min-cut hypergraph partitioning [16, 25, 41, 44, 77]. Some of these works are available for public use in the form of popular softwares such as hMETIS [42], PaToH [17], SCOTCH [69] and Zoltan [24]. These systems are considered state-of-the-art and we note that although these systems provide very efficient hypergraph partitioning algorithms, none of these systems scale for very dense hypergraphs that we deal with in this work.

2.1.3 Energy Efficiency and Data Management

Issues in energy-efficient computing are being increasingly studied at all layers in today's computing infrastructures. Harizopoulos et al. [36] reported the first results on software-level optimizations to achieve better energy efficiency; they experiment with a system that was configured similarly to an audited TPC-H server and show that making the right physical design decisions can improve energy efficiency. Additionally, they use relational scan operator as a basis to demonstrate that optimizing for performance is different from optimizing for energy efficiency. It is also among the first papers [35, 36, 53] to practically show the importance of energy efficiency in database systems. Graefe [36] also points out various research challenges and promising approaches in energy-efficient database management. In his paper he indicates various promising approaches and techniques to achieve energy efficiency in database systems. He discusses two approaches in this context: processor frequency control and explicit delays. Leverich et al. [55] and Lang et al. [54] suggest approaches to conserving energy by powering down Hadoop cluster nodes during periods of low load, and observe that the default replica placement policy is highly inefficient in this regard. In particular, they observe that powering down any three nodes is likely to lead to some data being unavailable, and instead suggest a replication policy such

that a small set of cluster nodes *cover* (contain) at least one replica of each data item.

Lang et al. [54] suggest and evaluate an alternative approach where all cluster nodes are powered up (to answer queries), and powered down at the same time, and show that their approach leads to better energy utilization. The approach that we consider here is more fine-grained in that, we consider shutting down individual disks (or nodes) during periods of low load, and wake them up as needed. Tsirogiannis et al. [88] analyze the energy efficiency of a single-node database server, and argue that the most energy-efficient configuration is typically the highest performing one. However, this assertion is valid only for single node database server, and does not hold for scale-out architectures involving multiple machines where parallelization, communication, and startup overheads come into play. From our experiments over the TPC-H benchmark, it is evident that, as the number of machines involved in processing a query increases, total resources consumed to process the query also rise.

2.1.4 Distributed Search

Most prior research on web information retrieval assumes that documents are already assigned to physical partitions [8, 15, 80, 84]. Simplest way of assigning documents to partitions is to distribute these documents randomly

across the partitions. Query is sent to all the partitions to retrieve the documents of interest. This approach of distributed exhaustive search increases the query cost significantly [47]. There have been few other studies that have looked into partitioning of a document collection into topical clusters. These studies have shown that search efficiency can be further increased by document cluster selection, that is, by querying a small number of promising document clusters for each query. Kulkarni et al., [47] present the topic-based clustering and partitioning of documents into distributed index or document clusters and show that this approach reduces the search cost significantly when compared to the exhaustive search with no loss of accuracy, on average. Our work in part 2 of this dissertation (Chapter 4) is very close to this work, important difference being that, they only partition documents into topical clusters and do not discuss how to assign these topical document clusters to physical partitions. In our work, we propose an effective approach to assign document clusters to the underlying physical partitions by analyzing the search workload history. On the other hand, given a topically clustered system, Aly et al., [6] present a vocabulary-based document cluster selection algorithm that represents document clusters by statistics of terms in the vocabulary. In their work, topical partitioning is given and hence they do not deal with document partitioning or document cluster-machine assignment.

2.2 Mapreduce on Multi-core Shared Memory Systems

Next, we perform literature survey where we review various papers in the literature related to our work presented in Chapter 5 and compare them with our work on in-memory map reduce system.

A series of incremental advances on shared-memory MapReduce implementations have been presented by Kozyrakis et al. Their first system, Phoenix [74], evaluates the suitability of MapReduce as a programming environment for shared-memory systems. Phoenix2 [92] makes improvements over Phoenix system by identifying the inefficiencies in handling large-scale data. It proposes user-tunable hash table-based data structure to store intermediate keys. Phoenix++ [82] makes further improvements by observing that data structures for intermediate data storage cannot be fixed *a priori*, as they depend on the nature of the application. Thus, they provide container objects used to store map output as an abstraction to the user.

We observe several shortcomings of their approach that limits its broad applicability. First, the Phoenix systems are implemented in C++ and are not compatible with Hadoop API. Thus, scaling down Hadoop using Phoenix involves essentially a full re-implementation. Second, hash table-based containers are not a feasible solution for a Java-based implementation, especially for applications where intermediate data size is more than the input dataset

size; Java objects tend to be heavyweight, and off-of-the-shelf Java containers tend to be inefficient for large datasets. We discuss this issue in more detail in Chapter 5. In another related work, Mao et al. [61], in their system Metis, propose using containers based on hash+B-trees to store intermediate outputs. Chen et al. [18] proposed a tiled MapReduce approach to iteratively process small chunks of data at a time with efficient use of resources. Jiang et al. [39] build upon Phoenix and provide an alternate API for MapReduce. Both these systems are implemented in C++ and are not Hadoop compatible; moreover they modify the MapReduce API. Note that all the above systems that are related do not scale their system on large datasets, their datasets are mostly in the range of MBs, we evaluate our system for worst case and scale for GBs of dataset on single shared-memory high-end machine.

Shinnar et al. [78] propose a main memory map-reduce (M3R) is a new implementation of the Hadoop map-reduce API targeted at online analytics on high mean-time-to-failure clusters. Although it is close to our system, they mainly focus on scale-out architectures, whereas we focus explicitly on scaled-up single shared memory machine. They do not provide insights on single machine scalability and how does various workloads behave when processed in-memory on single machine. Moreover, M3R is implemented in X10 language, and X10 only supports concurrent constructs for only up to four cores, and nowadays it is normal to have more than eight cores on a

single machine. Single machine scalability is different from that of scalability on scale-out architectures. In our work, we mainly focus on single machine scalability and in-general distributed frameworks carry significant amount of overheads because of all the distributed parallel programming layers in the framework which limits their scalability on single machine.

Chapter 3

Minimizing Query Span through Workload-aware Data Replication and Placement

3.1 Introduction

A variety of complex analysis tasks and queries are being executed today using parallel and distributed data management systems like parallel databases and MapReduce-based systems (e.g., Hadoop). In parallel databases, the queries typically consist of multiple joins, group-bys on multiple attributes, and complex aggregations. On Hadoop, the tasks often have similar flavor, with simplest of MapReduce programs being aggregation tasks that form the basis of analysis queries. There have also been many attempts

to combine the scalability of Hadoop and declarative querying abilities of relational databases [67, 85].

Use of such parallel or distributed frameworks is expected to accelerate in the coming years, putting further strain on already-scarce resources like compute power, network bandwidth, and energy. For reducing total execution times, there is a trend towards increasing the execution parallelism by spreading out data across a large number of machines. As we discussed in the Chapter 1 and experimentally illustrated, the trend toward increasing parallelism often increases the total resource consumption significantly. We argue that, for most analytical workloads, minimizing the query[1] latencies may not be critically important since the queries are often not run in an interactive mode. Instead, we argue that we should aim for reducing the total resource consumption by *decreasing* the degree of single-query execution parallelism, i.e., by trying to reduce the number of machines involved in the execution of a query (called *query span*). Minimizing query span, in most cases, directly leads to reduction in the total amount of resources consumed.

In this chapter, we address the problem of minimizing the average query span for a query workload through judicious replica selection (by choosing which data items to replicate and how many times), and data placement. Our

[1]We use the term *query* to denote both SQL queries and analysis tasks written using map-reduce or analogous frameworks.

techniques work on an abstract representation of the query workload, and are applicable to both multi-site data warehouses and general purpose data centers. We assume that a query workload trace is provided that lists the data items that need to be accessed to answer each query. The data items could be database relations, parts of database relations (e.g., tuples or columns), or arbitrary files. We represent such a workload as a *hypergraph*, where the nodes are the data items and each query is translated into a *hyperedge* over the nodes. The goal is to store each data item (node in the graph) onto a subset of machines/sites (also called *partitions*), obeying the storage capacity requirements for the partitions. Note that the partitions do not have to be machines, but could instead represent racks or even datacenters. This specifies the layout completely. The cost for each query is defined to be smallest number of partitions that contain all the data the query needs. Our goal is to find a layout that minimizes the average cost over all queries. Our algorithms can optimize for load or storage constraints, or both.

Our key contributions include formulating and analyzing this problem, drawing connections to several problems studied in the graph algorithms literature, and developing efficient algorithms for data placement. In addition, we examine the special case when each query accesses at most two data items – in this case the hypergraph is simply a graph. For this case, we are able to develop theoretical bounds for special classes of graphs that gives an

understanding of the trade-off between energy cost and storage.

Significant work has been done on the converse problem of minimizing query response times or latencies. *Declustering* refers to the approach of leveraging parallelism in the partition subsystem by spreading out blocks across different partitions so that multi-block requests can be executed in parallel. In contrast, we try to cluster data items together to minimize the number of sites required to satisfy a complex analytical query.

We have also built a trace-driven simulation framework that enables us to systematically compare different algorithms, by automatically generating varying types of query workloads and by calculating the total energy cost of a query trace. We conducted an extensive experimental evaluation using our framework, and our results show that our techniques can result in high reduction in query span compared to baseline or random data placement approaches that can help minimize distributed overheads.

3.2 Problem Definition; Analysis

We begin with a formal definition of the core problem of data partitioning with replication, at an abstract level. We then draw connections to some closely related prior work on graph algorithms. We also analyze a special case of the problem formally, and show an interesting theoretical result.

Problem Definition: We are given a set of data items $\mathcal{D}$ and their sizes –

the data items may be files, database relations, vertical or horizontal partitions of database relations, or tuples. We are also given a set of *partitions* with associated storage capacities, and an expected query workload in the form of a set of queries over the data items. The queries may be read-only queries, or update transactions. Our goal is to decide which data items to replicate and how to place them on to the partitions so as to minimize the average query span for the queries in the workload. In addition, we may be given a constraint that specifies how much each item should be minimally replicated for fault tolerance and availability. For simplicity, we assume for now that we are given a total of N identical partitions each with capacity C units, and further that the data items are all unit-sized (we will relax both these assumptions later). Clearly, the number of data items must be smaller than $N \times C$ (so that each data item can be placed on at least one partition). Further, let N_e denote the minimum number of partitions needed to place the data items (i.e., $N_e = \lceil |\mathcal{D}|/C \rceil$).

Modeling Workload: The query workload is represented as a hypergraph, $\mathcal{H} = (V, E)$, where the nodes are the data items and each (hyper)edge $e \in E$ corresponds to a query in the workload. Each hyperedge is associated with an edge weight w_e which represents the frequency of such queries in the workload. Each vertex $v \in V$ is associated with a weight w_v representing either

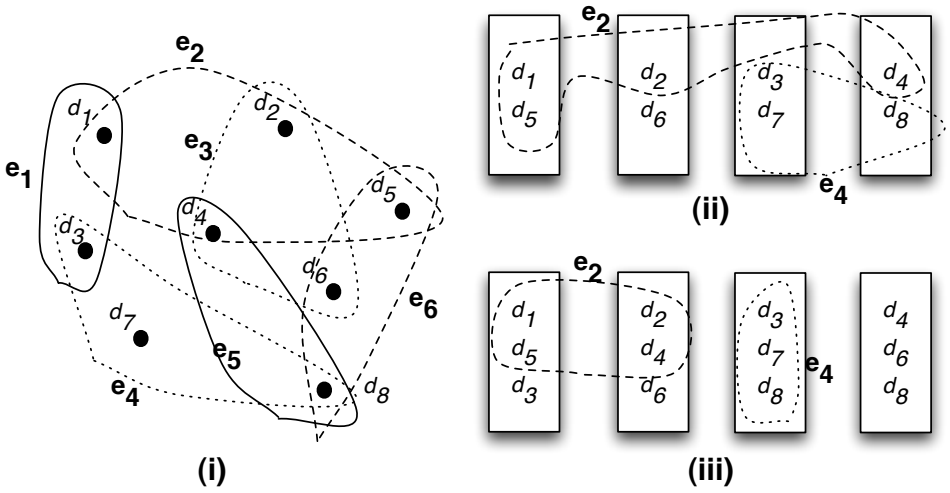


Figure 3.1: **(i) Modeling a query workload as a hypergraph – d_i denotes the data items, and e_i denotes the queries represented as hyperedges; (ii) A layout w/o replication onto 4 partitions – the span of two of the hyperedges is also shown; (iii) A layout with replication – span for both queries reduces by 1.**

data size or access frequency or a combination of both providing balancing in terms of size and load on each partition. Figure 3.1 shows an illustrative example, where we have 6 queries over 8 data items, each of which is represented as a hyperedge over the data items. The figure also shows two layouts of the data items onto 4 partitions of capacity 3 each, without and with replication.

Calculating Span: When there is no replication, calculating the span of a query is straightforward since each data item is associated with a single partition. However, if there is replication, the problem becomes NP-Hard in general. In the simplest case, for read-only queries with strongly consistent

replicas (i.e., if all replicas of a data item are kept up-to-date when it is updated), it is identical to the *minimum set cover* problem [33] – where we are given a collection of subsets of a set (in our case, the partitions) and a query subset, and we are asked to find the minimum number of subsets (partitions) required to cover the query subset.

As an example, for query e_2 in Figure 3.1, the span in the first layout is 3. However, in the second layout, assuming it is a read-only query, we have to choose which of the two copies of d_4 to use for the query. Using the first copy (on second partition) leads to the lowest span of 2. Overall, the average query span for the first layout is $\frac{13}{6}$, but use of replication in the second layout reduces this to $\frac{8}{6}$.

We can use a standard greedy algorithm for choosing replicas to use for a query and for calculating the span. For each of the partitions, we compute the size of its intersection with the query subset. We choose the partition with the highest intersection size, remove all items from the query subset that are contained in the partition, and iterate until there are no items left in the query subset. This simple greedy algorithm provides the best known approximation to the set cover problem with approximation ratio of $\log |Q|$, where $|Q|$ is the query size [31]. The algorithm can also be directly used for queries that both read and write data items with strongly consistent replicas – for data items that are updated, all replicas must be accessed, but the

greedy algorithm can be used for the items that are only read.

Hypergraph Partitioning: Without replication, the problem we defined above is essentially the k-way (balanced) hypergraph partitioning problem that has been very well-studied in the literature. However, the optimization goal of minimizing the average span is unique to this setting; prior work has typically studied how to minimize the number of *cut* hyperedges instead. Several packages are available for partitioning very large hypergraphs efficiently [1, 2]. The proposed algorithms are typically heuristics or combinations of heuristics, and most often the source code is not available. We use one such package (hMETIS) as the basis of our algorithms [1].

Finding Dense Subgraphs of a specified size: Given a set of nodes S in a graph, the *density* of the subgraph induced by S is defined to be the ratio of the number of edges in the induced subgraph and $|S|$. The dense subgraph problem is to find the densest subgraph of a given size. To understand the connection to the dense subgraph problem, consider a scenario where we have exactly one "extra" partition for replicating the data items (i.e., $N_e = N-1$). Further, assume that each query refers to exactly two data items, i.e., the hypergraph $\mathcal{H}$ is just a graph. One approach would then be to first partition the data items into $N-1$ partitions without replication, and then try to use this extra partition optimally. To do this, we can construct a *residual*

graph, which contains all edges that were cut in this partitioning. The spans of the queries corresponding to these edges is exactly 2. Now, we find the subgraph of size C such that the number of induced edges (among the nodes of the subgraph) is maximized, and we place these data items on the extra partition. The spans of the queries corresponding to these edges are reduced from 2 to 1, and hence this is an optimal way to utilize the extra partition. We can generalize this intuition to hypergraphs and this forms the basis of one of our algorithms.

Unfortunately, the problem of finding the most dense subgraph of a specified size is NP-Hard (with no good worst case approximation guarantees), so we have to resort to heuristics. One such heuristic that we adapt in our work is as follows: recursively remove the lowest degree node from the residual graph (and all its incident edges) till the size of the residual graph is exactly C. This heuristic has been analysed by Asahiro et al. [9] who find that this simple greedy algorithm can solve this problem with approximation ratio of approximately $2(\frac{|V|}{C} - 1)$ (when $C \leq |V|/3$).

Sublinear Separators in Graphs: Consider the special case where $\mathcal{H}$ is a graph, and further assume that there are only 2 partitions (i.e., $N = 2$). Further, lets say that the graph has a small *separator*, i.e., a set of nodes whose deletion results in two connected components of size at most $n/2$. In

that case, we can replicate the separator nodes (assuming there is enough redundancy) and thus guarantee that each query has span exactly 1. The key here is the existence of small separators of bounded sizes. Such separators are known to exist for many classes of graphs, e.g., for any family of graphs that excludes a minor [4].

A separator theorem is usually of the form that, any n-vertex graph can be partitioned into two sets A, B, such that $|A \cap B| = c\sqrt{n}$ for some constant c, $|A - B| < 2n/3$, $|B - A| < 2n/3$, and there are no edges from a node in $A - B$ to a node in $B - A$. This directly suggests an algorithm that recursively applies the separator theorem to find a partitioning of the graph into as many pieces as required, replicating the separator nodes to minimize the average span. Such an algorithm is unlikely to be feasible in practice, but may be used to obtain theoretical bounds or approximation algorithms. For example, we prove that:

Theorem 3.2.1 *Let G be a graph with n nodes that excludes a minor of constant size. Further, let N_e denote the number of partitions minimally required to hold the nodes of G (i.e., $N_e = \lceil n/C \rceil$). Then, asymptotically, $N_e^{1.73}$ partitions are enough to partition the nodes of G with replication so that each edge is contained completely in at least one partition.*

Proof: The proof relies on the following theorem by Alon et al. [4]:

Theorem 3.2.2 *Let G be a graph with n nodes that excludes a fixed minor with h nodes. Then we can always find a separation (A, B) such that $|A \cap B| \leq h^{\frac{3}{2}} n^{\frac{1}{2}}$, $|A - B|, |B - A| \leq \frac{2}{3}n$.*

Consider a recursive partitioning of G using this theorem. We first find a separation of G into A and B. Since A and B are subgraphs of G, they also exclude the same minor. Hence we can further partition A and B into two (overlapping) partitions each. Now, both $|A|$ and $|B|$ are $\leq \frac{2}{3}n + h^{\frac{3}{2}} n^{\frac{1}{2}}$. For large n, the second term is dominated by ϵn, for any $\epsilon > 0$. We choose some such $\epsilon = 1/300$. Then, we can write: $|A|, |B| \leq (\frac{2}{3} + \epsilon)n = 0.67n$ for large enough n.

Now we continue recursively for l steps getting us 2^l subgraphs of the original graph G, such that each of the subgraphs fits in one partition. Note that, by construction, every edge is contained in at least one of these subgraphs; thus 2^l partitions are sufficient for data placement as required. Since the partition capacities are $O(n)$, we can use the above formula to compute l. We need: $0.67^l n < C = n/N_e$. Solving for l, we get: $l > log_2(N_e^{1.73})$. Hence, the number of partitions needed to partition G with replication so that each edge is contained in at least one partition is less than $N_e^{1.73}$.

For general graphs, we show that:

Theorem 3.2.3 *If the optimal solution uses βN_e partitions to place the data*

items so that each edge is contained in at least one partition, then either we can get an approximation with factor $\frac{2}{2-\alpha}$ *for* $0 \leq \alpha \leq 1$ *using* N_e *partitions, or a placement using* $\frac{CN_e\beta}{2\alpha}$ *partitions with span 1 for each edge.*

Proof: Given a graph $G = (V, E)$ (special case when the hypergraph $\mathcal{H}$ has size two edges) – our objective is to store the data items in a collection of partitions, each of capacity C. For each edge the cost is either 1 or 2. This gives rise to a trivial 2-approximation since $|E|$ is a lower bound on the optimal solution and $2|E|$ is a trivial upper bound on the solution that picks an arbitrary layout. Note that replication is allowed, and we may store more than one copy of each data item.

Assume that there is an optimal solution that creates at least one copy of each data item – uses $N_e (= \frac{n}{C})$ partitions (for simplicity we assume that n is a multiple of C). We now prove the bound for the following method. We order the nodes in decreasing order by degree.

For each node v_i, assume that E_i is the set of edges adjacent to v_i that go to nodes v_j with $j > i$. We use N_i partitions to store v_i where in the first partition we store v_i together with its first $C - 1$ neighbors, the second partition with v_i together with its next $C - 1$ neighbors etc. We thus use $N_i = \lceil \frac{|E_i|}{C-1} \rceil$ partitions for each node v_i.

The total number of partitions used is $\sum_{i=1}^{n} N_i = \sum_{i=1}^{n} \lceil \frac{|E_i|}{C-1} \rceil$.

Now consider an optimal solution with cost OPT that stores the nodes of G using N' partitions. Note that with N' partitions, each holding C nodes, the maximum number of local edges (edges for which the optimal solution incurs a cost of 1) within each partition is at most $\frac{C(C-1)}{2}$. We thus get $|E^*| \leq N'\frac{C(C-1)}{2}$ where E^* is the set of local edges in an optimal solution. Note that $OPT = |E^*| + 2(|E| - |E^*|) = 2|E| - |E^*|$ where OPT is the cost of an optimal solution.

We first note that if $|E^*| \leq \alpha|E|$ then we get a better lower bound on OPT, namely that $OPT \geq (2-\alpha)|E|$. Thus our solution, which has cost at most $2|E| \leq \frac{2}{2-\alpha}OPT$. This gives us a good approximation when α is significantly smaller than 1.

If $|E^*| > \alpha|E|$ then we get $\alpha|E| < |E^*| \leq N'\frac{C(C-1)}{2}$. Dividing by $\alpha(C-1)$ we get $\frac{|E|}{C-1} < |E^*| \leq N'\frac{C}{2\alpha}$. Since $|E| = \sum_i |E_i|$ we get $\sum_i \frac{|E_i|}{C-1} < |E^*| \leq N'\frac{C}{2\alpha}$.

Recall that the total number of partitions we used is $\sum_{i=1}^{n} N_i = \sum_{i=1}^{n} \lceil \frac{|E_i|}{C-1} \rceil$. Ignoring the fact that we really need to take the ceiling, we can re-write this as $\sum_{i=1}^{n} \frac{|E_i|}{C-1} < N'\frac{C}{2\alpha}$. If $N' = \beta\frac{n}{C}$ for some constant β, then we get $\frac{n\beta}{2\alpha}$ as the bound on the number of partitions

We thus conclude.

3.3 Data Placement Algorithms

In this section, we present several algorithms for data placement with replication, with the goal to minimize the average query span. We continue to assume a homogeneous setup where each partition has capacity C units, and each data item has size 1; we relax both these assumptions in Section 3.3.7. Instead of starting from scratch, we chose to base our algorithms on existing hypergraph partitioning packages. As we discussed in the previous sections, the problem of balanced and unbalanced hypergraph partitioning has received a tremendous amount of attention in various communities, especially the VLSI community. Several very good packages are freely available for solving large partitioning problems [1, 2, 14, 40]. We use a hypergraph partitioning algorithm (denoted HPA) as a blackbox in our algorithms, and focus on replicating data items appropriately to reduce the average query span. An HPA algorithm typically tries to find a balanced partitioning (i.e., all partitions are of approximately equal size) that minimizes some optimization goal. Usually, allowing for unbalanced partitions results in better partitioning. In the algorithm descriptions below, we assume that the HPA algorithm can return an exactly balanced partition, where all partitions are of equal size, if needed.

Following the discussion in the previous section, we develop four classes

avgDataItemsPerQuery($\mathcal{H}$)**:** Suppose V_i is the set of data items covered by hyperedge $e_i \in \mathcal{H}$. The $\Sigma_{e_i \in \mathcal{H}} \lvert V_i \rvert / \lvert V \rvert$ gives the average number of data items covered per query.
getSpanningPartitions($\mathcal{G}, e$)**:** Given the current placement (during the course of the algorithm) and a hyperedge e, find a minimal subset of partitions $MD_e \subseteq \mathcal{G}$, such that every node in e is contained in at least one partition in MD_e. We use the set cover-based algorithms presented in Section 4.2.1 and Section 5 as appropriate.
getQuerySpan($\mathcal{G}, e$)**:** Similar to above, but we only return the size of the minimal subset of spanning partitions.
getAccessedItems($\mathcal{G}, e, g \in \mathcal{G}$)**:** Given a current placement $\mathcal{G} = \{G_1, \cdots, G_N\}$, a hyperedge e and a partition $g \in \mathcal{G}$, this returns the set of items that the query corresponding to e would access from partition g, as computed by the greedy Set Cover algorithm. This may be empty even if $e \cap g \neq \phi$.
pruneHypergraphBySpan($\mathcal{G}, \mathcal{H}$, $minSpan$)**:** Given a current placement $\mathcal{G}$ and a value of $minSpan$, this routine removes all hyperedges from $\mathcal{H}$ with span less than or equal to $minSpan$ and any nodes with 0 incident hyperedges.
getKDensestNodes($\mathcal{H}, K$)**:** Given a hypergraph $\mathcal{H}$, this procedure returns a dense subgraph containing nodes having total weight of atmost K. We use the greedy algorithm described in the previous section.
pruneHypergraphToSize($\mathcal{H}, K$)**:** Given a current placement $\mathcal{G}$ and a value of K, this routine uses the same algorithm as for getKDensestNodes to find a (dense) hypergraph over nodes having total weight of K.
totalWeight(V)**:** Given a set of vertices V, return the total weight of vertices.
getHittingSet(MD_e)**:** This procedure takes the minimal subset of partitions MD_e returned by getSpanningPartitions as input. Now, for hyperedge e_{d_i} that is incident on a data item d_i, let G_{d_i} denote the set of partitions that e_{d_i} spans. The procedure returns a set of partitions, S, such that each of G_{d_i} contains at least one partition from this set (i.e., $S \cap \mathcal{G}_{d_i} \neq \phi$).

Table 3.1: **Description of subroutines used by the algorithms.**

of algorithms:

- **Iterative HPA (IHPA)**: Here we repeatedly use HPA until all the extra space is utilized.
- **Dense Subgraph-based (DS)**: Here we use a dense subgraph finding algorithm to utilize the redundancy.
- **Pre-replication (PR)**: Here we identify a set of nodes to replicate a priori, modify the input graph by replicating those nodes, and then run HPA to get a final placement.
- **Local Move-based (LM)**: Starting with a partitioning returned by HPA, we improve it by replicating small groups of data items at a time.

As expected the space of different variants of the above algorithms is very large. We experimented with many such variants in our work. We begin with a brief listing of some of the key subroutines that we use in the pseudocodes. We then describe a representative set of algorithms that we use in our performance evaluation.

3.3.1 Preliminaries; Subroutines

The inputs to the data placement algorithm are: (1) the hypergraph, $\mathcal{H}(V, E)$, with vertex set V and (hyper)edge set E that captures the query workload, and (2) the number of partitions, N and (3) the capacity of each

partition C. We use N_e to denote the minimum number of partitions needed to partition the hypergraph ($N_e \leq N$).

Our algorithms use a hypergraph partitioning algorithm (HPA) as a blackbox. HPA takes as input the hypergraph to be partitioned, the number of partitions, and an *unbalance factor* (UBfactor). The unbalance factor is set so that HPA has the maximum freedom, but the number of nodes placed in any partition does not exceed C. For instance, if $|V| = N_e \times C$ and if HPA is asked to partition into N_e partitions, then the unbalance factor is set to be the minimum. However, if HPA is called with $N' > N_e$ partitions with $H'(V', E')$ as the hypergraph, then we appropriately set the unbalance factor such that maximum imbalance does not exceed C. Specifically:

$$UBfactor = \frac{N' - N_e}{N_e} \tag{3.1}$$

Finally, we modify the output of HPA slightly to ensure that the partition capacity constraints are not violated. This is done as follows: if there is a partition that has higher than maximum number of nodes, we move a small group of nodes to another partition with fewer than maximum number of nodes. We use one of our algorithms developed below (LMBR) for this purpose.

In the pseudocodes shown, apart from HPA, we also assume existence of the subroutines shown in Table 3.1. We note that, because of the modularized way our framework is designed, we can easily use different, more efficient algorithms for solving these subproblems.

3.3.2 Iterative HPA (IHPA)

Here, we start by using HPA to get a partitioning of the data items into exactly N_e partitions (recall that N_e is the minimum number of partitions needed to store the data items). We then prune the original hypergraph $\mathcal{H}(V, E)$ to get a residual hypergraph $\mathcal{H}^{'}(V^{'}, E^{'})$ as follows: we remove all hyperedges with low spans (specifically with span less than the average number of data items in a query) and we then remove all the data items that are not contained in any hyperedge. If the number of nodes in $\mathcal{H}'$ is less than $(N - N_e)C$ (i.e., if the data items fit in the remaining empty partitions), we apply HPA to obtain a balanced partitioning of $\mathcal{H}'$ and place the partitions on the remaining partitions. This process is repeated if there are still empty partitions (Algorithm 1).

If the number of nodes in $\mathcal{H}'$ is larger than the remaining capacity, we prune the graph further by removing the hyperedges with the lowest span one at a time (these hyperedges are likely to see the least improvement by replication) and the data items that now have 0 degree, until the number of

Algorithm 1 Iterative HPA (IHPA)

Require: $\mathcal{H}(V, E), N, C$
1: Run HPA($\mathcal{H}, N_e$) to get an initial partitioning: $\mathcal{G} = \{G_1, G_2, \ldots, G_{N_e}\}$;
2: $edgeCost$ = avgDataItemsPerQuery($\mathcal{H}$);
3: **while** $edgeCost \neq 0$ **and** $|\mathcal{G}| \neq N$ **do**
4: $\mathcal{H}^{'}(V', E')$ = pruneHypergraphBySpan($\mathcal{G}, \mathcal{H}, edgeCost$);
5: $N_{cur} = totalWeight(V^{'})/C$;
6: **if** $|\mathcal{G}| + N_{cur} \leq N$ **and** $|\mathcal{H}^{'}| \neq 0$ **then**
7: $\mathcal{G} = \mathcal{G} \cup$ HPA($\mathcal{H}^{'}, N_{cur}$);
8: **else if** $|\mathcal{G}| + N_{cur} > N$ **then**
9: **increment** $edgeCost$ **by** 1;
10: **else**
11: **decrement** $edgeCost$ **by** 1;
12: **return** final partitions $G_1, G_2, \cdots, G_N$

nodes in $\mathcal{H}'$ becomes sufficiently low; then we apply HPA to obtain a balanced partitioning of $\mathcal{H}'$ and place the partitions on the remaining partitions.

IHPA, as described above, assumes that the cost of replicating any data item is the same. There is unfortunately no natural way to modify the algorithm to allow for different costs to replicate different items, say because they have different write frequencies (aside from removing any data items with write frequencies above a threshold from the residual graph). Hence, we only use IHPA for read-heavy workloads.

3.3.3 Dense Subgraph-based (DS)

This algorithm directly follows from the discussion in the previous section. As above, we use HPA to get an initial partitioning. We then fill the remaining $N - N_e$ partitions one at a time, by identifying a dense subgraph

Algorithm 2 Dense Subgraph-based (DS)

Require: $\mathcal{H}(V, E), N, C, minSpan(default = 1)$
1: Run HPA($\mathcal{H}, N_e$) to get an initial partitioning: $\mathcal{G} = \{G_1, G_2, \ldots, G_{N_e}\}$;
2: **while** $|\mathcal{G}| \neq N$ **do**
3: $\mathcal{H}^{'}$ = pruneHypergraphBySpan($\mathcal{G}, \mathcal{H}, minSpan$);
4: **if** $|\mathcal{H}'| = 0$ **break**;
5: $denseNodes$ = getKDensestNodes($\mathcal{H}^{'}, C$);
6: Add a partition containing $denseNodes$ to $\mathcal{G}$;
7: **return** final partitions $G_1, G_2, \cdots, G_N$

of the residual hypergraph $\mathcal{H}'$. This is done by removing the lowest degree nodes and their incident hyperedges from $\mathcal{H}'$ one-by-one until the number of nodes in it reaches C (the partition capacity). These data items are then placed on one of the remaining partitions, and the procedure is repeated until all partitions are utilized (Algorithm 2). The hyperedges in the densest subgraph found in one iteration thus have span = 1, and are not part of the residual hypergraph in the next iteration. Like IHPA, DS cannot be easily modified to handle variable replication costs, and is more suitable for read-heavy workloads.

3.3.4 Pre-Replication-based Algorithm (PRA)

This algorithm is based on the idea of identifying small separators and replicating them. However, we do not directly adapt the recursive algorithm described in Section 4.2.1 for two reasons. First, since we have a fixed space budget for replication, we must somehow distribute this budget to the various stages. More importantly, the basic algorithm of bisecting a graph and

then recursing is not considered a good approach for achieving good partitioning [43, 81].

We instead propose the following algorithm. We start with a partitioning returned by HPA, and identify "important" nodes such that by replicating these nodes, the average query span would be reduced the most. Then, we create a new hypergraph by replicating these nodes (until we have enough nodes to fill all the partitions), and run HPA once again to attain a final partitioning. However, neither of these steps is straightforward.

Identifying Important Nodes:The goal is to decide which nodes will offer the most benefit if replicated. We start with a partitioning obtained using HPA, and then analyze the partitions to decide on this. We describe the intuition first. Consider a node a that belongs to some partition G_i. Now count the number of those hyperedges that contain a but do not contain any other node in G_i; we denote this number by $score_a$. If this number is high, then the node is a good candidate for replication since replicating the node is likely to reduce the query spans for several queries. We use the partitioning returned by HPA to rank all the nodes in the decreasing order by this count, and then process the nodes one at a time.

Replicating Important Nodes: Let d be the node with the highest value of $score_d$ among all nodes. We now have to decide how many copies of d

Algorithm 3 Pre-replication-based Algorithm (PRA)

Require: $\mathcal{H}(V, E), N, C$

Run HPA to get an initial partitioning into N_e partitions: $\mathcal{G} = \{G_1, G_2, \ldots, G_{N_e}\}$;
for $v \in V$ **do**
  let v be contained in partition G_v;
  compute $score_v = |\{e \in E \mid e \cap G_v = \{v\}\}|$;
$H^r = H$;
for $v \in V$ in decreasing order by $score_v$ **do**
  $E_v = \{e \in E \mid v \in e\}$;
  $G_v = \{\text{getSpanningPartitions}(\mathcal{G}, e) \mid e \in E_v\}$;
  $S = \text{getHittingSet}(G_v)$;
  for $g \in S$ **do**
    $copy_g = \text{makeNewCopy}(v)$;
    for $e \in E_v$ **s.t.** $g \in$getSpanningPartitions$(\mathcal{G}, e)$ **do**
      $e = e - \{v\} + \{copy_g\}$;
$\mathcal{G} = \text{HPA}(\mathcal{H}^r, N)$;
return final partitions $G_1, \cdots, G_N$

to create, and more importantly, which copies to assign to which hyperedge. Figure 3.2(ii) illustrates the problems with an arbitrary assignment. Here we replicate the node d to get one more copy d', and then we assign these two copies to the hyperedges e_1, e_2, e_3, e_4 as shown (i.e., we modify some of the hyperedges to remove d and add d' instead). However, the assignment shown is not a good one for a somewhat subtle reason. Since e_1 and e_3 (which are assigned the original d) do not share any other nodes, it is likely that they will span different sets of partitions, and one of them is likely to still pay a penalty for node d. On the other hand, the assignment shown in Figure 3.2(iii) is better because here the copies are assigned in a way that would reduce the average query span.

We formalize this intuition in the following algorithm. For node d, let $E_d = \{e_{d_1}, e_{d_2}, \cdots, e_{d_k}\}$ denote the set of hyperedges that contain d. For hyperedge e_{d_i}, let $\mathcal{G}_{d_i}$ denote the set of partitions that e_{d_i} spans. We then identify a set of partitions, S, such that each of $\mathcal{G}_{d_i}$ contains at least one partition from this set (i.e., $S \cap \mathcal{G}_{d_i} \neq \phi$). Such a set is called a "hitting set". We then replicate d to make a total of $|S|$ copies. Finally, we assign the copies to the hyperedges according to the hitting set, i.e., we uniquely associate the copies of d with the members of S, and for a hyperedge e_{d_i}, we assign it a copy such that the associated element from S is contained in $\mathcal{G}_{d_i}$ (if there are multiple such elements, we choose one arbitrarily).

The problem of finding the smallest hitting set is NP-Hard. We use a simple greedy heuristic. We find the partition that is common to the maximum number of sets $\mathcal{G}_{d_i}$, include it in the hitting set, remove all sets that contain it, and repeat. Algorithm 3 depicts the pseudocode for this technique.

3.3.5 Local Move Based Replication (LMBR)

Finally, we consider algorithms based on local greedy decisions about what to replicate, starting with a partitioning returned by HPA. For simplicity and efficiency, we chose to employ moves involving two partitions. More specifically, at each step, we copy a small group of data items from one

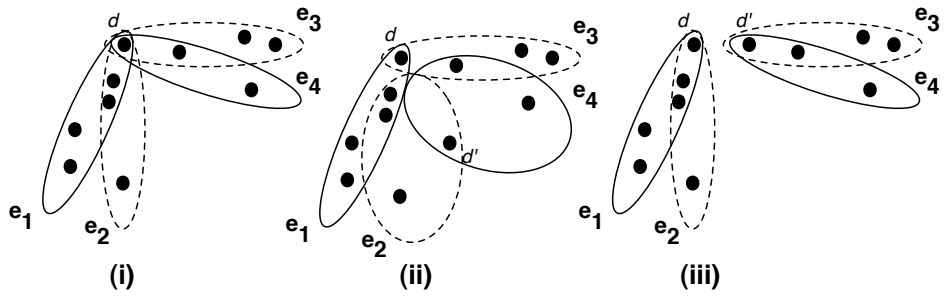


Figure 3.2: **When replicating a node, distribution of the copies to the hyperedges must be done carefully. Distribute the replica copies such that it results in entanglement of the incident hyperedges.**

partition to another. The decisions are made greedily by finding the move that results in the highest decrease in the average query span ("benefit") per data item copied ("cost"). For this purpose, at all times, we maintain a priority queue containing the best moves from $partition_i$ to $partition_j$, for all $i \neq j$. For two partitions $partition_i, partition_j$, the best group of data items to be copied from $partition_i$ to $partition_j$ is calculated as follows. Let $E_{ij} = \{e_{ij_1}, \cdots, e_{ij_l}\}$ denote the hyperedges that contain data items from both the partitions. We construct a hypergraph $H_{i \to j}$ on the data items of $partition_i$ as follows: for every edge e_{ij_k}, we add a hyperedge to $H_{i \to j}$ on the data items common to e_{ij_k} and $partition_i$. Figure 3.3 illustrates this with an example.

Now, if we were to copy a group of data items X from $partition_i$ to $partition_j$, the resulting decrease in total span (across all edges) is exactly the number of hyperedges in $H_{i \to j}$ that are completely contained in X. Thus,

the problem of finding the best move from $partition_i$ to $partition_j$ is similar to the problem of finding a dense subgraph, with the main difference being that, we want to minimize the cost/benefit ratio and not maximize the benefit alone. Hence, we modify the algorithm for finding dense subgraph as follows. We first compute the cost/benefit ratio for the entire group of nodes in $H_{i \to j}$. The cost is set to ∞ if the number of nodes to be copied is more than the empty space in $partition_j$. We then remove the lowest degree node from $H_{i \to j}$ (and any incident hyperedges), and again compute the cost/benefit ratio. We pick the group of items that results in the lowest cost/benefit ratio.

After finding the best moves for every pair of partitions, we choose the overall best move, and copy the data items accordingly. We then recompute the best moves for pairs that were affected by this move (i.e., pairs containing the destination partition), and recurse until all partitions are full.

Improved LMBR: Although the above looks like a reasonable algorithm, it did not perform very well in our first set of experiments. As described above, the algorithm has a serious flaw. Going back to the example in Figure 3.3, say we chose to copy data item d_6 from $partition_1$ to $partition_2$. In the next step, the same move would still rank the highest. This is because the construction of hypergraph $H_{1 \to 2}$ is oblivious to the fact that d_6 is also now present in $partition_2$. Further, it is also possible that, because of replication,

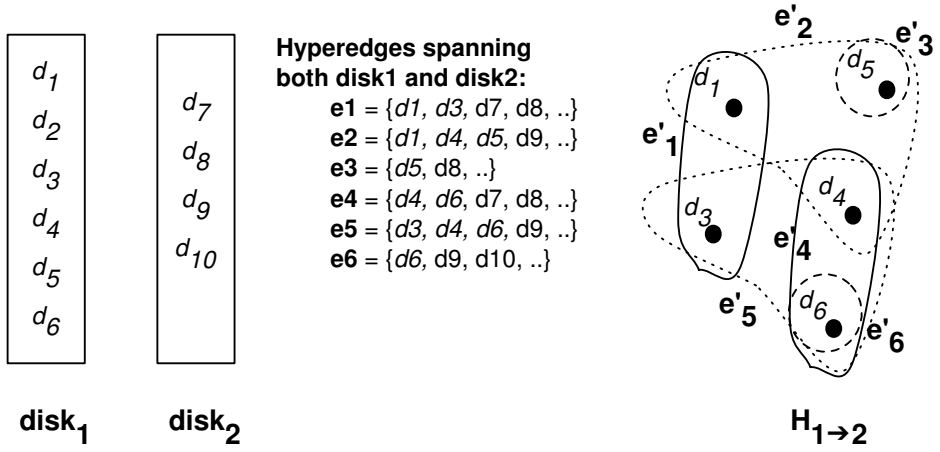


Figure 3.3: **Constructing $H_{1\to 2}$: e.g., corresponding to hyperedge e_1 that spans both partitions, we have a hyperedge e'_1 over d_1 and d_3.**

neither of the partitions is actually accessed at all when executing the queries corresponding to e_4, e_5 or e_6. To handle these issues, during the execution of the algorithm, we maintain the exact list of partitions that would be activated for each query; this is calculated using the Set Cover algorithm described in Section 4.2.1. Now when we consider whether to copy a group of items from $partition_i$ to $partition_j$, we make sure that the benefit reflects the actual query span reduction given this mapping of queries to partitions. Pseudocode for this algorithm is given in Algorithms 4 and 6.

3.3.6 k-Way Replication Algorithms

For fault tolerance, load balancing, and availability, many data management systems usually keep several copies of each data item (e.g., Hadoop file system (HDFS) maintains at least 3 copies of each data item by default [89]).

Algorithm 4 Improved LMBR

Require: $\mathcal{H}(V, E), N, C$

1: Run HPA to get initial partitions $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$ into N partitions;
2: Compute the set cover MD_e for each query e;
3: Initialize PQ (priority queue) to empty;
4: **for** $g = G_1$ to G_N **do**
5: **for** $g' = G_1$ to $G_N, g \neq g'$ **do**
6: PQ.insert($g \rightarrow g'$, maxGain($\mathcal{G}, g, g'$));
7: **while** *all partitions are not full* **do**
8: $(g_{src} \rightarrow g_{dest})$ = PQ.bestMove();
9: **copy** appropriate items from g_{src} to g_{dest};
10: **for** $g = G_1$ to G_N, $g \neq g_{dest}$ **do**
11: PQ.update($g \rightarrow g_{dest}$, maxGain($\mathcal{G}$, g, g_{dest}));
12: PQ.update($g_{dest} \rightarrow g$, maxGain($\mathcal{G}$, g_{dest}, g));
13: **return** final partitions $G_1, \cdots, G_N$;

Here we briefly discuss how the algorithms described above can be modified to handle k-way replication, where each data item must be replicated exactly k times.

PRA-Based k-Way Replication: We identify PRA as the most suitable algorithm to do this effectively, and modify PRA as follows. Considering that we are interested in replicating all the nodes k-way, we eliminate the step of finding important nodes from PRA and we replicate each node k times.

Simple Distribution Algorithm: In this algorithm, for each node d in the hypergraph we find the set of incident hyperedges E_d. We assign k copies of d among $|E_d|$ edges randomly, by assigning every $\frac{|E_d|}{k}$ hyperedges single copy of d. Only difference between this algorithm and PRA-based k-way replication algorithm is that PRA-based algorithm makes best effort to distribute the

Algorithm 5 Improved LMBR maxGain Method

Require: $\mathcal{G} = \{G_1, \cdots, G_N\}, \mathcal{H}(V, E), G_{src} \in \mathcal{G}, G_{dest} \in \mathcal{G}$

1: $E_{src} = \{e \in E \mid \text{getAccessedItems}(\mathcal{G}, e, G_{src}) \neq \phi\}$;
2: $E_{dest} = \{e \in E \mid \text{getAccessedItems}(\mathcal{G}, e, G_{dest}) \neq \phi\}$;
3: $E = E_{src} \cap E_{dest}$;
4: **if** $|E| \neq 0$ **then**
5: $V' = \cup_{e \in E}$ getAccessedItems$(\mathcal{G}, e, G_{src})$;
6: $E' = \{\text{getAccessedItems}(\mathcal{G}, e, G_{src}) | e \in E\}$;
7: **create hypergraph** $\mathcal{H}'(V', E')$;
8: $C_{dest} = C - |G_{dest}|$;
9: **if** $C_{dest} \neq 0$ **then**
10: $H' = \text{pruneHypergraphToSize}(H', C_{dest})$;
11: **while** $|H'| > 0$ **do**
12: **compute** gain $= |E'|/|V'|$
13: **remove** lowest degree node from H' and incident edges;
14: **return** the best value of gain found in the process and the corresponding V';

copies of node d among incident hyperedges E_d.

IHPA-Based Algorithm: In IHPA for k-way replication we run HPA to get partitioning without replication. We remove all the hyperedges with span 1 from the input graph, and run HPA again on the residual graph to get additional partitions. We repeat this process $k-2$ more times to replicate each node exactly k times.

3.3.7 Discussion

We presented four heuristics for data placement with replication. There are clearly many other variations of these algorithms, some of which may work better for some inputs, that can be implemented quickly and efficiently using our framework and the core operations that it supports (e.g., finding

dense subgraphs). In practice, taking the best of the solutions produced by running several of these algorithms would guarantee good data placements.

Further, while describing the algorithms, we assumed a homogeneous setup where all partitions are identical and all data items have equal size. We have also extended the algorithms to the case of heterogeneous data items. The hMETIS package that we use, and also other hypergraph partitioning packages, allow the nodes to have weights. For heterogeneous case the dense subgraph algorithm is modified to account for the weights, by removing the node with the lowest value of degree till we have nodes having total specified weight (for both DS and LMBR). Similarly, PRA is modified by allowing the replication in the original hypergraph such that total weight of replicated nodes is no greater than the sum of all extra available partition capacities.

Finally, we note that for read-write workloads, where data item replicas are generated based on read-write frequencies of the data items, PRA is the best suited algorithm as it is capable of assigning the custom number of replicas per data item to incident hyperedges in a systematic way. On the other hand, for read-only workloads any of the proposed algorithms can be used as per scalability and partitioning quality requirements of the application domain.

3.4 Experimental Setup

Our experiments are performed on Amazon EC2 medium instances. Each EC2 medium instance has Intel Xeon 64 bit processor, 3.75 GB memory with 410 GB storage. We use MySQL cluster 7.0 and MySQL 5.5 as the database servers. We conduct two experiments on analytical data stores:

First, we conducted a set of experiments on TPC-H dataset with scale factor 25, analyzing the effect of query span on the total amount of resources consumed, and the total energy consumed, under a variety of settings. We performed this experiment on 20 Amazon EC2 medium instances. We use the same two settings and the same set of queries that we used in the experiments presented in Section 1.3 in Chapter 1. The first setting is a horizontally partitioned MySQL cluster, where we evaluate two complex analytical join queries (TPC-H1, TPC-H2), and two single-table aggregate queries (TPC-H3, TPC-H4), on a TPC-H dataset. The second setting is a homegrown distributed query processor that sits atop multiple MySQL instances running on a cluster where predicate evaluations are pushed on to the individual nodes and data is shipped to a single node to perform the final steps. We evaluate a complex join query (Q-Join) and a single-table aggregate query (Q-Sum) on that setup.

Second, we evaluate effectiveness of our proposed algorithms by building

a trace-driven simulator to experiment with different data placement policies. The simulator instantiates a number of partitions as needed by the experimental setup, uses a data placement algorithm for distributing the data among the partitions, and replays a query trace against it to measure the query span profiles.

Datasets: We conducted an extensive experimental study to evaluate our algorithms, using several real and synthetic datasets. Specifically, we used the following three datasets:

- Random: Instead of generating a query workload completely randomly, we use a different approach to better understand the structure of the problem. We first generate a random *data item graph* of a specified density (edges to nodes ratio). We then randomly generate queries such that the data items in the query form a connected subgraph in the data item graph. For low density data item graphs, this induces significant structure in the query workload that good data placement algorithms can exploit for better performance.

- Snowflake: This is a special case of the above where the data item graph is a tree. This workload attempts to mimic a standard SQL query workload. We treat each column of each relation as a separate data item. An SQL query over such a schema that does not contain a Cartesian product

corresponds to a connected subgraph in this graph.

- ISPD98 Benchmark Data Sets: In addition to the above synthetic datasets, we tested our algorithms on standard ISPD98 benchmarks [5]. ISPD98 is a standard VLSI circuit hypergraph benchmark, used heavily in VLSI community for evaluating performance of hypergraph partitioning algorithms. ISPD98 circuit benchmark suite contains 18 circuits ranging from 12,752 to about 210,000 nodes. Hypergraph density (hyperedges to nodes ratio) in all the ISPD98 circuit benchmarks is close to 1, i.e., these graphs are quite sparse. We show results for the first 10 circuit datasets, that contain 12,752 to 69,429 nodes.

We compare the performance of six algorithms: (1) **Random**, where the data is replicated and distributed randomly, (2) **HPA**, the baseline hypergraph partitioning algorithm, (3-6) the four algorithms that we propose, **IHPA, PRA, DS,** and **LMBR** (Section 3.3). We use the hMETIS hypergraph partitioning algorithm [1, 40] as our HPA algorithm. All plotted numbers (except the numbers for the ISPD98 benchmark) are averages over 10 random runs. For reproducibility, we list the values of the remaining hMETIS parameters: $Nruns = 20$, $CType = 2$, $RType = 1$, $VCycle = 1$, $Reconst = 1$, $dbglvl = 0$.

The key parameters of the dataset that we vary are: (1) $|D|$, the number

of data items, (2-3) *minQuerySize* and *maxQuerySize*, the bounds on the query sizes that are generated, (4) *NQ*, the number of queries, (5) *C*, the partition capacity, (6) *numPartitions (NPar)*, the number of partitions, and (7) *density* of the data item graph (defined to be the ratio of the number of edges to the number of nodes). The default values were: |D| = 1000, minQuerySize = 3, maxQuerySize = 11, NQ = 4000, C = 50, NPar = 40, and density = 20.

In several of the plots, we also show the average number of data items per query, denoted *ADI*.

3.4.1 Query Span and Resource Consumption

To compare the cost of our best colocation scheme LMBR, we run around 10000 additional queries with TPC-H1, TPC-H2, TPC-H3, TPC-H4 , Q-join and Q-Sum on our setup (described in Section 3.4), so that we can construct the hypergraph of these queries. We then perform min-cut partitioning over this hypergraph to get a 20-way partitioning, and then we apply LMBR on this setup. Based on placement given by LMBR, we place the data items across the 20 machines. Then we execute our test queries and carefully make sure that each query is executed on the set of machines that it spans. Query span is calculated by using set-cover algorithm on the placement suggested by LMBR. Average span over these test queries was 3, i.e., data needed for

these queries were located on an average of 3 machines using LMBR.

In Figure 3.4a, we plot the query response times of our test queries on the horizontal partitioning placement on 20 machines and we compare it with the query response times when executed on LMBR-suggested placement. We notice that query response times for complex analytical test queries TPC-H1, TPC-H2 and Q-join decrease significantly when executed on LMBR suggested placement. This is because of minimization of overheads caused by distributed analytical processing, e.g., communication overheads in processing complex joins. On the other hand, query response times for test queries TPC-H3, TPC-H4 and Q-Sum increase with colocation. This confirms our intuition that parallelism is more effective for simple queries than for complex queries.

Figure 3.4b shows that, irrespective of the type of the query, energy consumption decreases significantly with colocation of accessed data items. It shows that most reduction in energy consumption for complex analytical query is for TPC-H1 that is almost 79%, whereas for Q-Join we observe 94.24% reduction. For simple aggregate queries, we observe that there can be a tradeoff between query response time and energy consumption on colocation. For queries TPC-H3, TPC-H4 we observe that the reduction in energy consumption is 78% and 33% and for Q-Sum there is a 70% reduction. Depending upon the optimization goal such as query response time or energy

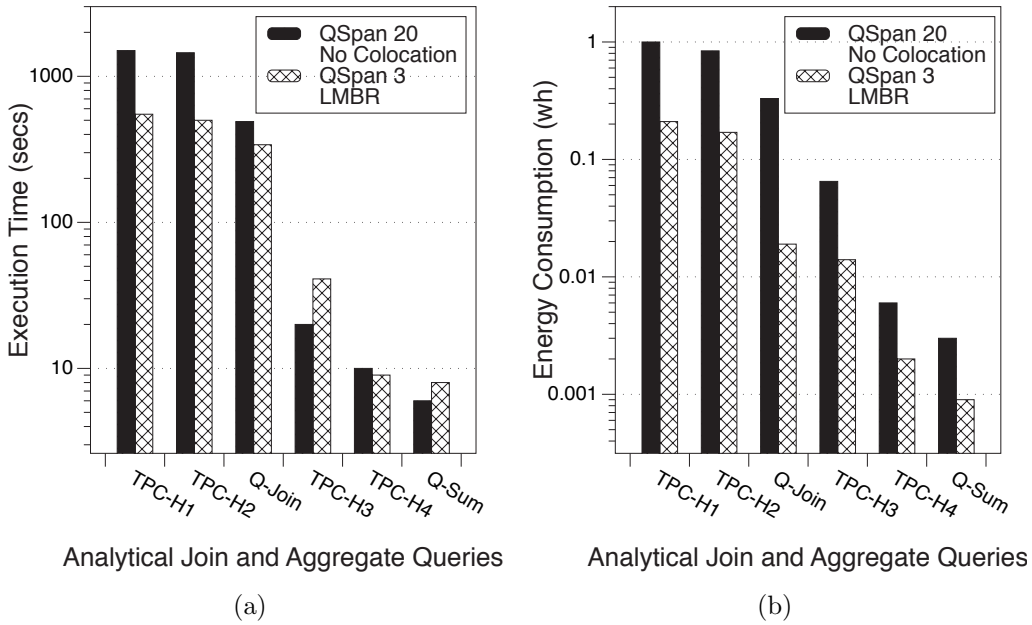
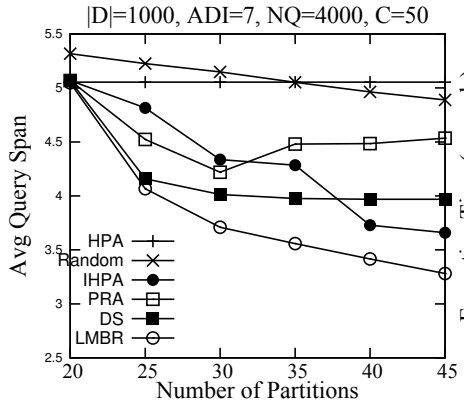


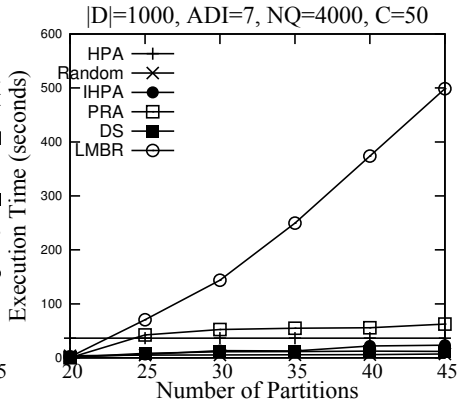
Figure 3.4: **Experiments on a TPC-H benchmark showing an effect of colocation on query response times and resource consumption (normalized). Y-axis for both plots is in log scale.**

minimization or both, one may choose to colocate the data items or not. In this work, we specifically focus at opportunities where colocation is applicable and provides us significant benefits in terms of minimization of energy consumed per query, it may also minimize query response times, for example: in case of complex analytical queries.

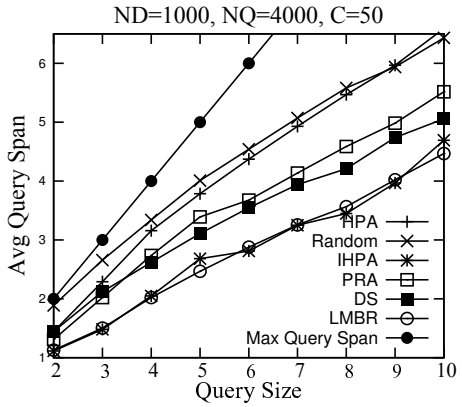
This experiment highlights the fact that, query response time may increase or decrease with colocation depending up on the nature of the query (complex analytical or simple aggregate). But in all cases, energy costs reduces with a good data colocation, for example: colocation provided by LMBR.


|D|=1000, ADI=7, NQ=4000, C=50
Avg Query Span
Number of Partitions
HPA
Random
IHPA
PRA
DS
LMBR

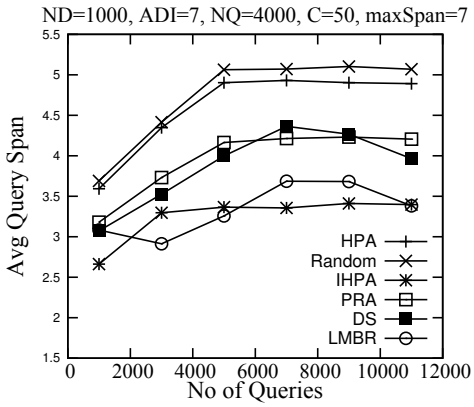
(a)


|D|=1000, ADI=7, NQ=4000, C=50
Execution Time (seconds)
Number of Partitions
HPA
Random
IHPA
PRA
DS
LMBR

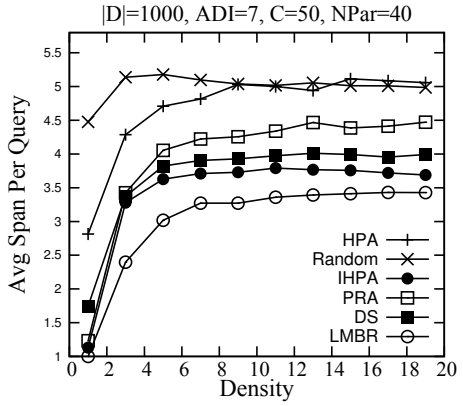
(b)


ND=1000, NQ=4000, C=50
Avg Query Span
Query Size
HPA
Random
IHPA
PRA
DS
LMBR
Max Query Span

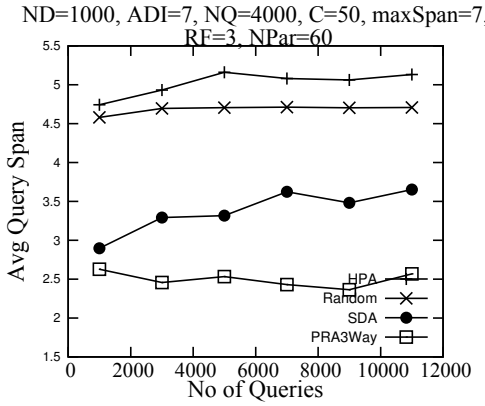
(c)


ND=1000, ADI=7, NQ=4000, C=50, maxSpan=7
Avg Query Span
No of Queries
HPA
Random
IHPA
PRA
DS
LMBR

(d)


|D|=1000, ADI=7, C=50, NPar=40
Avg Span Per Query
Density
HPA
Random
IHPA
PRA
DS
LMBR

(e)


ND=1000, ADI=7, NQ=4000, C=50, maxSpan=7,
RF=3, NPar=60
Avg Query Span
No of Queries
HPA
Random
SDA
PRA3Way

(f)

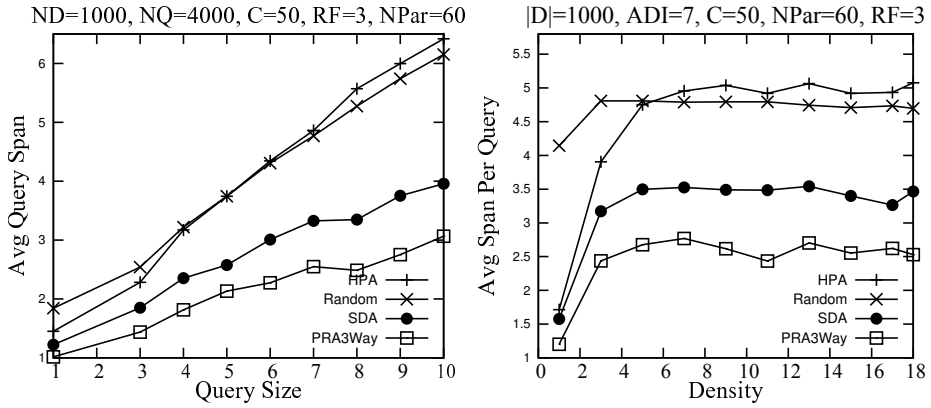


Figure 3.3: $(a)-(e)$ **Experiments on the Random dataset with homogeneous data items illustrate the benefits of intelligent data placement with replication; the LMBR algorithm produces the best data placement in almost all scenarios. Note that, for clarity, the y-axes for several of the graphs do not start at 0.** $(f)-(h)$ **3-way replication results with replication factor of each node $RF = 3$.**

3.4.2 Experiments on Random Dataset

We study the effectiveness of our data placement and replica selection algorithms in reducing the query span by varying the parameters such as number of partitions, query size, number of queries and graph density.

We begin with showing the results for the Random dataset with homogeneous data items.

Increasing Number of Partitions (ND): First, we run experiments with increasing the number of partitions. With the default parameters, a minimum of 20 partitions are needed to store the data items. We increase the number of partitions from 20 to 45, and compute the average query spans,

and average execution times, for the six algorithms over 10 runs. Figures 3.5a, and 3.5b show the results of the experiment. HPA does not do replication, and hence the corresponding plot is a straight line. The performance of the rest of the algorithms, including Random, improves as we allow for replication. Among those, LMBR performs the best, with IHPA a close second. We saw this behavior consistently across almost all of our experiments (including the other datasets). LMBR's performance does come with a significantly higher execution times as shown in Figure 3.5b. This is because LMBR tends to do a lot of small moves, whereas the other algorithms tend to have a small number of steps (e.g., DS runs the densest subgraph algorithm a fixed number of times, whereas PRA only has three phases). Since data placement is a one-time offline operation, the high execution time of LMBR may be inconsequential compared to the reduction in *query span* it guarantees.

Increasing Query Size (*ADI*): Second, we vary the number of data items per query from 2 to 10 (by setting minQuerySize = maxQuerySize), choosing the default values for the other parameters. As expected (Figure 3.5c), the average span increase rapidly as the query size increases. The relative performance of the different algorithms is largely unchanged, with LMBR and IHPA performing the best.

Increasing Number of Queries (NQ): Next, we vary the number of queries from 1,000 to 11,000, thus increasing the density of the hypergraph (Figure 3.5d). The average query span increase rapidly in the beginning and much more slowly beyond 5,000 queries. Once again the LMBR algorithm finds the best solution by a significant margin compared to the other algorithms.

Increasing Data Item Graph Density: Finally, we vary the data item graph density from 2 (very sparse) to 20 (dense). The number of partitions was set to 40. As we can see in Figure 3.5e, for low density graphs, the average span of the queries is quite low, and it increases rapidly as the density increases. Note that the average query size did not change, so the performance gap is entirely because of the structure of the query hypergraph for low density data item graphs. Further, we note that the curves flatten out as the density increases, and don't change significantly beyond 10, indicating that the query workload essentially looks random to the algorithms beyond that point.

3.4.3 Effectiveness of 3-Way Replication Algorithms

Figures 3.5f, 3.4g and 3.4h show a set of experimental results comparing the 3-way replication algorithms that we have discussed in Section 3.3.6.

Increasing Number of Queries (NQ): Increasing the number of queries, thus increasing the density of the graph, we observe that PRA-based 3-way

replication algorithm performs the best. This is in comparison with HPA (no replication), Random 3-way replication, and simple distribution algorithm (SDA). As the average number of incident hyperedges per node increases, it is more likely that SDA, which distributes the 3 copies of a node randomly to the hyperedges incident on it, will make bad decisions. Hence SDA's average span increases with the number of queries. On the other hand, PRA employs the *hitting set* technique to do a more judicious replica distribution among the incident hyperedges. Increase in the number of queries doesn't seem to affect the query span for PRA, which indicates the effectiveness of the PRA approach. Hence, PRA-based technique performs consistently better than SDA in this experiment.

Increasing Query Size (ADI): Query span for all the algorithms increases with an increase in average data items per query. PRA again performs consistently better than SDA and other algorithms.

Increasing Data Item Graph Density: PRA performs much better than Random and SDA when density of the graph is varied. Analysis is similar to what we have discussed before in Section 3.4.2.

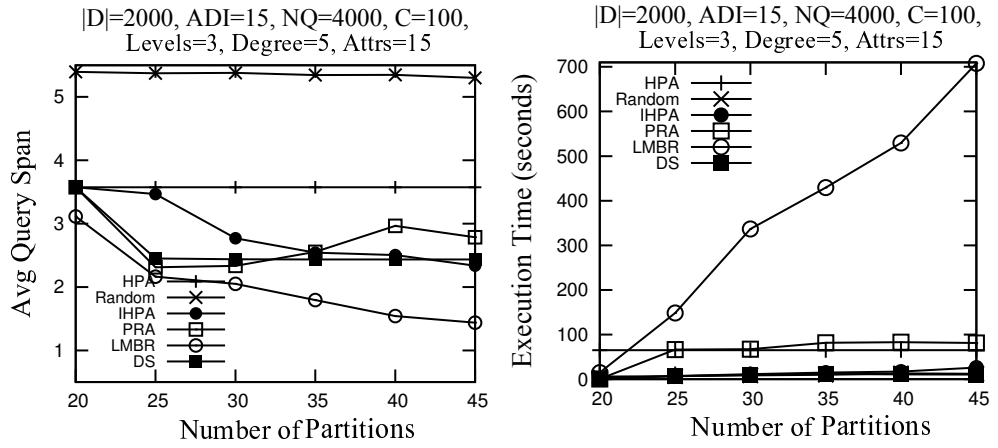
We do not compare with LMBR for this scenario due to its high running time, and because it cannot guarantee the replication constraint of 3-way replication.

3.4.4 Experiments on Snowflake Dataset

We further show our results on Snowflake dataset to corroborate our observations and claims.

Figures 3.4a and 3.4b show a set of experimental results for the Snowflake dataset. Each of the plotted numbers corresponds to an average over 10 random query workloads. The data item graph itself was generated with the following parameters: the number of levels in the graph was 3, the degree of each relation (the maximum number of tables it may join with) is set to 5, and the number of attributes per table is set to 15. The total number of data items was 2000, requiring a minimum of 20 partitions to store them. Note that we assume homogeneous data items in this case. We plot the average query spans, and the average execution times as the number of partitions increases from 20 to 45.

We also conducted a similar set of experiments with heterogeneous data item sizes, where we generated TPC-H style queries with data item sizes adhering to the TPC-H benchmark. We chose the scale factor of 25, which means the highest data item size is 28 GB and smallest data item size is 25KB. This results in a high skew among the table column sizes. Data item size is calculated as $Size(columnDatatype) * noRows$. The partition capacity was fixed at 100 GB, and we once again plot the average query spans and



(a) (b)

Figure 3.4: **Results of the Experiments on the Snowflake Dataset**

the average execution times as the number of partitions increases from 20 to 45. The results are shown in Figures 3.5a and 3.5b.

Our results here corroborate the results on the Random dataset. We once again see that LMBR performs the best, finding significantly better data layouts than the other algorithms. The performance differences are quite drastic with homogeneous data item sizes – with 45 partitions, LMBR is able to achieve an average query span of just 1.5, whereas the baseline HPA results in an average span of 3.5. However, we observe that with heterogeneous data item sizes, the advantages of using smart data placement algorithms are lower. With an extreme skew among the data item sizes, the replication and data placement choices are very limited.

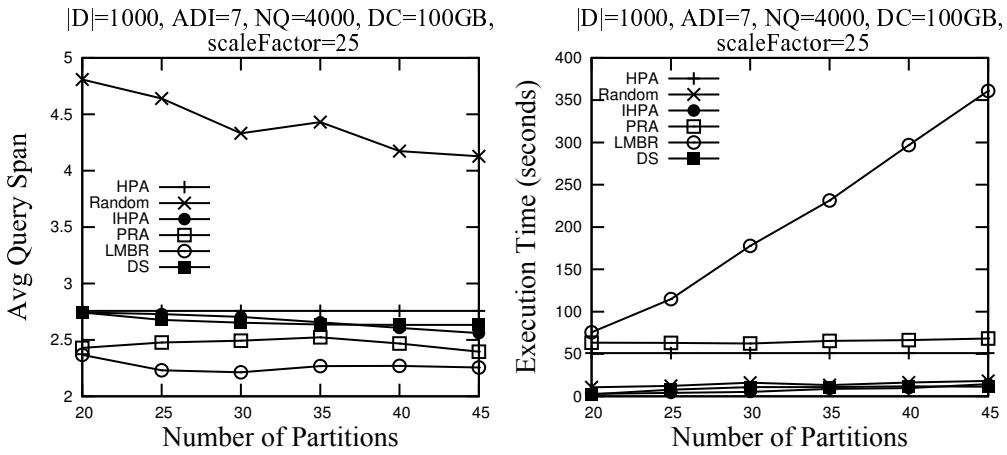


Figure 3.5: **Results of the Experiments on a TPC-H style Benchmark with unequal data item sizes. The relation sizes were calculated assuming a scale factor of 25.**

3.4.5 Experiments on ISPD98 Benchmark Dataset

Finally, Figure 3.6 shows the comparative results for first ten of hypergraphs from the ISPD98 Benchmark Suite, commonly used in the hypergraph partitioning literature. The number of hyperedges in the datasets range from 14111 to 75196 and number of nodes range from 12752 to 69429. Here we set the partition capacity so that exactly 20 partitions are sufficient to store the data items, and we plot the results with number of partitions set to 35. The hypergraphs in this dataset tend to have fairly low densities, resulting in low query spans. In fact, LMBR is able to achieve an average query span of close to the minimum possible (i.e., 1) with 35 partitions. Most of the other algorithms perform about 20 to 40% worse compared to LMBR.

3.4.6 Discussion

Our experimental evaluation corroborates our claim that intelligent data placement with replication can significantly reduce the coordination overheads in data centers. We also observe a trade-off between scalability of the data placement algorithms and the quality of data colocation. LMBR provides the best data colocation, but the performance comes with significantly higher execution times. This is because LMBR tends to do a lot of small moves, whereas the other algorithms tend to have a small number of steps (e.g., DS runs the densest subgraph algorithm a small number of times, whereas PRA only has three phases). Also with increase in number of data items, LMBR's execution time degrades sharply. On the other hand, since data placement is a one-time offline operation, the high execution time of LMBR may be inconsequential compared to the reduction in *query span* it guarantees.

The selection of data placement algorithm should primarily be based on the requirements of the application scenario at hand and the granularity of data items. On the other hand, in analytical workloads where the data items are relations or files, LMBR may be more appropriate given it usually results in best colocation.

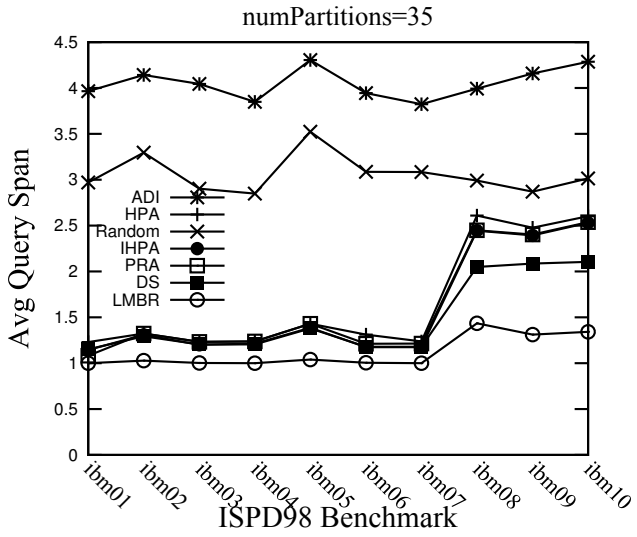


Figure 3.6: **Results of the experiments on the first 10 hypergraphs, *ibm01, . . . , ibm10*, from the ISPD98 Benchmark Dataset**

3.5 Summary of Contributions

We develop a workload-driven approach that aims to reduce the average query span in distributed data management systems by co-locating data items that are frequently accessed together by queries. We observe that, for fault tolerance, load balancing, and availability, those systems typically maintain several copies of each data item (e.g., Hadoop file system (HDFS) maintains at least 3 copies of each data item by default [89]), and we propose exploiting this inherent replication to achieve higher colocation by judicious replica creation and placement. Our approach is workload-driven in that, we propose capturing a historical query workload over a period of time, and optimizing data placement and replication for that workload. Our techniques work on an abstract representation of the query workload, and are applica-

ble to both multi-site data warehouses and general purpose data centers. we represent the query workload as a *hypergraph*, where the nodes are the data items and each query is translated into a *hyperedge* over the nodes. The data items could be database relations, parts of database relations (e.g., tuples or columns), or arbitrary files. The goal is to store each data item (node in the graph) onto a subset of machines/sites (also called *partitions*), obeying the storage capacity requirements for the partitions. Note that the partitions do not have to be machines, but could instead represent racks or even datacenters. The span of a query is defined to be the smallest number of partitions that contain all the data that the query needs. Our goal is to find a layout that minimizes the average span over all queries in the workload. Further, our algorithms can optimize for load or storage constraints, or both.

Our key contributions include formulating and analyzing this problem, drawing connections to several problems studied in the graph algorithms literature, and developing efficient algorithms for data placement. In addition, we examine the special case when each query accesses at most two data items – in this case the hypergraph is simply a graph. For this case, we are able to develop theoretical bounds for special classes of graphs that gives an understanding of the trade-off between resource consumption and storage. We have also built a trace-driven simulation framework that enables one to systematically compare different algorithms, by automatically gener-

ating varying types of query workloads and by calculating the total energy cost of a query trace. We conducted an extensive experimental evaluation using this framework, and results show that our techniques can result in high reductions in query spans and resource consumption compared to baseline or random data placement approaches.

A recent system, CoHadoop [30], also aims at co-locating related data items to improve performance of Hadoop; the algorithms that we develop here can be used to further guide the data placement decisions in their system. We can use similar techniques as discussed in this work to partition large graphs across a distributed cluster; smart replication of some of the (boundary) nodes can result in significant savings in the communication cost to answer queries (e.g., to answer subgraph pattern queries). More recently, Curino et al. [21] also proposed a workload-aware approach for database partitioning and replication to minimize the number of sites involved in distributed transactions; our algorithms can be applied to that problem as well. However, we note that replication costs become critical in that case. Our techniques are also applicable in partition farms such as MAID [20], PDC [70], or Rabbit [7], that utilize a subset of a partition array as a workhorse to store popular data so that other partitions could be turned off or sent to lower energy modes.

Minimizing average query spans through replication and data placement raises two concerns. First, does it adversely affect load balancing? Focusing

simply on minimizing query spans can lead to a load imbalance across the partitions. However, in many cases this may not be a major concern and can be solved by employing smart routing of queries to physical partitions, so we believe total resource consumption should be the key optimization goal. Most analytical workloads are typically not latency-sensitive, and we can use temporal scheduling (by postponing certain queries) to balance loads across machines. We can also easily modify our algorithms to incorporate load constraints. A second concern is the cost of replica maintenance. However, most distributed systems do replication for fault tolerance, and hence we do not add any extra overhead. Secondly, most systems focused on large-scale analytics do batch inserts, and the overall cost of inserts is relatively low.

Chapter 4

WAFEL: Data Placement Framework for Cost Effective Distributed Information Retrieval

Minimizing *query spans* has wide applicability and benefits. In this chapter, we discuss one such application, namely, distributed information retrieval where minimizing *query spans* can have a major impact. We also note that, minimizing query spans through data colocation can often lead to increased load imbalance. To mitigate that problem, we propose several techniques to effectively trade search cost with load imbalance.

4.1 Introduction

Information retrieval (IR) has never been as important and essential as it is in this age of Big Data. We have witnessed IR applications being brought out of the confines of libraries to our personal computers, laptops, tablets and mobile phones for everyday usage. The volume of search requests that commercial search engines service daily is an attestation of our increasing search requirements. For the largest search engine by volume, the figures are in the order of few billion queries per day. Figure 4.1 shows the architecture of distributed information retrieval system where users interact with the IR system through a search engine interface. User query is sent to query federator which then dispatches this query to distributed repositories of search indexes. Ranked document lists from each repository are sent back to query federator and final results are then sent back to the user.

This increased reliance on and need for search has been driven by many factors, one of which is the growing availability of large, information-rich, and search-friendly collections. The Web is its most prominent example but it is not the only one. More and more organizations and businesses are digitizing all types of internal data in order to make it searchable. The information that can be mined from these large collections is often invaluable to the organizations. This deluge of data and increasing importance and need

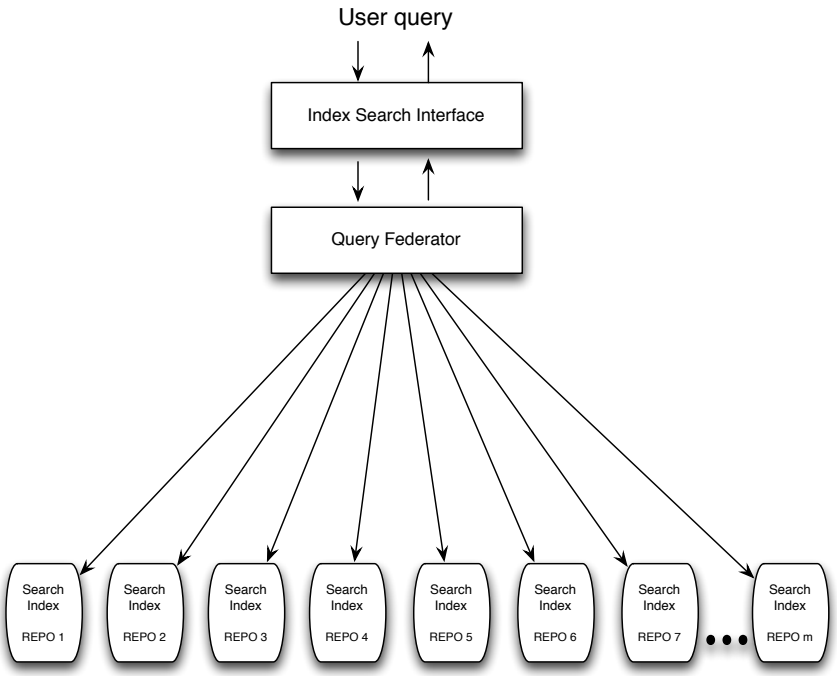


Figure 4.1: **Distributed Information Retrieval**

to search demands scalable search mechanisms to effectively and efficiently search, analyze and gain insights from the data. Typically, partitioning of web documents onto a large number of machines is done to scale up information retrieval.

Simplest way of partitioning the web documents is through random partitioning where documents are randomly hashed to partitions. In this, search engine indexes for large document collections are distributed across multiple physical partitions and searched in parallel to provide rapid interactive search. This means that each query has to hit every partition to retrieve all the relevant documents. Typically this leads to *distributed exhaustive search*

where all partitions are searched for each query, which means that queries will have large *query spans*. This can be problematic for both large and small organizations in terms of overall amount of resources spent to cater to search queries. Today, most of the large search giants are aware of seriousness of resource wastage which has direct influence on overall energy consumption, and are making every effort to improve per query energy efficiency. Google already is measuring per search-query cost [38] and is being very careful about it, as several billion search queries hit Google in a very short time frame. Even for small organizations with modest computing resources the high query processing cost of this exhaustive search setup can be a deterrent to working with large collections. On the brighter side, this approach also naturally leads to low load imbalance.

So to deal with high processing cost of *distributed exhaustive search*, a while back, researchers had the idea — why not cluster documents by *topic*, and then only search top p% of clusters. This approach is termed *distributive selective search* [48]. The previous implementation of *selective search* was to assign clusters to physical partitions randomly, which leads to high *query span*. This approach reduces the search cost to certain extent, but is not optimal. Let us consider an example where we have 1 billion web documents spread across 100 physical partitions, then in *distributed exhaustive search* setting, search query is shipped to all the physical partitions. Now lets say, we

cluster these million documents to 10000 document clusters and spread these clusters randomly or in a round robin fashion on to the physical partitions. If we consider that each query accesses top 50 clusters, then in most cases query span will be 50. But if we analyze the workload and carefully place the clusters on to physical partitions, then query spans can be much less than 50, reducing the search cost significantly.

The topic-based partitioning of the collection is central to the success of *distributed selective search* approach. However, this topically skewed arrangement of documents can make the task of balancing the computing load on the cluster difficult. This might be especially challenging in search environments where the query stream exhibits high fluctuations in the popularity of topics, as is often observed in Web search. In such scenarios, the resources assigned to a few of the currently popular document clusters could be over-utilized while the other document clusters' resources may idle. We solve this problem by considering query frequency (as a proxy for topic popularity) into account in our solution model so that document clusters are assigned to resources in a load-balanced way.

In this work, our goal is to reduce the per query search cost by trying to reduce the number of physical partitions involved in the execution of a search query (i.e., *query span*). Minimizing query span can have multiple advantages in distributed IR setting, firstly, because this approach allows

search queries to be shipped to only a few physical partitions overall search cost reduces drastically. Secondly, from Google's experience [22] we know that as search query touches large number of physical partitions for relevant document retrieval, service latency increases drastically because of variability in execution times. From this observation, we can naturally deduce that minimizing search query span can improve search service latencies. Thirdly, we note that total amount of resources consumed typically increases with increased query span as demonstrated in Chapter 1. Lastly, minimizing the total amount of resources consumed directly reduces the total energy consumption of the task.

While minimizing overall search cost, we also need to guarantee load balancing. However, these two optimization goals have inherently different dynamics – search cost minimization is tied to minimizing *query span* which requires clustering of the data, such that search query accesses minimum number of physical partitions, whereas load balancing is usually tied with de-clustering of the data. Hence, we decouple the overall problem into two phases: (1) we cluster the web documents into document clusters and then by analyzing workload history we assign these document clusters to partitions (data partitioning), such that overall *search cost* can be minimized, and (2) then make the load-aware replication and routing decisions to minimize *load imbalance*. Data partitioning to minimize average query spans is

a computationally expensive process, and we expect that the partitioning, once performed, will be useful for a long period of time with incremental partitioning to handle minor changes in the workload. So, in this work, in order to handle variations in the workload we also provide techniques to incrementally repartition the document clusters by moving only subsets of document clusters across the partitions to minimize the load imbalance without compromising on search cost.

4.2 Overview

We begin with providing preliminaries, our experimental setup and a high-level overview of WAFEL's architecture.

4.2.1 Definitions and Notations

Definitions of hypergraph, N-way partitioning, query span and cut are same as described in Chapter 3. Below are the definitions and notations that are useful in the current context.

Partition degree d_{P_i} of P_i is equal to the sum of the weights of the hyperedges that contain at least one vertex in P_i and one vertex in $P - P_i$. can be denoted as:

$$d_{P_i} = \sum_{e \in E_i | qs_e > 1} w(e)$$

where E_i is the set of hyperedges incident on partition P_i. Then set of

partition degrees can be denoted by $d_P = \{d_{P_1}, d_{P_2}, \ldots, d_{P_N}\}$.

Load (L_p): Partition load can be defined as the number of queries incident on a partition in a given time window.

Load Imbalance (LI): Load imbalance is defined as ratio of maximum load and minimum load that is denoted by $LI = \frac{max(L_p)}{min(L_p)}$.

Load Constraint: *Load constraint* is defined as ratio of total number of queries to the maximum number of queries allowed to serve a partition.

4.2.2 WAFEL Architecture

In this work, we develop a system WAFEL, that facilitates the data partitioning, repartitioning, replication and routing for cost effective distributed information retrieval. We begin with providing a high-level overview of WAFEL's architecture. The key components of WAFEL are shown in Figure 5.1, and can be functionally divided into four groups: clustering of web documents and workload modeling, data partitioning and replication, search cost and load-aware routing and incremental repartitioner. We briefly discuss the key functionality of the different components next.

Clustering Component: This module takes the web documents as an input and using appropriate settings runs the topical clustering algorithm such as k-means clustering over these documents to come up with document clusters. This module makes use of Apache Mahout software to cluster the

documents. These document clusters are then passed on to the workload modeling component. Additional information about clusters is also passed onto workload analyzer and statistical module, so that given the workload history, the statistical module can analyze the query access patterns to these clusters.

Workload Analyzer: As information retrieval engine serves queries, this module records the statistics regarding the relevant documents accessed by each query. It also records per document access frequencies. Once this module receives document clusters from clustering component, based on query-documents access statistics it re-calculates the query-clusters access statistics with per cluster access frequencies.

Workload Modeler: This module primarily interacts with workload analyzer and statistical module, and clustering component to get information about document clusters and query-clusters access patterns. After collecting required information, this module generates a model that represents the workload. Once this workload model is generated it is passed onto data partitioner.

Data Partitioner: Primary objective of the data partitioner is to partition the given clusters such that search cost is minimized. Once data partitioner module receives the workload model from workload modeler module, it par-

titions these clusters and places them onto multiple partitions.

Replicator: Once data is partitioned on to multiple partitions, replicator module gets the cluster access frequency information from statistical module and performs K-way load-aware replication. This data partitioning and replication information is then collectively passed onto the routing module.

Search Cost and Load-aware Router: This module smartly routes the queries to the partitions such that it effectively minimizes search cost and load imbalance whenever possible. The router determines the document clusters accessed by the search query, their replicas, and their location information using the mappings provided by the data partitioner. The router then uses a load-aware set-cover based algorithm to compute the minimum number of partitions that the search query needs to be executed on (i.e., query span), to access all the required clusters and replicas.

Search Cost and Load-aware Repartitioner: The workload monitoring and statistical module monitors the workload changes and maintains statistics on the workload access patterns. It provides this input to the incremental repartitioning module which identifies when the current partitioning is sub-optimal and triggers data migration to deal with workload changes. The data migration is done in incremental steps through the data placement module during periods of low activity.

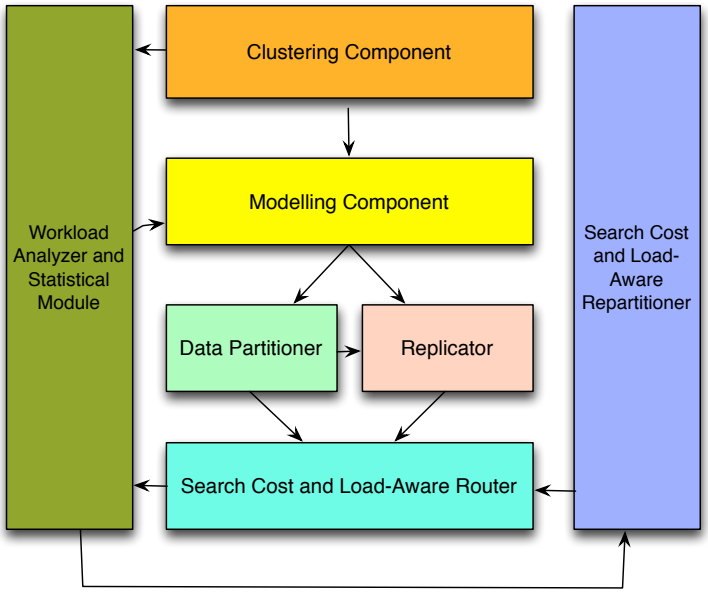


Figure 4.2: **WAFEL Architecture**

4.3 System Design

In this section, we first discuss our document cluster assignment strategy. We then present our proposed techniques for minimizing search cost through scalable workload-aware partitioning. We then discuss various techniques for achieving load-imbalance minimization where we discuss our load-aware replication and routing techniques. Finally, we discuss incremental repartitioning to cater to workload variations.

4.3.1 Minimizing Search Cost

In this work, we minimize per query cost by minimizing number of physical partitions accessed by the query.

Minimizing query span: One of the important objectives of this work is to assign search engine indexes document clusters to physical partitions so that per query cost is minimized. We model our query workload as hypergraph where vertices represent document clusters and hyperedges represent search queries accessing the documents in these document clusters. N-way min-cut balanced partitioning over this hypergraph gives us the partitions that minimizes the number of hyperedges that span the multiple partitions.

4.3.1.1 Workload-aware Data Partitioning

Figure 4.3 shows the set of documents $\mathcal{DC}$ that are clustered into M document clusters by a standard clustering algorithm such as K-means or any topical clustering technique. By analyzing query logs, query workload containing query frequencies and query-document cluster access statistics is retrieved. Given a set of search engine document clusters $\mathcal{D}$ and a set of partitions, our goal is to decide which document clusters to replicate and how to place them on the partitions to minimize the average span of an expected query workload. For simplicity, we assume that we are given a total of N identical partitions each with capacity C units, and further that the data items are all homogenous in terms of size (we will relax this assumption later). Clearly, the number of document clusters must be smaller than $N \times C$ (so that each data item can be placed on at least one partition). Further,

let N_e denote the minimum number of partitions needed to place the index document clusters (i.e., $N_e = \lceil |\mathcal{D}|/C \rceil$).

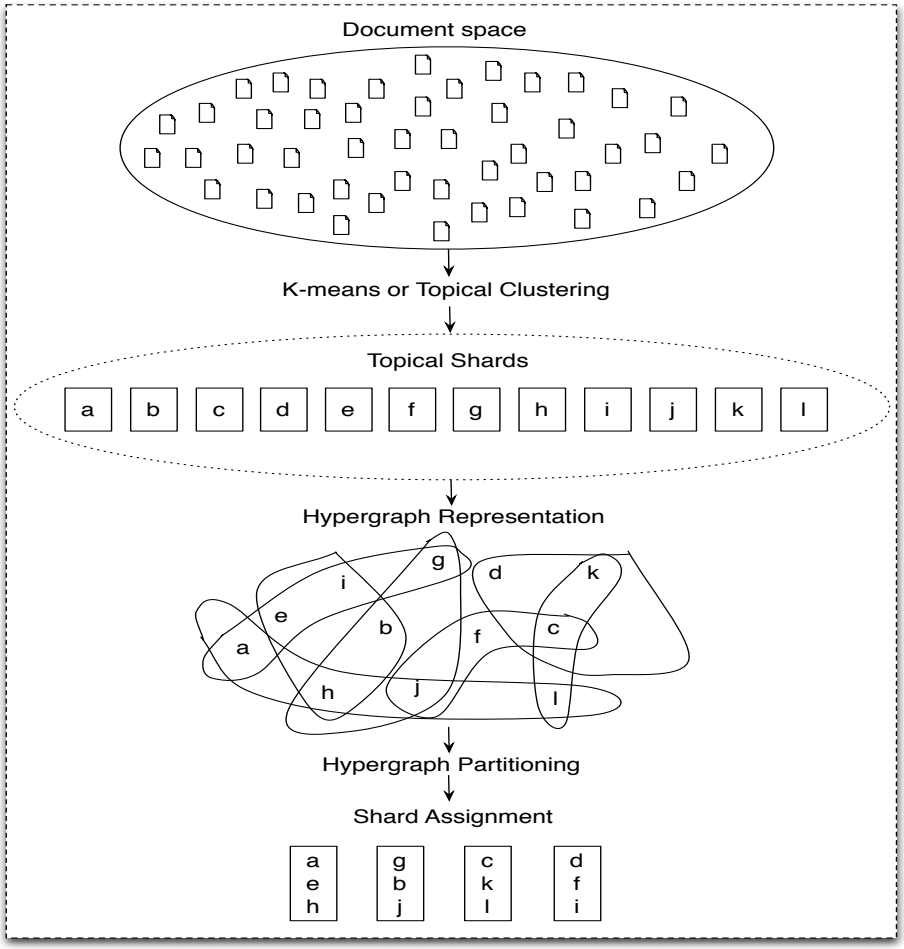


Figure 4.3: **Proposed document cluster assignment framework idea**

Creating a Hypergraph Once we get the topical clusters by applying K-means clustering on web documents, we select top $x\%$ of clusters according to the relevance per search query and regard these clusters as the nodes V and queries themselves become hyperedges E. By default, in this work, we

assume top 10% of clusters according to the relevance per search query.

4.3.1.2 Novel Scalable Hypergraph Partitioning

State-of-the-art hypergraph partitioning approaches do not scale to large hypergraphs with millions of hyperedges with at least tens of thousands of nodes that we deal with in this work. In this work, we provide an effective and generic heuristic to scale current hypergraph partitioners to partition massive hypergraph.

Parallel Partitioning: To solve this problem, let us consider a hypergraph partitioning algorithm (HPA) that partitions a given hypergraph $\mathcal{H} = (V, E)$ in to K balanced partitions to give mapping of nodes to partitions $VP = \{V \rightarrow P\}$. Now given a massive hypergraph $\mathcal{H}^m = (V^m, E^m)$ that contains millions of hyperedges, our goal is to partition $\mathcal{H}^m$ into N partitions. But because of massive scale of the hypergraph in terms of number of hyperedges, current hypergraph partitioning packages fail to partition this hypergraph which we have verified through our experiments on different hypergraph partitioning packages like hMetis, and PaToH. In order to scale partitioning of extremely dense hypergraphs the key idea behind our approach is to split the hyperedges into M disjoint pieces: $\mathcal{H}^m = \{\mathcal{H}^m_1, \mathcal{H}^m_2, \ldots, \mathcal{H}^m_M\}$. We then run HPA on each smaller hypergraph, $\mathcal{H}^m_i$ independently, to get a set of mappings, VP_i, between nodes and partitions. To determine the node to

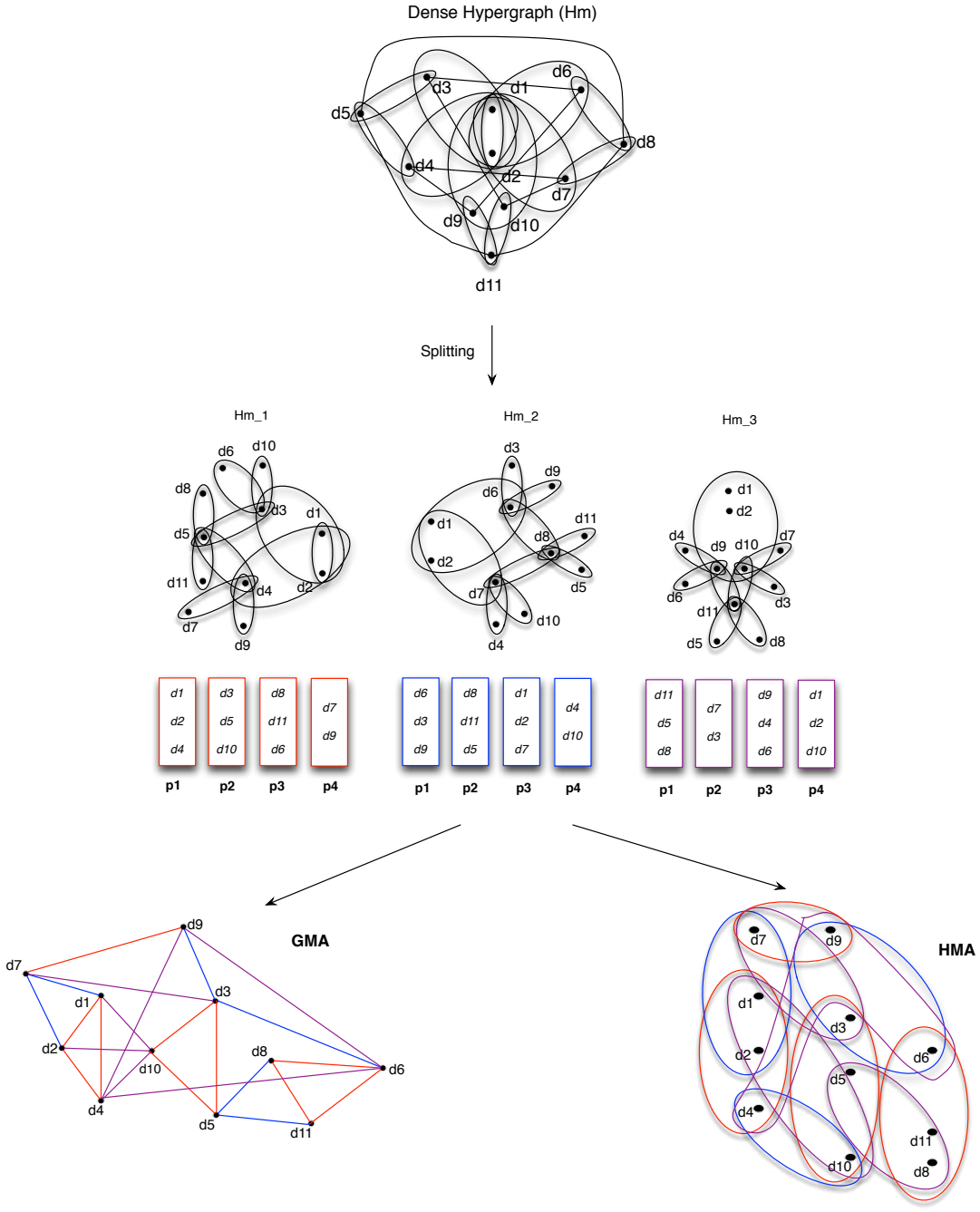


Figure 4.4: **Example figure showing modeling of hypergraph-base membership approach (HMA) and graph-based membership approach (GMA)**

partition mapping we come up with several approaches:

- **Maximum Membership Approach (MMA):** First approach is *Maximum Membership Approach* (MMA). For a node n_j, given the M mappings, we find the partition that it is mapped to most often, i.e., $P_{n_j} = mode\{VP_1, .., VP_i\}$. Key idea here is to choose the partition P_{n_j} which is mapped to this node maximum number of times. MMA decides the membership of a node in a partition with certain membership confidence. If nt_j is the maximum number of times a node n_j is mapped to a partition, then membership confidence of a node n_j is defined as $MC_j = \frac{nt_j}{M} \times 100$. Average membership confidence over all nodes $MC = \Sigma_{n_j=1}^{N} MC_j/N$.

  Constraints: In order for this heuristic to give maximum benefits, following are the constraints that have to be followed: 1) Each hypergraph piece $\mathcal{H}_i^m$, should contain all the data nodes. (For extremely dense graphs, this is mostly true). 2) M should always be greater than N, $(M > N)$, bigger the M (satisfying the constraint 1.) better the partitioning quality. Key intuition behind requiring that $M > N$ is that, it helps to ensure that, for every node n_j, there is a single partition that dominates among the mappings of n_j.

  Figure 4.5 shows that as we increase the number of hypergraph splits

M, after a certain threshold, the average query span starts to increase. This happens because as hypergraph granularity increases, density of each hypergraph split decreases; because of this, when HPA is applied, partition membership confidences for the nodes are low. For very high M, MMA starts to assign nodes to partition randomly because of very low node to partition membership confidence.

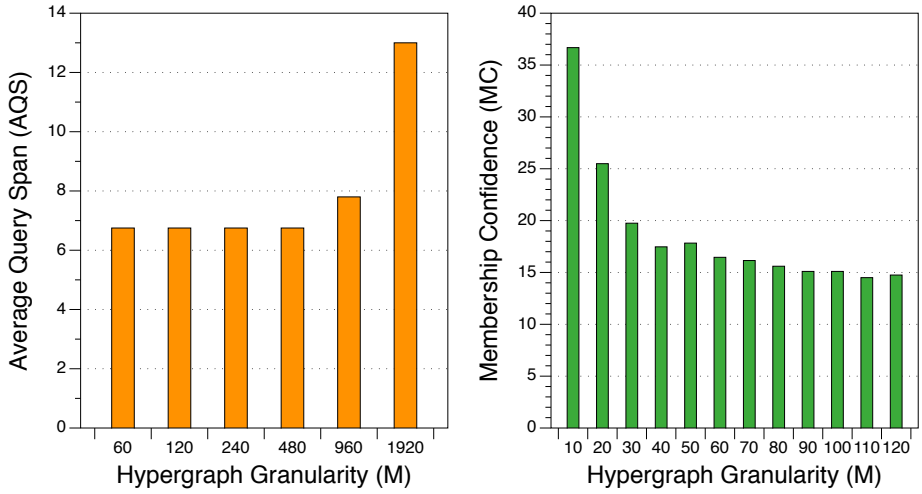


Figure 4.5: **a) Effect of hypergraph granularity** (M) **on average query span, b) Effect of** M **on membership confidence.**

Problem: This approach of splitting the large hypergraph and executing HPA on smaller hypergraphs parallely and heuristically determining the final mapping between node to partitions only works if HPA assigns partition ids deterministically to the partitions for every individual run. Figure 4.6 shows a simple example of this problem. In this example figure, a hypergraph with 5 hyperedges and 5 nodes are shown. Lets

consider executing a variant of MMA on this example, where we apply HPA on this hypergraph in 3 different runs. In this example, for each run, HPA assigns partition id to partitions randomly and places data items into partitions according to min-cut partitioning. In this hypergraph one can notice that nodes d_1 and d_2 have strong co-occurrence, so we expect them to be assigned to the same partition. But MMA approach assigns d_1 and d_2 to different partitions because they appear in three different partitions in three different runs and there is no clear majority of node to partition mapping. In short, when HPA assigns partition ids randomly to partitions, then MMA fails to make meaningful assignments of node to partitions.

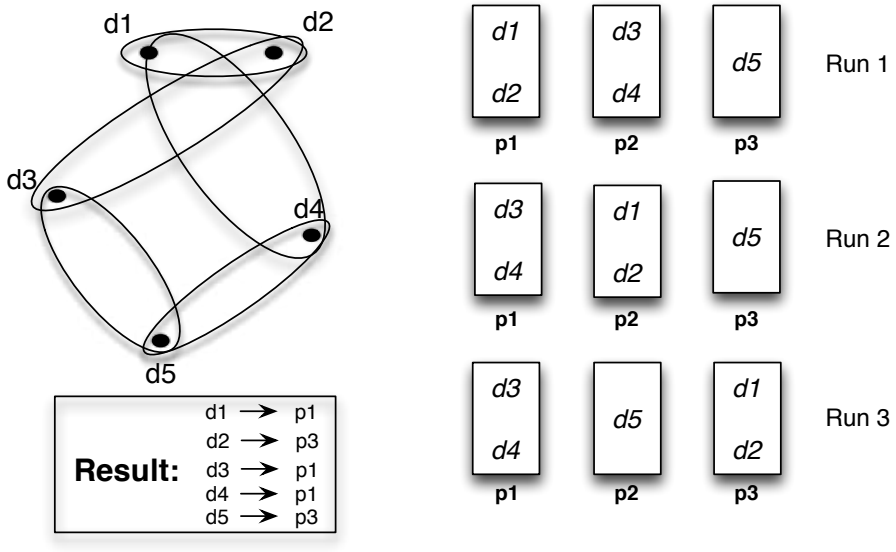


Figure 4.6: **Example figure showing the problem with Maximum Membership Approach to scale hypergraph partitioning.**

- **Hypergraph-based Membership Approach (HMA):** To solve the problem of random partition id assignment, our second approach is to develop a membership technique that is not dependent on partition id assignment performed by HPA. Main idea is to analyze each nodes to partitions mapping VP_i, and build a hyperedge e_k on the set of nodes that lie in same partition P_k such that $\cup_k e_k = E'_i$. By combining all the hyperedges for every VP_i we build an hypergraph $\mathcal{H}'_i = (V, E')$, where $\cup_i E'_i = E'$. We have $|E'| = N \times M$. Once we have $\mathcal{H}'$, we perform N-way min-cut partition over this hypergraph to get the final node to partition assignments. Advantage of this approach is that the node assignments are not partition id dependent. On the other hand, main disadvantage of this approach is that, in most cases we note $|E'| << V$ which means that the hypergraph $\mathcal{H}'$ is very sparse and that may adversely affect the partitioning quality.

- **Graph-based Membership Approach (GMA):** Another approach is to preserve the collocation of the nodes in same partitions by creating the clique between the nodes within a partition. So by looking at all the nodes to partition mappings, we can create a graph representation of this as shown in Figure 4.4 (left bottom). So when we partition this graph representation then the nodes that are densely connected are

placed onto same partition. This approach also solves the sparsity issue in HMA, where the hypergraph $\mathcal{H}'$ in HMA can be converted to a graph $\mathcal{G}$ where each hyperedge in $\mathcal{H}'$ is converted to a clique connecting every node with every other node within a hyperedge. Finally, N-way min-cut partitioning is applied on $\mathcal{G}$ to partition the graph into N pieces. This approach inherits the advantage of HMA where node to partition assignments are not partition id dependent and moreover this approach is free from sparsity issue and can give very good partitioning quality.

4.3.2 Minimizing Load Imbalance

Load imbalance is a serious issue to consider in our problem, because it can cause a drop in overall throughput as few of the physical partitions will be overloaded by query accesses whereas some of them will be grossly underutilized. When a load on a particular partition L_p is significantly higher than load on other partitions then there is said to be load imbalance. We need to balance number of query accesses across the physical partitions. In our hypergraph partitioning model, where partitions represents physical partitions and hyperedges represent the queries, we would want to balance the number of hyperedges incident on partitions.

In hypergraph theory, number of hyperedges that are cut by a particular partition is termed *partition degree*. Since, maximum partition degree is the

lower bound on the amount of routing resources that are required, being able to find partitions that both minimize the number of interconnects (queries spanning multiple partitions) which is achieved by minimizing the cut and also evenly distribute these interconnects across the physical partitions to eliminate high density interconnect regions (which is achieved by minimizing the maximum partition degree) can significantly reduce the peak demand of routing resources and thus, help in reducing the peak congestion. Partition degree and load are formally defined in Section 4.2.1.

4.3.2.1 Replication

Workload-aware partitioning of data document clusters essentially helps to minimize query cost through data co-location. In addition, in this work, to achieve our second objective which is minimization of load-imbalance we perform r-way replication of data items. Replication also helps in improving data availability and fault tolerance. In this section, we will describe our load-aware r-way replication algorithm. Let PL be the mapping between partition and its load and CL be the mapping between cluster and its access frequency. PL and CL are created by analyzing the workload history. Let C be the set of clusters and RF be the replication factor that is equal to r. Also, let NR_c be the current number of replicas made of cluster c. Key idea is to replicate the clusters by making copy of cluster from a maximum load

partition to minimum load partition and repeat this till total load imbalance is minimized, i.e., LI is approximately equal to 1. At every iteration, we identify two partitions, denoted P_{max} and P_{min}, that respectively have the maximum and the minimum load. For each partition we sort the clusters in it in ascending order according to their access frequencies. At every step we pick the cluster with highest access frequency from P_{max} and make a copy of it in P_{min}. As we make copies of clusters, we adjust the load values of P_{max} and P_{min} and also adjust the access frequency of copied cluster c with $\frac{CL_c}{NR_c}$. As we make copies of clusters and adjust the partition loads, the partitions with maximum and minimum load tend to change, so at every iteration we pick updated P_{max} and P_{min}. Algorithm terminates when ratio between maximum load and minimum load is close to 1.

4.3.2.2 Routing

Routing is an important part of the WAFEL, where for each query a decision is made to route the query to appropriate partitions. In this work, we come up with three types of routing.

- **Set-cover based Routing (S-Routing):** When a query arrives at WAFEL, this routing algorithm essentially selects the minimum set of partitions that contains all the data needed to satisfy the given query. Given the data and its replicas placed on multiple partitions, selecting the mini-

Algorithm 6 Load-aware r-Way Replication Algorithm

Require: $PL, CL, C = V, RF = r, NR_C, N$

```
while (loadImbalance(PL) ≉ 1) do
    PL = sortByValue(PL);
    P = keys(PL);
    P_min = P[0];
    P_max = P[N − 1];
    CL = sortByValue(CL);
    for (c ∈ CL) do
        if (NR_c < RF and c ∈ C_{P_max} and c ∉ C_{P_min} and CL_c < (PL_{P_max} − PL_{P_min})) then
            NR_c+ = 1;
            load = CL_c / NR_c;
            C_{P_min} ∪ = c;
            PL_{P_min} + = load;
            PL_{P_max} − = CL_c;
            PL_{P_max} + = load;
            break;
return C_P;
```

mum of partitions to satisfy a query can be mapped to set cover problem which is a NP-Hard problem. Hence, we implement greedy set cover algorithm where at each step partition that covers maximum number of uncovered document cluster is selected till all the data items needed by the query is covered. Then given query is routed to these minimum number of partitions selected by the greedy set cover algorithm.

- **Round Robin Set-cover based Routing (S-Routing-RR):** Although S-Routing results in smaller query spans, main problem with S-Routing is that, if there are 3 copies of each partition content, then for each query type only one out of three copies is accessed which results in high

load imbalance. To solve this problem, set cover algorithm selects the replicas in a round robin fashion which gives the exactly same query spans as S-Routing but with relatively lower load imbalance.

- **Simple Load-aware Routing (LA-Routing):** In this routing scheme, given a set of nodes or document document clusters V_e accessed by the query e and their replicas R_e; a set of partitions $\{P^e_{RV} \mid V_e \cup R_e \subseteq P^e_{RV}, P^e_{RV} \subseteq \mathcal{P}\}$, a query e is routed to random set of partitions $P_e \subset P^e_{RV}$. A partition is not considered if its current load exceeds the max load.

- **Load-aware Set-cover based Routing (LAS-Routing):** Let $L^{max}_{P_i}$ be the max load constraint on each partition P_i. Also, let L_{P_i} be the current load on partition P_i where $L_{P_i} \in L_{\mathcal{P}}$. The minimum set-cover problem to minimize the query span can be defined as follows: given a query e, a universe $\mathcal{U}_e$ is the set of elements required to satisfy query e, that is $\mathcal{U}_e = \{V_e | V_e \subseteq V\}$; Let V'_e be the set of elements so far covered by the set cover algorithm, that is $\mathcal{S}_e = \{V'_e \mid V'_e \subseteq V_e\}$; determine the minimum number of partitions $S \subseteq P^e_{RV}$ that cover the universe $\mathcal{U}_e$. In each iteration, the algorithm determines the partition P_i whose current load $L_{P_i} \leq L^{max}_{P_i}$ and which covers the maximum uncovered elements UC_e in the universe $\mathcal{U}_e$ given by $max(\{\mathcal{U}_e \backslash S_e\} \cap P_i)$. $\{\mathcal{U}_e \backslash S_e\}$ denotes

the operation wherein the counts of the elements in the universe $\mathcal{U}_{e'}$ are decremented by the count of the corresponding elements in S_e. As we include partition P_i into the set-cover, the partition load L_{P_i} is updated by one. If current partition load exceeds the max load, that is $L_{P_i} > L_{P_i}^{max}$ then partition will be evicted and will not be considered for further routing of queries, that is $\mathcal{P} = \mathcal{P} - P_i$. The set-cover S is updated with the partition P_i, i.e., $S = S \cup P_i$, $S_e = S_e + P_i$ which increases the count of common elements in the set-cover map by one. The uncovered elements are updated by $UC_{e'} = \mathcal{U}_e \backslash S_e$ which reduces the counts of common elements in $\mathcal{U}_e$ by the counts of the corresponding elements in S_e. The algorithm terminates when the counts of all elements in $UC_e = 0$ and outputs S. The algorithm for computing the set-cover is shown in Algorithm 7.

Algorithm 7 Load-aware Set-cover Algorithm

Require: $\mathcal{H}, e \in E, P_i \in P_{RV}, \mathcal{U}_e = \{V_e | V_e \subseteq V\}, \mathcal{S}_e = \{V'_e \mid V'_e \subseteq V_e\}, L_{\mathcal{P}}, L_{P_i}^{max}$

1: **while** $UC_e \neq 0$ **do**
2: $p_{index} = \text{argmax}_i(\{\mathcal{U}_e \backslash S_e\} \cap P_i)$
3: $S \cup = P_{p_{index}}$
4: $S_e + = P_{p_{index}}$
5: $UC_e = \mathcal{U}_e \backslash S_e$
6: $L_{P_{p_{index}}} + = 1$
7: **if** $L_{P_{p_{index}}} > L_{P_{p_{index}}}^{max}$ **then**
8: $\mathcal{P} = \mathcal{P} - P_{p_{index}}$
9: **return** S

4.3.3 Cost and Load-Aware Incremental Repartitioning

Although it is easier to guarantee certain amount of load balancing while minimizing hyperedge cuts during initial partitioning stage through hMetis, but it is certainly a non-trivial problem to deal with workload changes. Changes in workload might result in higher average query spans and also may adversely affect load balancing. A naive way to deal with workload changes is to remodel the updated query workload as hypergraph and then perform complete repartitioning using hMetis. Complete repartitioning is extremely inefficient, and may not be practical in large distributed settings. Instead, we will focus on in-place repartitioning similar to the approach adopted by Sword system [71]. In Sword, focus is mainly on minimizing affected cuts due to workload changes and they do not consider load-balancing issues at all in their approach. But in this work, we perform in-place swapping of appropriate nodes to minimize both hyperedges cut and load imbalance. Intuitively, minimizing cuts should minimize maximum partition degree too, but this has been disproved by Karypis et al., in their paper [76]. So we combine these two objectives and model this problem as a *bi-objective repartitioning problem*. These two objectives are defined as: ($Obj1$) is:

$$Obj1 = min(cut) = min(\sum_{e \in E | qs_e > 1} w(e))$$

$$Obj2 = min(Load) = min(max(d_P))$$

In general, N-way balanced hypergraph partitioning is a NP-Hard problem. Adding multiple objectives makes it an even harder problem. For the same reason, repartitioning problem is also NP-Hard. Therefore, in this section, we will discuss several repartitioning heuristic algorithms that will aim to minimize both hyperedge cuts and load.

Let $\mathcal{P}$ be initial partitioning obtained after running multi-level k-way partitioning algorithm (MHPA) on initial workload hypergraph $\mathcal{H}(V, E)$. $\mathcal{H}'(V, E')$ be the changed workload with additional hyperedges such that $E \subseteq E'$. Let d'_{P_i} be of new partition degree of P_i such that: $d'_{P_i} = \sum_{e \in E'_i | qs_e > 1} w(e)$ where E'_i is the set of hyperedges in the changed workload incident on partition P_i. Then set of updated partition degrees can be denoted by $d'_P = \{d'_{P_1}, d'_{P_2}, \ldots, d'_{P_N}\}$. Let $P'_{max} = P_{argmax(d'_{P_1}, d'_{P_2}, \ldots, d'_{P_N})}$ be the partition with current maximum load, $E'_{max} \subset E'$ be the new set of hyperedges incident on P'_{max}. $\mathcal{P}'_{sub} \subset \mathcal{P}$ denotes the subset of partitions that are incident by set of hyperedges E'_{max}.

More formally, we are given a set $\{C \mid C \subseteq \mathcal{P}'_{sub}\}$ of movable nodes and a set $\mathcal{P}'_{sub}$ of partitions. Each node $n \in C$ has a size, denoted by $size(c)$, and each partition $p \in \mathcal{P}'_{sub}$ has a capacity, denoted by $cap(p)$. Moreover, for each pair $(c, p) \in C \times \mathcal{P}'_{sub}$ we know the costs $d_{cut}((c, p))$ and $d_{load}((c, p))$ of moving

node c to region p. Cost $d_{cut}((c,p))$ represents the total hyperedge cut and $d_{load}((c,p))$ is the total load after moving node c to partitions p. The task is to find a re-mapping $g : C \rightarrow \mathcal{P}'_{sub}$ such that $\sum_{c \in C: g(c)=r} size(c) \leq cap(p)$ for all $p \in \mathcal{P}'_{sub}$, minimizing $\sum_{c \in C} d_{cut}((c, g(c)))$ and $\sum_{c \in C} d_{load}((c, g(c)))$. Unfortunately, even for single cost function, to decide if this problem has any feasible solution is NP-complete even if $|\mathcal{P}'_{sub}| = 2$.

Key idea in our approach is to incrementally move K nodes across the partitions in order to disperse the load from maximum load partition P'_{max} to other partitions in $\{\mathcal{P} - P'_{max}\}$. Figure 4.7 shows the working example of Algorithm 8. In this Figure, (i) initial partitioning through MHPA gives fairly load-balanced partitions (ii) Over the time, change in workload can result in load-imbalance, which triggers Algorithm 8 that in the first iteration selects partition with maximum load P'_{max} and corresponding connected partitions in $\{\mathcal{P} - P'_{max}\}$ (shaded in grey). (ii), (iii) and (iv) Load is dispersed from P'_{max} to across partitions in partitions in $\{\mathcal{P} - P'_{max}\}$. In the second iteration, new P'_{max} is selected and load is dispersed, this is continued till load can no more be dispersed. In our approach we handle two important cases, first is the case of unbalanced partitioning where some of the partitions have more data than other, for an instance lets say number the of nodes in P'_{max} is much greater than the number of nodes in $P' \in \{\mathcal{P} - P'_{max}\}$, that is $|V'_{P'_{max}}| >> |V'_{P'}|$. Second is the case of balanced partitioning when the number of nodes in

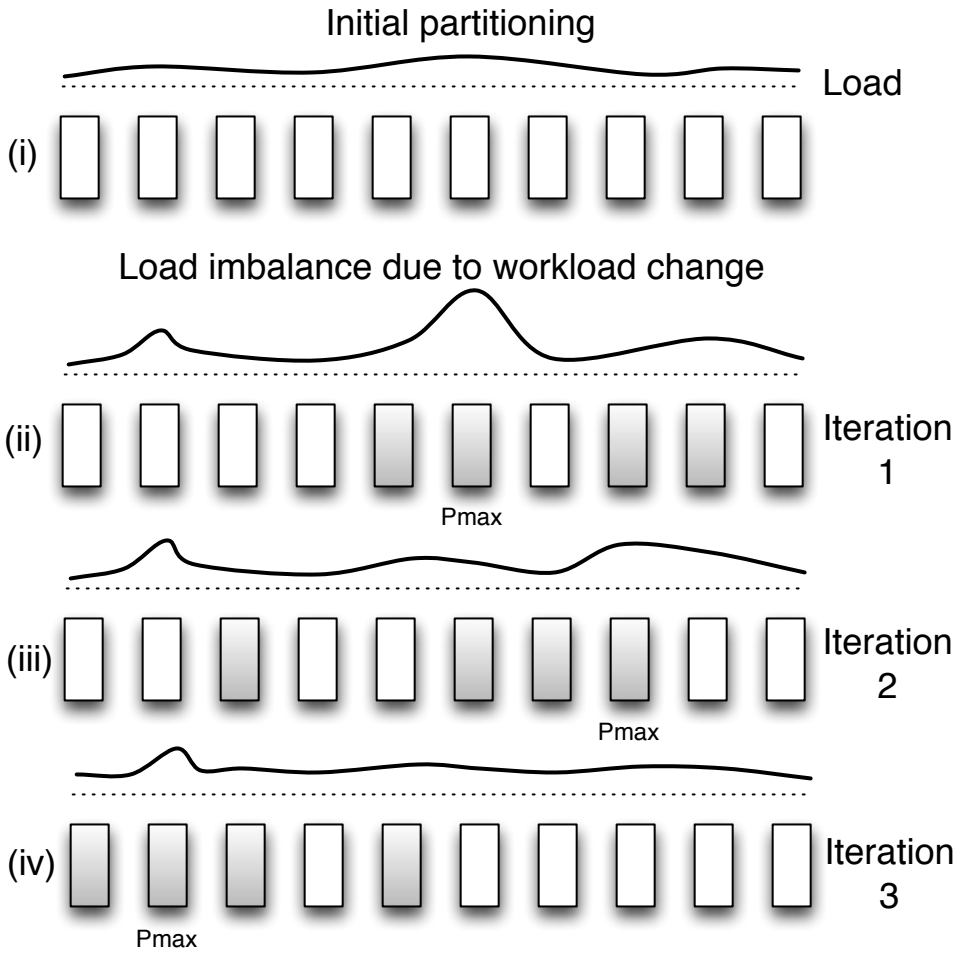


Figure 4.7: **Key idea of local load dispersion approach.**

P'_{max} is almost equal to the number of nodes in $P' \in \{\mathcal{P} - P'_{max}\}$, that is $|V'_{P'_{max}}| \simeq |V'_{P'}|$.

Simple Load Dispersion Algorithm (SimpleIncRep): We first develop a simple load dispersion algorithm to handle the variations in the load resulted by workload changes. Key idea of this algorithm is, in the case of $|V'_{P'_{max}}| >> |V'_{P'}|$ we move high degree nodes from partition P'_{max} to partition

$P' \in \{\mathcal{P} - P'_{max}\}$ so that load can be shifted from P'_{max} to P'. For the case $|V'_{P'_{max}}| \simeq |V'_{P'}|$, high degree nodes in P'_{max} are swapped with low degree nodes in P', so that load on P'_{max} is reduced. Although intuitively this algorithm is effective in terms of reducing load imbalance, but this approach is not search cost-aware, so it does not have control over search cost while incrementally repartitioning. To achieve both search cost and load awareness we need to make this approach workload-aware.

Workload-aware Load Dispersion Algorithm (WAIncRep): We improve the previous algorithm by carefully analyzing the workload while incrementally repartitioning the data across the partitions. In this approach, we incrementally move K nodes across the partitions in order to disperse the load from maximum load partition P'_{max} to other partitions $\mathcal{P}'_{sub}$ connected through hyperedges E'_{max} incident on P'_{max} where $\mathcal{P}'_{sub} \subseteq \{\mathcal{P} - P'_{max}\}$. Intuition behind choosing partitions that are connected to P'_{max} is that, swapping nodes between P'_{max} and partitions in $\mathcal{P}'_{sub}$ allows us to minimize cuts or at least reduce average query span that may have been affected by change in workload and at the same time dispersing the load from P'_{max} to other connected partitions in $\{\mathcal{P}'_{sub} - P'_{max}\}$. Key intuition here is that, when we move a high degree node d from P'_{max} to P'_{sub} to minimize the load on P'_{max}, there are three possibilities regarding the search cost, 1) query span of subset of

queries incident on c might decrease by one, 2) query span of subset of queries incident on c may remain same, and 3) query span of subset of queries incident on d might increase by one. In order to minimize the variation in search cost while moving the nodes across the partitions, we want to maximize 1) and 2) and minimize 3) for majority of the queries incident on c. To achieve this objective, we assign a gain score to each data node c. We sort these data items in P'_{max} in descending order. For the case $|V'_{P'_{max}}| >> |V'_{P'_{sub}}|$, we move K such data items at a time equally to the partitions in $\mathcal{P}'_{sub}$. Whereas for the case $|V'_{P'_{max}}| \simeq |V'_{P'_{sub}}|$, top K high gain nodes are swapped with top low degree nodes in the partition belonging the set $\mathcal{P}'_{sub}$. Such movement of data nodes, not only disperses the load from P'_{max} to the partitions in $\mathcal{P}'_{sub}$ but also carefully trades the load balancing with the search cost unlike the previous approach.

Calculating Gain: We begin by explaining our gain function. Let $\mathcal{H}''(V'', E'')$ be the sub-hypergraph such that $V'' = V \bigcap \mathcal{P}'_{sub}$ and $E'' = E'_{P'_{max}} \cap E'_{\mathcal{P}'_{sub}}$. If we want to move a data item c from P'_{max} to $P'_{sub} \in \mathcal{P}'_{sub}$, then let $|E_c|$ be the degree of c, in other words the number of hyperedges incident on c. Let $\mathcal{H}''_{common}(V''_{comm}, E''_{comm})$ be the sub-hypergraph with the hyperedges that are incident on both P'_{max} and P'_{sub} where $\mathcal{H}''_{common} \subseteq \mathcal{H}''$. Then $|E''_{comm_c}|$ be the number of hyperedges incident on both P'_{max} and P'_{sub}. Let E_{uncomm_c} be

the hyperedges that are incident on c but are not incident on P'_{sub}, that is $E''_{uncomm_c} = E_c - E''_{comm_c}$. So gain G_c can be calculated as:

$$G_c = \frac{|E''_{comm_c}|}{|E''_{uncomm_c}|} \tag{4.1}$$

Algorithm 8 shows the pseudocode of our algorithm. Algorithm runs until change in load as a result of algorithm steps shown in lines 1-8 is no more than small value ϵ. In line 2, we build a sub-hypergraph $\mathcal{H}''(V'', E'')$. Then if K is the fixed number of data nodes that we want to move around per iteration and if there are $\#\mathcal{P}'_{sub}$ number of partitions in $\mathcal{P}'_{sub}$ then we move $\frac{\text{K}}{\#\mathcal{P}'_{sub}}$ number of nodes across P'_{max} and $\mathcal{P}'_{sub}$ for each $P'_{sub} \in \mathcal{P}'_{sub}$. For each $P'_{sub} \in \mathcal{P}'_{sub}$ we calculate the common hyperedges E''_{comm} that are incident on both P'_{max} and P'_{sub}. Also we calculate $V''_{P'_{max}}$ as number of nodes in P'_{max} incident by hyperedges in E''_{comm}. Next, we sort the nodes in $V''_{P'_{max}}$ according to the gain values calculated according to Equation 4.1 in descending order using **sortByGain** method. Similarly, we calculate $V''_{P'_{sub}}$ as number of nodes in P'_{sub} incident by hyperedges in E''_{comm}. In this case, we sort the nodes in $V''_{P'_{sub}}$ according to the node degrees in ascending order using **sortByDegree** method. In lines 9-21, based on the nature of partitioning, we decide whether to just move nodes from P'_{max} to P'_{sub} or swap nodes across these two partitions. After repartitioning the nodes in $\mathcal{P}'_{sub}$ and P'_{max}, to deal with

uneven load dispersion we update current P'_{max} and calculate updated $\mathcal{P}'_{sub}$ and repeat the whole process until the current load imbalance is expected to be less than user defined threshold τ.

Algorithm 8 Workload-aware Load Dispersion Algorithm

Require: $\mathcal{H}(V, E)$, $\mathcal{H}'(V, E')$, $N, C, \mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ obtained after initial partitioning, P'_{max}, E'_{max}, $\mathcal{P}'_{sub} \subset \mathcal{P}$, L_{curr}, K.

1: **while** $\triangle_{L_{curr}} > \epsilon$ **do**
2: $\mathcal{H}'' =$ buildSubHypergraph($\mathcal{P}'_{sub}$, E');
3: $\#nodesToMove = \mathrm{K}/\#\mathcal{P}'_{sub}$;
4: **for** ($P'_{sub} \in \mathcal{P}'_{sub}$) **do**
5: $E''_{comm} = E'_{P'_{max}} \cap E'_{P'_{sub}}$;
6: $V''_{P'_{max}} = sortByGain(V'_{P'_{max}} \cap E''_{common}$, descending);
7: $V''_{P'_{sub}} = sortByDegree(V'_{P'_{sub}} \cap E''_{common}$, ascending);
8: $i = 0$;
9: **if** $((|V'_{max}| - |V'_{sub}|) >= \#nodesToMove)$ **then**
10: **for** ($v \in V''_{P'_{max}}$ && $i{+}{+} < \#nodesToMove$) **do**
11: $P'_{sub} \cup = v$; $P'_{max} - = v$;
12: **else**
13: **for** ($v_l \in V''_{P'_{max}}, v_m \in V''_{P'_{sub}}; i{+}{+} < \#nodesToMove$) **do**
14: $P'_{sub} \cup = v_l$; $P'_{sub} - = v_m$;
15: $P'_{max} \cup = v_m$; $P'_{max} - = v_l$;
16: $L_{curr} = \frac{max(d'_{\mathcal{P}})}{avg(d'_{\mathcal{P}})}$;
17: **if** $L_{curr} <= \tau$ **then**
18: **break**;
19: $P'_{max} = P_{argmax(d'_{P_1}, d'_{P_2}, \ldots, d'_{P_N})}$;
20: $\mathcal{P}'_{sub} = \mathcal{P}(E'_{max})$;
21: **if** $\mathcal{P}'_{sub}$ is hot **then**
22: wait for lean activity period;

4.4 Experiments

4.4.1 Setup

WAFEL is a prototype for smart and effective web document document clusters-machine assignment framework for large-scale information retrieval. We conduct our experiments on single scaled-up machine with dual Intel Xeon quad-core processors (E5620 2.4 GHz) and 128 GB RAM. This architecture has a 64KB L1 cache per core, a 256KB L2 cache per core, and a 12MB L3 cache shared by all cores of a single processor. With hyperthreading, this machine can support up to 16 threads concurrently. The machine has six 2TB 7200 RPM disks, each with 64MB cache, arranged in a RAID6 configuration. On this machine, we create several logical disk partitions and regard them as individual physical partitions. This setup does not take away the generality of our system when deployed in distributed setting as our techniques are not network communication intensive as web queries are simple in nature and do not involve data joins in them.

We perform our experiments on real-word dataset and queries. For real-world dataset, we work with TREC Category B Section 1 dataset which consists of 50 million English pages. For queries we consider 40 million AOL search queries. In order to process these 50 million documents to clusters, we perform K-means clustering using Apache Mahout where K=10000. We

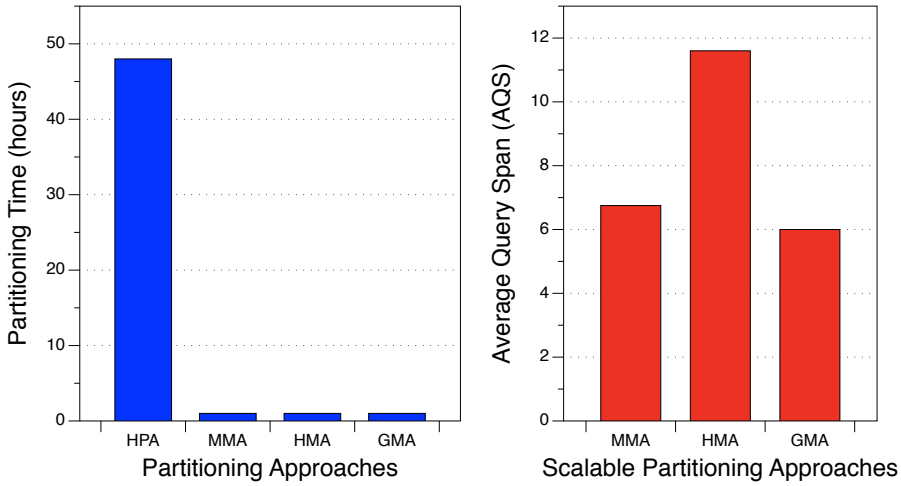
consider each cluster as a data item in this work. Also, we divide our query dataset into two parts, first part has 30 million queries that is used as history for analyzing and applying techniques discussed in this work. Then we evaluate our techniques on test dataset which is the second part containing 10 million queries. We use hMETIS 1.5 for state-of-the-art hypergraph partitioning algorithm (HPA). Unless specified we use unbalanced factor of 35 for HPA.

4.4.2 Effectiveness of our Scalable Dense Hypergraph Partitioning

Figure 4.4 shows the example of HMA and GMA approaches where a dense hypergraph $\mathcal{H}^m$ is split into three smaller hypergraphs $\mathcal{H}^m = \{\mathcal{H}_1^m, \mathcal{H}_2^m, \mathcal{H}_3^m\}$. HPA is then run on these smaller hypergraphs individually to get three different node to partition mappings shown in different colors in the Figure. For HMA, these node to partition mappings are then converted to a single hypergraph where each partition is a hyperedge and data items are the nodes. For GMA, node to partition mappings are modeled as a graph, where nodes in a partition form a clique.

We conduct an experiment to evaluate effectiveness of our scalable partitioning approaches where M=120. Our first result shows that our scalable partitioning approaches (MMA, HMA, and GMA) are atleast 40X times

faster than HPA, Figure 4.8a) shows the result. This speedup is because of parallel executions of HPA on smaller hypergraph splits. Figure 4.8b) shows that MMA and GMA approaches result in better average query spans when compared to HMA; this happens because of sparsity problem in HMA whereas GMA solves this problem.



(a) (b)

Figure 4.8: **a) Comparison of our novel scalable hypergraph partitioning approaches with state-of-the-art HPA b) Comparison between our scalable hypergraph partitioning approaches.**

Next, we evaluate the performance of our workload-aware data placement framework. For this evaluation, we increase the number of partitions from 10 to 50 and measure the average query span (AQS) for the random data placement and distributed exhaustive search comparing with our workload-aware data placement and distributed selective search. In distributed exhaustive search, query is shipped to all the partitions and ranked list of documents

is returned from each partition, therefore we note that average query span equals the number of partitions at each step. In short, our experiment shows that the distributed exhaustive search result in very high search costs. In WAFEL's distributed selective search, because each search query is sent to relatively fewer machines when compared with distributed exhaustive approach, as we vary number of partitions, our approach gives up to 75% reduction in the search cost.

We also compare WAFEL's selective search with state-of-the-art selective search where after performing K-means clustering on the web documents to get documents clusters, these clusters are then partitioned randomly into partitions. Then, for each query top $x\%$ of the document clusters are selected, this is accomplished by sending the query to the partitions that contains the clusters relevant to the query. In this experiment, $x = 10\%$. To find out relevant clusters for a given query, we maintain a separate index consists of all centroid documents of the clusters. Given a query we obtain a ranked list of these centroids and send the query to the partitions containing the top 10% clusters corresponding to the centroids. Our results demonstrates that WAFEL's selective search significantly performs well and gives up to 74% reduction in search cost or average query span when compared with distributed selective search baseline. Figure 4.9 shows the result.

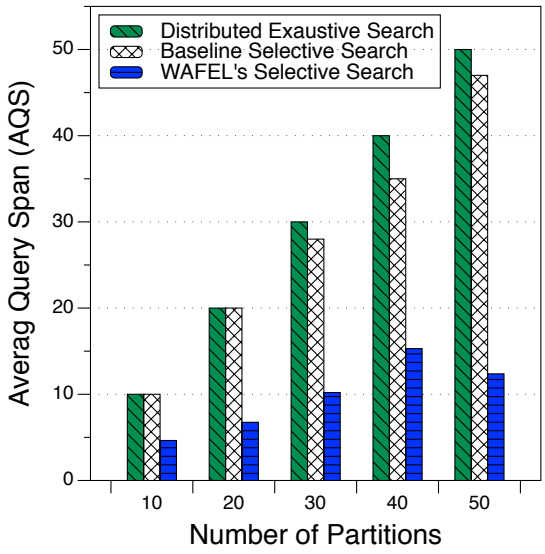


Figure 4.9: **Effect of distributed exhaustive search and WAFEL's selective search on the overall search cost.**

4.4.3 Study of Different Routing Techniques

As we experiment with different routing schemes as shown in the Figure 4.10, we measure two important metrics, first is average query span (AQS) for the given set of queries and second is load imbalance (LI) at the partitions. In order to evaluate effectiveness of these schemes, we come up with a metric called as *load constraint*. *Load constraint* is defined as ratio of total number of queries to the maximum number of queries allowed to serve per partition. As we perform set-cover based routing (S-Routing) of queries over workload-aware partitioning with replication, we note that query costs are minimized significantly. This scheme only minimizes the query spans and is not at all load-aware which leads to severe load imbalance. Also, as

this scheme is not load-aware, varying load constraint has no effect on this routing scheme. Whereas S-Routing-RR scheme because of its round robin access of replicas achieves exactly same average query span as S-Routing but with relatively lower load imbalance.

Next we experiment with load-aware set-cover based routing scheme (LAS-Routing). In this scheme, because they have high load, some of the partitions are not considered in the set-cover computation which causes increase in query span for some of the queries. We observe that as load constraint on the partitions is increased, this routing scheme chooses next best set of partitions (in-terms of query span) containing all the data items required for the query which increases the average query span gradually. With increase in load constraint, we observe the trade-off between query span and the load imbalance where as query span increases load imbalance starts to decrease. In other words, when load constraint is very flexible then LAS-Routing is able to achieve minimum query cost, whereas when load constraint is extremely tight, then this scheme achieves best load imbalance while trading query cost/span. When compared to S-Routing-RR, the LAS-Routing results in 27× times lower load imbalance by increasing the query spans by just 2× times.

We compare our techniques with two baseline routing techniques, Random-Exaustive and Random-Selective techniques. Routing-Exaustive approach

sends the query to all the partitions. Routing-Selective approach sends the query to the partitions containing top $p\%$ clusters and their replicas. Both baselines give very high query spans with perfect load balancing. On the other hand, we observe that simple load-aware routing with no set-cover computation (LA-Routing) on a smart workload-aware partitioned system performs almost same as a system with random partitioning and routing. When compared to the baseline techniques our smart LAS-Routing can give 2.2× times lower query spans with almost same load imbalance that too under very tight load constraints (LC=2).

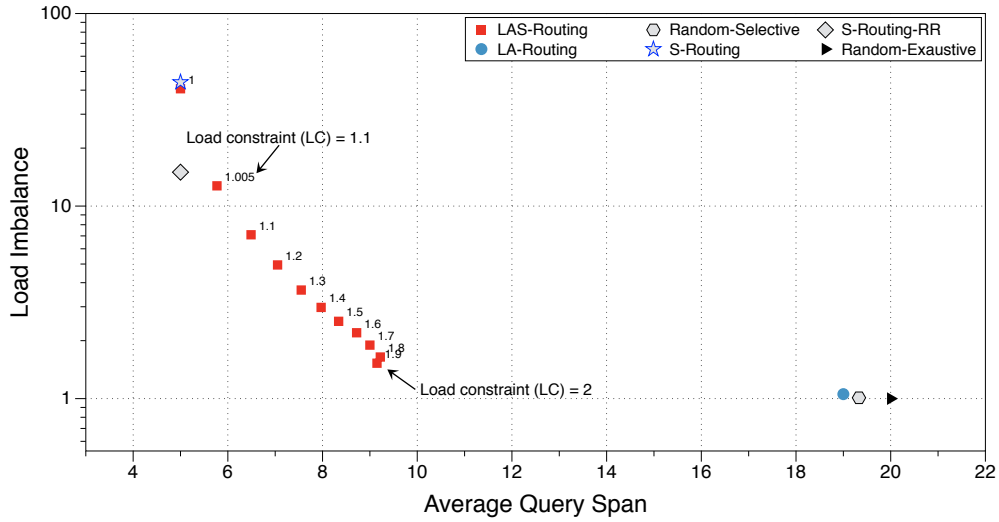
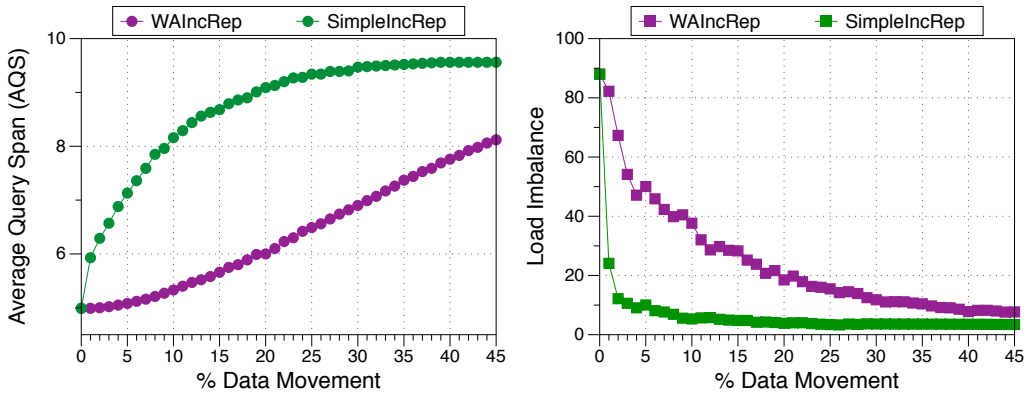


Figure 4.10: **Comparison between different routing techniques. Note that y-axis is in log-scale.**

4.4.4 Effect of Cost- and Load-aware Incremental Repartitioning

To demonstrate the true potential of our incremental repartitioning algorithms, we perform our workload-aware data partitioning with replication as discussed in Section 4.3.1.1 to minimize query spans and then route the queries using our set-cover based router (S-Routing). As we know that S-Routing only focuses on selecting the minimum number of partitions to minimize query spans significantly and completely ignores the load imbalance, we witness a very high load imbalance. Now over this partitioning and routing, we apply our incremental repartitioning techniques where our goal is to minimize the load imbalance and variation in query spans while moving only subset of nodes. Plots in Figure 4.11 show our experimental result. Figure 4.11 a) shows the effect of incremental data movement on search cost or average query span (AQS) whereas Figure 4.11 b) shows the effect on load imbalance (LI). We observe that although SimpleIncRep approach minimizes the load imbalance significantly even for very small percentage of data movement, but it abruptly increases the average query span. Whereas our workload-aware approach WAIncRep carefully trades the load imbalance minimization with search cost AQS as we perform incremental data movement.

To understand this result more carefully we introduce a metric known as



(a) (b)

Figure 4.11: **Comparison of incremental repartitioning algorithms, a) Effect of amount of data movement on average query span, b) Effect of amount of data movement on load imbalance.**

Gain Index (GI). Key idea behind *Gain Index* (GI) is that, it indicates an effectiveness of incremental repartitioning technique regarding how effectively it has traded load balancing with the search cost. GI can be calculated as following:

$$Gain\ Index\ (GI) = \frac{\%\ decrease\ in\ load\ imbalance}{\%\ increase\ in\ search\ cost} \tag{4.2}$$

In short, bigger the GI better the effectiveness of the incremental repartitioning algorithm. If LI_{before} and LI_{after} is the load imbalance before and after performing incremental repartitioning, then *% decrease in load imbalance* can be calculated by:

$$\%\ decrease\ in\ load\ imbalance = \frac{(LI_{before} - LI_{after})}{LI_{before}} \times 100 \tag{4.3}$$

Similarly *% increase in search cost* can be calculated by:

$$\% \; increase \; in \; search \; cost = \frac{(AQS_{after} - AQS_{before})}{AQS_{before}} \times 100 \qquad (4.4)$$

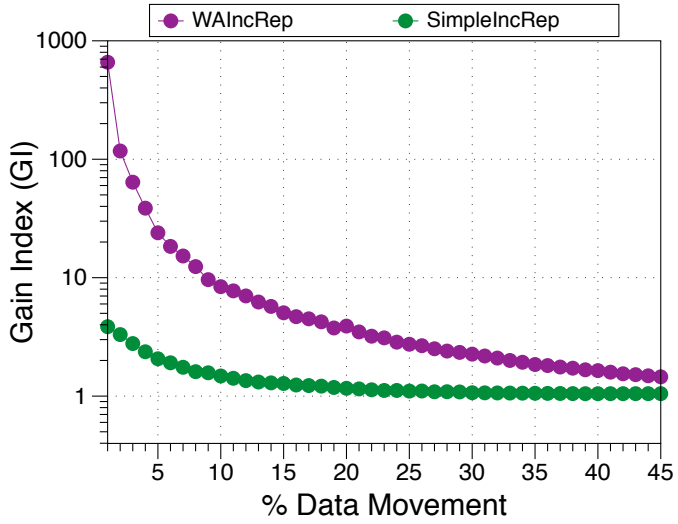


Figure 4.12: **Comparison of incremental repartitioning strategies in terms of *Gain Index* (GI). Note that Y-axis is in log scale.**

Figure 4.12 shows that comparison of incremental repartitioning strategies that we have developed in terms of *Gain Index* (GI). We make several observations, first is that, as the algorithm progresses, the number of iterations increase, in-turn increasing % of data movement we note that *Gain Index* GI sub-linearly decreases and remains constant after a certain point. Second is, we note that from $1 - 15\%$ data movement, WAIncRep is $4\times$ to $171\times$ times effective than SimpleIncRep approach, whereas for $15 - 30\%$ data movement, WAIncRep is $2\times$ to $4\times$ times effective than SimpleIncRec. Finally,

for $30 - 45\%$ data movement, our workload-aware approach is $1.3\times$ to $2\times$ times effective in terms of GI when compared to SimpleIncRec approach. In conclusion, our workload-aware incremental repartitioning technique (WAIncRep) consistently performs better than SimpleIncRep in minimizing the load imbalance while carefully trading the search cost.

4.5 Summary of Contributions

In this chapter, we presented the design of WAFEL, a data placement framework for cost effective distributed information retrieval. This system optimizes two metrics, first is, *search cost* and second is, *load imbalance*. To minimize these two metrics, WAFEL takes a two pronged approach, firstly to minimize the *search cost* it implements a workload-aware data partitioning to minimize the average query span of the queries. We show that our novel scalable workload-aware data partitioning atleast $40\times$ times faster than state-of-the-art techniques and it minimizes the search cost atleast 75% when compared to distributed exaustive search. Secondly, to minimize the *load imbalance*, our system implements a load-aware replication and load-aware set cover algorithm for routing. Finally, in order to handle workload changes or abrupt load-imbalances, our system provides a smart and effective framework for incrementally repartitioning the documents clusters K clusters at a time where load-balancing is carefully traded with search cost. We show that

our workload-aware incremental repartiitoning approach is 171× times more effective in trading *search cost* and *load imbalance* than the smart baseline. We believe that our approaches are general enough to be applied in data placement scenarios in several different data domains.

Chapter 5

HONE: "Scaling Down" Hadoop on Shared-Memory Systems

Till here, we have seen that by minimizing *query spans*, overall resource consumption of a given workload can be minimized. As we have discussed in Chapter 1, underlying system achieves highest resource efficiency when the data required by a job fits in single machine's memory and the job can be executed on a single machine (i.e., *query span*=1) as fast as possible. In this chapter, we discuss one such system called HONE that we have developed which is a multi-threaded in-memory map reduce system that is intended to be Hadoop API compatible. Using HONE one can execute an existing

Hadoop job on a single machine efficiently.

5.1 Introduction

The Hadoop implementation of MapReduce [23] has become the tool of choice for "Big Data" processing (whether directly or indirectly via higher-level tools such as Pig or Hive). Among its advantages are the ability to horizontally scale to petabytes of data on thousands of commodity servers, easy-to-understand programming semantics, and a high degree of fault tolerance. There has been much activity in applying Hadoop to problems in data management as well as data mining and machine learning. Over the past several years, the community has accumulated a significant amount of expertise and experience on how to recast traditional algorithms in terms of the restrictive primitives *map* and *reduce*.

Computing environments have evolved substantially since the introduction of Hadoop. For example, in 2008, a typical Hadoop node might have two dual-core processors with a total of 4 GB of RAM. Today, a high-end commodity server might have two eight-core processors and 256 GB of RAM—such a server can be purchased for roughly $10,000 USD. This means that a single server today has more cores and more memory than did a small Hadoop cluster from a few years ago. The assumption behind Hadoop and the need for distributed processing is that the data to be analyzed cannot

be held in memory on a single machine. Today, this assumption needs to be re-evaluated.

Although it is true that petabyte-scale datastores are becoming increasingly common, it is unclear whether datasets used in "typical" analytics tasks today are really too large to fit in memory on a single server. Of course, organizations such as Yahoo, Facebook, and Twitter routinely run Pig or Hive jobs that scan terabytes of log data, but these organizations should be considered outliers—they are not representative of data analytics in most enterprise or academic settings. Even still, according to the analysis of Rowstron et al. [75], at least two analytics production clusters (at Microsoft and Yahoo) have median job input sizes under 14 GB and 90% of jobs on a Facebook cluster have input sizes under 100 GB. Holding all data in memory does not seem too far-fetched.

There is one additional issue to consider: over the past several years, the sophistication of data analytics has grown substantially. Whereas yesterday the community was focused on relatively simple tasks such as natural joins and aggregations, there is an increasing trend toward data mining and machine learning. These algorithms usually operate on more refined, and hence, smaller datasets—typically in the range of tens of gigabytes.

These factors suggest that it is worthwhile to consider in-memory data analytics on modern servers—but it still leaves open the question of how we or-

chestrate computations on multi-core, shared-memory machines. Should we go back to multi-threaded programming? That seems like a step backwards because we embraced the simplicity of MapReduce for good reason—the complexity of concurrent programming with threads is well known. Our proposed solution is to "scale down" Hadoop to run on shared-memory machines [51]. In this paper, we present a prototype runtime called HONE ("Hadoop One") that is API compatible with standard (distributed) Hadoop. That is, we can take an existing Hadoop algorithm and efficiently run it, without modification, on a multi-core, shared-memory machine using HONE. This allows us to take an existing application and find the most suitable runtime environment for execution on datasets of varying sizes—if the data fit in memory, we can avoid network latency and significantly increase performance in a shared-memory environment.

Hadoop API compatibility is the central tenet in our design. Although there are previous MapReduce implementations for shared-memory environments, taking advantage of them would require porting Hadoop code over to another API. In contrast, HONE is able to leverage existing implementations. In this paper, we present experiments on a number of "standard" MapReduce algorithms (word count, PageRank, etc.) as well as a Hadoop implementation of Latent Dirichlet Allocation (LDA). This implementation represents a major research effort [93] and demonstrates API compatibility

on a non-trivial application.

5.2 Hadoop on Single Machine

We begin by discussing why Hadoop does not perform well on a single machine. To take advantage of multi-core architectures, Hadoop provides pseudo-distributed mode (PDM henceforth), in which all daemon processes run on a single machine (on multiple cores). This serves as a natural point of comparison, and below we identify several disadvantages of running Hadoop PDM.

Multi-process overhead: In PDM, mapper and reducer tasks occupy separate JVM processes. In general, multi-process applications suffer from inter-process communication (IPC) overhead and are typically less efficient than an equivalent multi-threaded implementation that runs in a single process space.

I/O Overhead: Another disadvantage of Hadoop PDM is the overhead associated with reading from and writing to HDFS. To quantify this, we measured the time it takes to read and write a big file (8 GB) and a single split of the file (64 MB) using HDFS as well as directly using Java file I/O (on the server described in Section 5.4.1). These results are shown in Table 5.1.

We find that direct reads from disk are much faster than reads from HDFS for both the 8 GB and 64 MB conditions. Performance improvements are

	Write 8 GB		Read 8 GB	
	Cold Cache	Warm Cache	Cold Cache	Warm Cache
HDFS	178.0s	32.7s	81.4s	28.9s
Disk	194.0s	25.3s	27.1s	1.7s
	Write 64 MB		**Read 64 MB**	
	Cold Cache	Warm Cache	Cold Cache	Warm Cache
HDFS	5.10s	1.72s	5.64s	2.12s
Disk	0.47s	0.11s	3.27s	0.20s

Table 5.1: **Performance comparisons between HDFS and direct disk access.**

observed under both cold and warm cache conditions, and the magnitude of improvement is higher under a warm cache. Interestingly, we find that writing 8 GB is faster using HDFS, but in all other conditions HDFS is slower. In the small data case (64 MB), writes are over a magnitude faster under both cold and warm cache conditions. These results confirm that disk I/O operations using HDFS can be extremely expensive [27, 60] when compared to direct disk access. In Hadoop PDM, mappers read from HDFS and reducers write to HDFS, even though the system is running on a single machine. Thus, Hadoop PDM suffers from these HDFS performance issues.

Framework overhead: Hadoop is designed for high-throughput processing of massive amounts of data on potentially very large clusters. In this context, startup costs are amortized over long-running jobs and thus do not

have a large impact on overall performance. Hadoop PDM inherits this design, and in the context of a single machine running on modest input data sizes, job startup costs become a substantial portion of overall execution time.

Hadoop PDM on a RAM disk provides negligible benefit: One obvious idea is to run Hadoop PDM using a RAM disk to store intermediate data. RAM disks tend to help most with random reads and writes, but since most Hadoop I/O consists of sequential operations, it is not entirely clear how much a RAM disk would help. Our initial experiments with Hadoop PDM did explore replacing rotational disk with RAM disk. We ran evaluations on the applications in Section 5.4.2, but results showed no benefits when using a RAM disk. Moreover, previous studies have shown that a RAM disk is at least four times slower than raw memory access [45,62]. We expected greater benefits by moving completely to managing memory directly, so we did not pursue study of Hadoop PDM on RAM disks any further.

5.3 HONE Architecture

The overall architecture of HONE is shown in Figure 5.1. Below, we provide an overview of each processing stage.

Mapper Stage: As in Hadoop, this stage applies the user-specified mapper to the input dataset to emit intermediate (key, value) pairs. Each mapper

is handled by a separate thread, which consumes the supplied *input split* (i.e., portion of the input data) and processes input records according to the user-specified InputFormat. Like Hadoop, the total number of mappers is determined by the number of input splits. This stage uses a standard thread-pooling technique to control the number of mapper tasks that execute in parallel. Mappers in HONE accept input either from disk or from a *namespace* residing in memory (see Section 5.3.2 for more details).

Data Shuffling Stage: In MapReduce, intermediate (key, value) pairs need to be shuffled from the mappers where they are created to the reducers where they are consumed. In Hadoop, data shuffling is interwoven with sorting, but in HONE these are two separate stages. The next section describes three different approaches to data shuffling. Overall, we believe that efficient implementations of this process is the key to a high-performance MapReduce implementation on multi-core, shared-memory systems.

Sort Stage: As with Hadoop, HONE sorts intermediate (key, value) pairs emitted by the mappers. Sorting is handled by a separate thread pool with a built-in load balancer, on streams that have already been assigned to the appropriate reducer (as part of the data shuffling stage). If the sort streams grow too large, then an automatic splitter divides the streams on the fly and performs parallel sorting on the split streams. The split information is passed

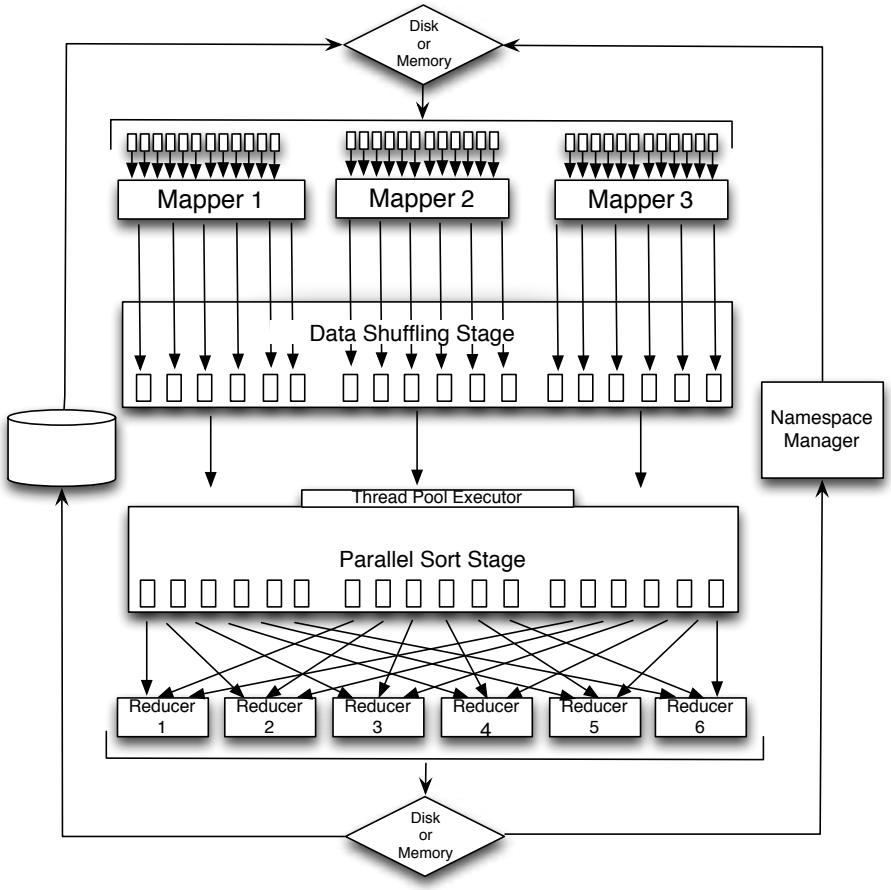


Figure 5.1: **HONE system architecture.**

to the reducer stage for proper stream assignment. The default stream split size can be set as part of the configuration.

Reducer stage: In this stage, HONE applies the user-specified reducer on values associated with each intermediate key, per the standard MapReduce programming model. A reducer either writes output (key, value) pairs to disk or to memory via the namespace abstraction for further processing.

Combiners: In a distributed setting, combiners mimic the functionality of

reducers locally on every node, serving as an optimization to reduce network traffic. Proper combiner design is critical to the performance of a distributed MapReduce algorithm, but it is unclear whether combiners are useful when the entire MapReduce application is running in memory on a single machine. For this reason, HONE currently does not support combiners: since they are optional optimizations, we can ignore them without affecting algorithm correctness.

Namespace Manager: This module manages memory assignment to enable data reading and writing for MapReduce jobs. It converts filesystem paths that are specified in the Hadoop API into an abstraction we call a *namespaces*: output is directed to an appropriate namespace that resides in-memory, and, similarly, input records are directly consumed from memory as appropriate.

5.3.1 In-Memory Data Shuffling

We propose three different approaches to implement data shuffling between mappers and reducers: (1) the pull-based approach, (2) the push-based approach, and (3) the hybrid approach. These are described in detail below.

Pull-based Approach: In the pull-based approach, each mapper emits keys to r streams, where r is the number of reducers. Each mapper applies the partitioner to assign each intermediate (key, value) pair to one of the r streams

based on the key (per the standard MapReduce contract). If m is the number of mappers, then there will be a total $m \times r$ intermediate streams. In the sort stage, these $m \times r$ intermediate streams are sorted in parallel. In the reducer stage, each reducer thread *pulls* from m of the $m \times r$ streams (one from each mapper). Figure 5.2 shows an example with three mappers and six reducers, with eighteen intermediate streams.

With this approach we encounter an interesting issue regarding garbage collection. In Java, a thread is its own garbage collection (GC) root. So any time a thread is created, irrespective of creation context, it will not be ready for garbage collection until its run method completes. This is true even if the local method which created the thread completes. In HONE, we maintain a pool of mapper threads containing tp_m threads (usually for large jobs, $tp_m \ll m$, where m is the number of mappers, determined by the split size). Thus, tp_m mappers are running concurrently, and the objects that each creates cannot be garbage collected until the mapper finishes. Increasing this thread pool size allows us to take advantage of more cores, but at the same time this increases the amount of garbage that cannot be collected at any given time. HONE needs to contend with the characteristics of the JVM, and garbage collection is one re-occurring issue we have faced throughout this project.

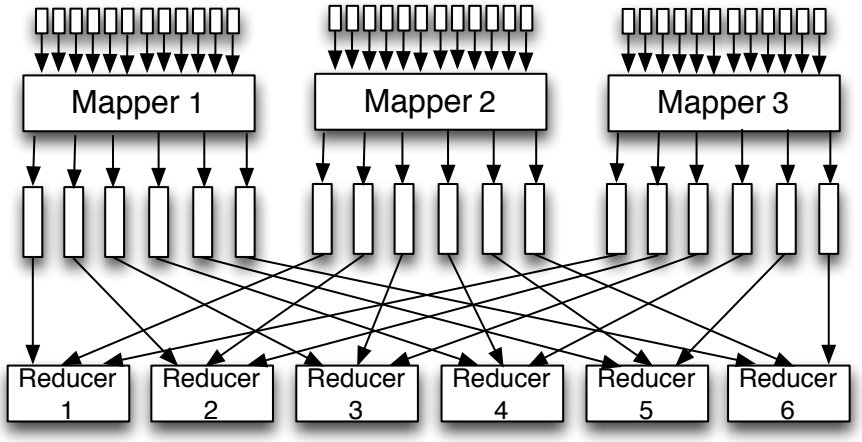


Figure 5.2: **Pull-based approach**

Push-based Approach: In this approach, HONE creates only r intermediate streams, one for each reducer. This is shown in Figure 5.3, where we have six streams. Each mapper emits intermediate (key, value) pairs directly into one of those r streams based on the partitioner. In this way, the mappers *push* intermediate data over to the reducers. Because r streams are being updated by the mappers in parallel, these streams must be synchronized and guarded by locks. Due to this synchronization overhead, contention is unavoidable, but this cost varies based on the distribution of the intermediate keys to reducers. There are two ways of dealing with contention cost: the first is to employ scalable and efficient locking mechanisms (more discussion below), and second is to increase the number of reducers so that key distribution to reducers is spread out, which in turn will reduce synchronization overhead. However, if we have too many reducers, context-switching

overhead of reducer threads will negatively impact performance.

The push-based approach creates fewer intermediate data structures for the same amount of intermediate (key, value) pairs, and thus in this manner is more efficient. In the pull-based approach, since *each* mapper output is divided among r streams, the object overhead in maintaining those streams is much higher relative to the actual data held in those streams (compared to the push approach). In order to take advantage of greater parallelism in the reducer stage (for the pull-based approach), we may wish to increase the number of reducers, which further exacerbates the problem.

Another advantage of the push-based approach is that reducers are only consuming from a single stream, so we would expect better reference locality (and the benefits of processor optimizations that may come from more regular memory access patterns) compared to the pull-based approach. The downside, however, is synchronization overhead since all the mapper threads are contending for write access to the reducer streams.

Hybrid Approach: As a middle ground between the pull and push approaches, we introduce a hybrid approach that devotes a small number of streams to each reducer. In Figure 5.4, each reducer reads from two streams, which is the default. There are two ways to distribute incoming (key, value) pairs to multiple streams for each reducer: the first is to distribute evenly,

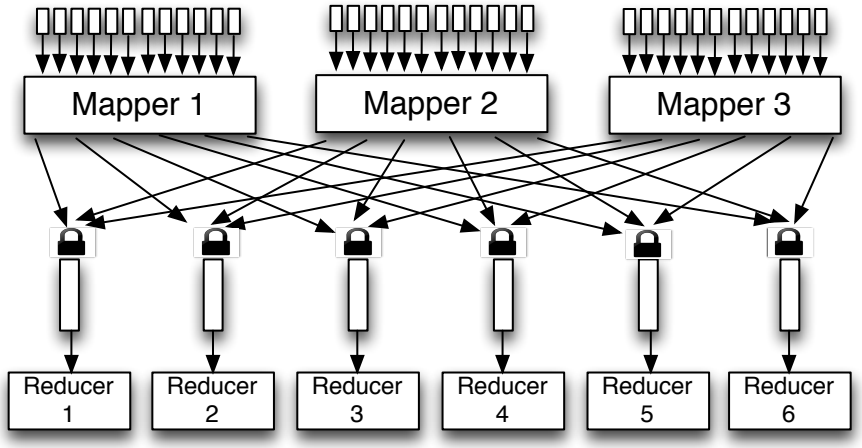


Figure 5.3: **Push-based approach**

and the second is to distribute according to the current lock condition of a stream. The second approach is perhaps smarter, but HONE currently implements the first method, which we empirically discovered to work well. Having multiple streams reduces, but does not completely eliminate lock contention, but at the same time, the hybrid approach does not suffer from a proliferation of stream objects. The number of streams per reducer can be specified in a configuration, which provides users a "knob" to find a sweet spot between the two extremes.

In the push and hybrid data-shuffling approaches, lock efficiency plays an important role in overall performance. We have implemented and experimented with various lock implementations, including Java synchronization, test-and-set (tas) spin lock, test and test and set (ttas) spin lock, and reentrant lock. Each lock implementation has its own advantages and disadvan-

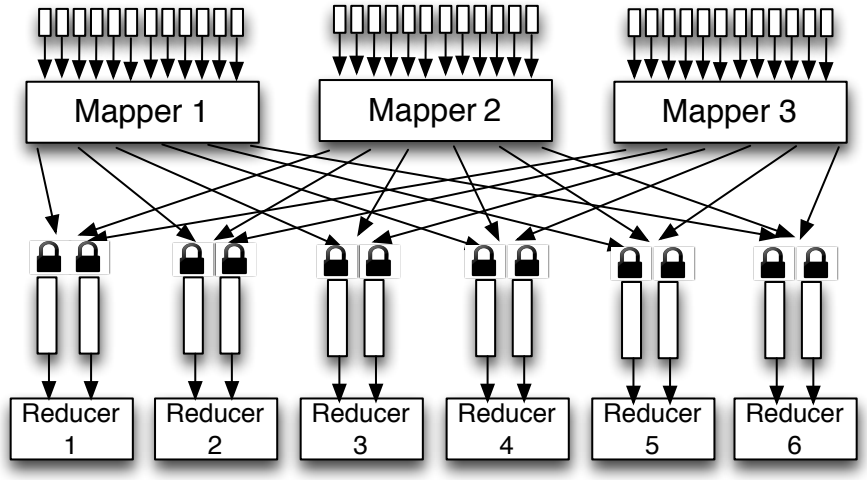


Figure 5.4: **Hybrid approach**

tages, but overall we find that Java synchronization in JDK7 performs the best.

Tradeoffs: We experimentally compare the three different data-shuffling approaches, but we conclude this section with a discussion of the factors that may impact performance.

Obviously, input data size is an important factor. Larger inputs translate into more splits, more mappers, and thus more active streams that are held in memory (for the pull-based approach). In contrast, there are only r streams in the push-based approach, where r is the number of reducers. Note that the number of reducers is a user-specified parameter, unlike the number of mappers, which is determined by the input data. As previously discussed, the cost of fewer data streams (less object overhead) is synchronization costs

and contention when writing to those streams. The hybrid approach tries to balance these two considerations.

Another factor is the amount of intermediate data that is produced. Some MapReduce jobs are primarily "reductive" in that they generate less intermediate data than input data, but other types of applications generate more intermediate data than input data; some text mining applications, for example, emit the cross product of their inputs [56]. This characteristic may have a significant impact on the performance of the three data-shuffling approaches.

Finally, the distribution of the intermediate keys will play an important role in performance—this particularly impacts synchronization overhead in the push-based approach. For example, with that approach, if the distribution is Zipfian (as with word count and certain types of graph algorithms), then increasing the number of reducers may not substantially lower contention, since the "head" of the distribution will always be assigned to a single reducer [57]. On the other hand, if the intermediate key distribution is more uniform, we would expect less lock contention since mapper output would be more evenly distributed over the reducer streams, reducing the chance that multiple mappers are contending for a lock.

5.3.2 Challenges and Solutions

This section discusses key challenges in developing HONE for the Java Virtual Machine on multi-core, shared-memory architectures and how we addressed them.

Memory consumption: To retain compatibility with Hadoop, we made the decision to implement HONE completely in Java, which meant contending with the limitations of the JVM. In a multi-core, shared-memory environment, the mapper, sort, and reducer threads compete for shared resources, and thus we must be careful about the choice of data structures, the number of object creations, proper de-referencing of objects for better garbage collection, etc. We discovered early that many Java practices scale poorly to large datasets. With a naïve implementation based on standard Java collections, on a server with 128 GB RAM, an initial implementation of MapReduce word count on an input size 10% of the total memory generated out-of-memory errors because standard Java collections are heavyweight [64]. For example, an implementation using a Java `TreeMap<String, Integer>` to hold intermediate data can have up to 95% overhead, i.e., only 5% of the memory consumed is used for actual data.

To address this issue, we extensively use primitive data structures such as byte arrays to minimize JVM-related overhead. In the mapper stage, (key,

value) pairs are serialized to raw bytes and in the reducer stage, new object allocations are reduced by reading pairs from byte arrays using bit operations and reusing container objects when possible. We avoid using standard Java collections in favor of more efficient custom implementations.

Sorting is expensive: Intermediate (key, value) pairs emitted by the mappers need to be sorted by key. For large intermediate data (on the order of GBs), we found sorting to be a major bottleneck. This is in part because operations such as swapping objects can be expensive, but the choice of data structures has a major impact on performance also. In Hadoop, sorting is accomplished by a combination of in-memory and on-disk operations. In HONE, however, everything must be performed in memory.

We experimented with two approaches to sorting. In the first, each thread from the mapper thread pool handles both mapper execution as well as sorting. In the second approach, mapper execution and sorting are handled by separate thread pools. We ultimately adopted the second design. Note that sorting is orthogonal to the pull, push, hybrid data-shuffling approaches.

We see a number of advantages to our decoupled approach. First, the optimal thread pool size depends on factors such as the number of cores available, the size of the intermediate data, and skew in the intermediate key distribution. The decoupled approach lets us configure the sort thread

pool size based on these considerations, independent of the mapper thread pool size. Second, the decoupled approach allows the garbage collector to clean up memory used by the mapper stage before moving to the sort stage. Finally, combining mapping and sorting creates a mix of different memory access patterns, which can negatively impact performance.

HONE implements a custom quick sort that works directly on byte arrays; these are the underlying implementations of the streams that the mappers write to when emitting intermediate data. The main idea is to store intermediate (key, value) pairs in a data byte array in serialized form, and to create an offset array that records offset information corresponding to the serialized objects in the data byte array. Offsets are also stored in byte arrays. Once mapper output is stored in these data and offset byte arrays, quick sort is applied. Offsets are read from the offset array and data are read using bit operations depending on the data type (to avoid object materialization whenever possible). Values are compared with each other, but only offset bytes are swapped. Usually, the size of the offset byte array is much less than the size of the data byte array, and therefore it is more efficient to perform swapping on the offset byte array. Moreover, most of the bytes in the offset byte array contain zeros (i.e., the high order bytes of an offset): only the non-zero bytes and the bytes that are not equal need to be swapped. This eliminates a large amount of the total cost of swapping elements during

sorting.

Interactions between data shuffling and sorting: In the pull-based approach to data shuffling described in Section 5.3.1, the sort stage takes maximum advantage of parallelism since the intermediate data are divided among $m \times r$ streams (usually a large number). However, in the push and the hybrid approaches, intermediate data are held in a much smaller number of streams: r in the case of the push approach and a small factor of r for the hybrid approach. In both cases, this reduces the amount of parallelism available, since each sorting thread must handle a much larger amount of data. For large datasets, this becomes a performance bottleneck. For the push and hybrid approaches, we remedy this by splitting intermediate streams into several smaller streams on the fly. The sizes of these streams is a customizable parameter, but we have heuristically settled on a value that works reasonably well across different applications (10000 bytes).

Disk-based readers and writers: One of the challenges in developing a Hadoop-compatible MapReduce implementation is that Hadoop application code makes extensive use of disk-based readers and writers, mainly implemented using the RecordReader and RecordWriter classes. The simplest way to avoid disk overhead is to provide an API to access memory directly and then change the application code to take advantage of these hooks. Since

we wanted to make HONE compatible with the existing Hadoop API, we needed deeper integration.

We introduce the notion of a *namespace*, which is a dedicated region in memory where data are stored. Application code can access namespaces through the standard Hadoop `Job` object. To maintain compatibility with the Hadoop API, we provide efficient in-memory alternatives for Hadoop FileReader and FileWriter classes. Disk paths in application code are automatically converted to appropriate namespaces and output is redirected to these namespaces.

Iteration support: Iterative MapReduce algorithms, where a sequence of MapReduce jobs are chained together such that the output of the previous reducer stage serves as the input to the next mapper stage, are a well-known weakness of Hadoop [13]. Since many interesting algorithms are iterative in nature (e.g., PageRank, LDA), this is an important problem to solve. The primary issue with Hadoop-based implementations of iterative algorithms is that reducer output at each iteration *must* be serialized to disk, even though it should be considered temporary since the data are immediately read by mappers at the next iteration. Of course, serializing serves the role of checkpointing and provides fault tolerance, but since Hadoop algorithms are forced to do this at *every* iteration, there is no way to trade off fault

tolerance for speed.

In HONE, all of these issues go away, since intermediate data reside in memory at the end of each iteration. The choice to serialize data to disk can be made independently by the developer. Thus, HONE provides natural support for iterative MapReduce algorithms. In a bit more detail: typically, in an iterative algorithm, there is a "driver program" that sets up the MapReduce job for each iteration, checks for convergence, and decides if another iteration is necessary. Convergence checks are usually performed by reading reducer output (i.e., files on HDFS). In HONE, this is transparently handled by our notion of namespaces.

Garbage collection and off-heap memory allocation: All objects allocated in the JVM heap are scanned periodically by the garbage collector, with frequency determined in part by the size of the heap and in part by the rate at which new objects are created. This significantly impacts the overall performance of HONE, with a major culprit being the mappers, which generate a large number of objects to store the intermediate data. The techniques that we have discussed so far (e.g., using byte arrays to serialize objects, reducing the number of data streams, etc.) help in reducing the garbage collection overhead. As an additional optimization, we tried taking advantage of *off-heap memory*, via the `ByteBuffer` class in the Java NIO package. This

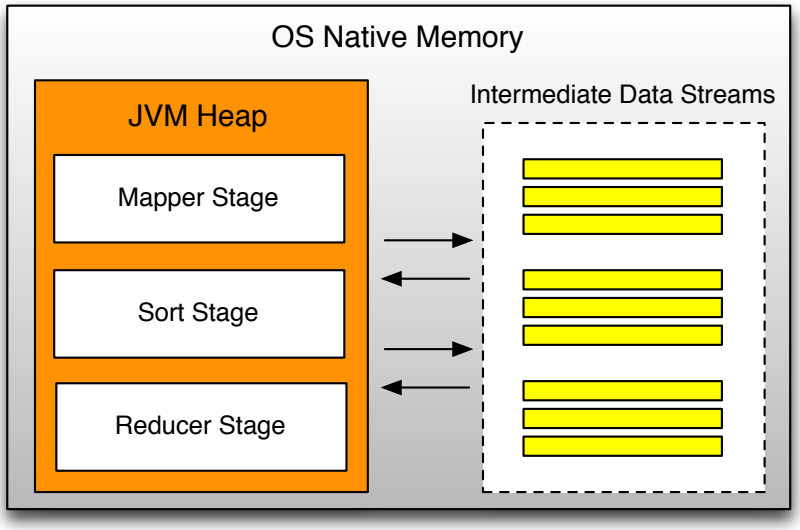


Figure 5.5: **Offloading intermediate data to *off-heap* direct native memory.**

allows us to manage the memory directly without interference, since the JVM garbage collector does not touch memory allocated in this fashion.

Figure 5.5 shows the memory heap managed by JVM encapsulated under OS native memory, where HONE operates. Memory needed for various MapReduce stages in HONE is still allocated through the JVM heap, but intermediate data created by the mappers are offloaded into data streams that are created off-heap. These off-heap data streams are then accessed by the sort stage, which performs in-place sorting, and then finally handed over to the reducers. Since this optimization uses non-standard APIs, we evaluated its impact in a separate set of experiments (see Section 5.6.2), but we do not use off-heap memory for most of our experiments.

NUMA support: To perform NUMA-specific optimizations, a system must support the ability to pin threads to specific cores. Unfortunately, Java does not provide explicit support for CPU affinity, as the assignment of threads to cores is handled opaquely by the JVM. However, thread-to-core affinity constructs can be supported via a library in C/C++ and accessed in Java via JNI. HONE currently does not take advantage of such optimizations.

Cache locality: High-performance algorithms usually require very careful tuning to take advantage of cache locality and processor prefetching—these effects can be substantial [11]. HONE, however, does not currently implement any optimizations along these lines, primarily because the intermediate JVM abstraction often obscures the machine instructions that are being executed, compared to a low-level language such as C where the programmer retains greater control over the system. Nevertheless, there may be opportunities to gain greater efficiencies via more cache-conscious designs. For example, combiners might help an application optimize for cache locality, e.g., by performing aggregation while intermediate data still reside in cache. However, this must be balanced against the overhead of context switching (from mapper to combiner execution). In the future, it would be interesting to explore whether such cache optimizations can be reliably implemented on the JVM.

5.4 Experimental Setup

5.4.1 Comparison Systems

HONE is open-source and can be downloaded at `hadoop1.org`. As previously discussed, it is implemented in Java for Hadoop API compatibility. There are a number of tunable parameters in HONE, which are summarized in Table 5.2. Unless otherwise specified, all experiments used default settings. We compared the performance of HONE against Hadoop PDM, a 16-node Hadoop cluster, a reimplementation of Phoenix in Java (described below), Phoenix system variants, and Spark. Details are provided below.

Parameter	**Description**
s	Split size. Default value is 64 MB.
r	Number of reducers.
tp_m	Mapper stage thread pool size. Default value is 15.
tp_r	Reducer stage thread pool size. Default value is 15.
tp_s	Sort stage thread pool size. Default value is 15.
arch	Approach to data shuffling: {*pull*, *push*, *hybrid*}. Default is *pull*.
lockType	Type of lock for *push* and *hybrid* data shuffling: Java synchronization, test-and-set (tas) spin lock, test and test and set (ttas) spin lock, and reentrant lock. Default is Java synchronization.
hs	Number of streams for each reducer for *hybrid* data shuffling.

Table 5.2: **Description of HONE parameters.**

Similar to Chapter 4, all single-machine experiments were performed on a server with dual Intel Xeon quad-core processors (E5620 2.4 GHz) and 128 GB RAM. This architecture has a 64KB L1 cache per core, a 256KB L2 cache per core, and a 12 MB L3 cache shared by all cores of a single processor. With hyperthreading, this machine can support up to 16 threads concurrently. The machine has six 2 TB 7200 RPM disks, each with 64 MB cache, arranged in a RAID6 configuration. For Hadoop PDM, we ran Hadoop YARN 2.0.3; the configuration parameters were set for the maximum allowable in-memory buffer sizes, but note that Hadoop buffer sizes are limited to 32-bit integers. For comparisons with Phoenix, we ran Phoenix2 and Phoenix++; for Spark, we used version 0.8.0. Our Hadoop cluster ran CDH4 (YARN) and comprises 16 compute nodes, each of which has two quad-core Xeon E5520 processors, 24 GB RAM, and three 2 TB disks. Note that with YARN, however, one node serves as the Application Master (AM), leaving only 15 actual worker nodes. The Hadoop cluster ran JDK6, whereas we used JDK7 everywhere else. Note that both the individual server and the Hadoop cluster were purchased around the same time; since they represent hardware from the same generation, our experiments fairly capture the capabilities of a high-end server and a modest Hadoop cluster.

Java implementation of Phoenix. The Phoenix project [74,82,92] shares

similar goals as HONE in terms of exploring MapReduce implementations for shared-memory systems. The biggest difference, however, is that Phoenix is implemented in C/C++ and thus abandons Hadoop API compatibility—this means that Hadoop applications need to be rewritten to take advantage of Phoenix.

Another substantial difference between Phoenix and HONE is how intermediate data are shuffled from the mappers to the reducers. Whereas HONE uses the pull, push, and hybrid approaches discussed in Section 5.3.1, Phoenix uses an array of hash tables: the array contains one hash table per mapper, and the entire data structure can be visualized as a 2D grid. Each hash table entry has an array of keys that hash to that location, and each key points to an array of associated values. Conceptually, we can think of each mapper as writing to a "column" in the 2D data grid. The reducers scan entries of the hash tables belonging to all the mappers to grab the appropriate intermediate (key, value) pairs; conceptually, we can think of this as reading the "rows" in the 2D data grid.

It did not appear to us that the Phoenix data-shuffling approach can be efficiently implemented in Java, but we nevertheless attempted an adaptation to help us better understand the differences between the languages and the unique challenges that Java imposes. Due to the overhead of materializing key and value objects in Java, a straightforward implementation was

utterly infeasible. Instead, we opted to minimize object overhead by replacing Phoenix's emission values arrays with byte arrays containing serialized data. The main data structure is a Java `HashMap` array, with one map per mapper, where the map keys are the intermediate keys and the map values are the intermediate values (associated with the key), both in serialized form. In order to support this data structure we also needed to create a second `HashMap` array, one map per mapper, so that reducers would know with which keys to query the main data structure. In each map, the map key is the reducer id and the map value is the reducer's keylist—once again, held in serialized form. Whenever a mapper emits an intermediate (key, value) pair, the system determines which reducer has ownership of that key and writes the key to that reducer's keylist. We feel that we have accurately captured the data-shuffling approach of Phoenix, and that our implementation represents a reasonable attempt to study how the "data grid" design would fare in Java.

5.4.2 Applications and Datasets

Our experiments explored a range of MapReduce applications, described below (see Table 5.3):

- Word Count (WC): This application counts the frequencies of all words in a collection of text documents.

Application	Dataset	Small	Medium	Large
Word Count (WC)	Wiki articles	128 MB, 256 MB, 512 MB	1 GB, 2 GB	4 GB, 8 GB, 16 GB
K-means (KM)	3D points	12M, 25M	51M, 102M	204M, 398M
Inverted Indexing (II)	Wiki articles	128 MB, 256 MB, 512 MB	1 GB, 2 GB	4 GB, 8 GB, 16 GB
PageRank (PR)	Wiki graph	0.4M, 0.8M	1.8M, 3.5M	7.2M
LDA	TREC docs	125 MB, 256 MB	512 MB	1 GB

Table 5.3: **Data conditions for each application. For k-means and PageRank, we show the number of points and number of graph nodes, respectively.**

- K-means (KM): This application implements k-means clustering using Lloyd's Algorithm. Since the algorithm is iterative, reducer output is passed to the input of the next mapper stage through the HONE namespace manager. These iterations proceed until convergence.
- Inverted Indexing (II): This application builds a simple inverted index, which comprises a mapping from terms to postings which hold information about documents that contain those terms. An inverted index is the core data structure used in keyword search.
- PageRank (PR): This application computes PageRank, the stationary distribution of a random walk over a graph. Like k-means clustering, this algorithm is iterative.
- Latent Dirichlet Allocation (LDA): This application builds topic models

over text documents using Latent Dirichlet Allocation [10] via variational inference. The implementation represents a substantial research effort [93] and demonstrates HONE on a non-trivial application. This algorithm is also iterative.

In terms of datasets, for word count and inverted indexing, we used articles from English Wikipedia totaling 16 GB. For k-means clustering, we randomly generated 398 million 3D coordinates; in all our experiments we ran clustering with $k = 10$. For PageRank, we used a Wikipedia article link graph that contains 7.2M nodes. For LDA, we used a document collection from the Text Retrieval Conference (TREC) that totals 1 GB [93].

One important variable in our experiments is the amount of data processed. To examine these effects, we generated subsets of varying sizes from the above datasets. We also divided the data conditions somewhat arbitrarily into small, medium, and large categories, summarized in Table 5.3.

In addition to the above real applications and datasets, we built a synthetic workload generator to better understand HONE performance under different workloads. Details of these experiments are discussed in Section 5.6.

5.5 Application Results

In this section, we present results of experiments that compared HONE against a variety of other systems on the applications described in Sec-

tion 5.4.2. To thoroughly characterize performance, we varied both the amount of compute resources available as well as the amount of data processed.

5.5.1 Strong Scalability Analysis

	HONE									
Threads	WC		KM		II		PR		LDA	
1	1892	-	768	-	2068	-	20	-	12214	-
2	946	100%	480	80%	1124	92%	14.0	73%	6296	97%
4	490	97%	281	68%	671	77%	10.8	56%	3160	97%
8	292	81%	185	52%	480	54%	9.5	27%	1675	91%
16	253	47%	166	29%	405	32%	7.8	17%	957	78%

Table 5.4: **Strong scalability experiments with Hadoop HONE: cells show running time in seconds and the strong scaling efficiency.**

	PDM									
Threads	WC		KM		II		PR		LDA	
1	4276	-	3275	-	5079	-	112	-	31991	-
2	2309	54%	2096	64%	3200	63%	92	82%	23673	74%
4	1498	54%	1676	40%	2102	47%	74	50%	12421	70%
8	1089	28%	1100	31%	1501	33%	69	27%	10024	44%
16	837	18%	849	19%	1383	18%	66	14%	8002	27%

Table 5.5: **Strong scalability experiments with Hadoop PDM: cells show running time in seconds and the strong scaling efficiency.**

In a strong scalability analysis, the problem size stays fixed but the number of processing elements varies. A program is considered to scale linearly if the speedup (in terms of work units completed per unit time) is equal to the number of processing elements used (N). Maintaining strong scalability

is challenging because coordination overhead increases with the number of processing elements.

If the amount of time to complete a work unit with one processing element is t_1 and the amount of time to complete the same unit of work with N processing elements is t_N, the strong scaling efficiency (SSE), as a percentage of linear, is given as follows:

$$\text{SSE} = \left(\frac{t_1}{N \cdot t_N}\right) \times 100\% \tag{5.1}$$

Table 5.4 and Table 5.5 shows the strong scalability results for HONE and Hadoop PDM on different applications. In all cases, the experiments ran on the server described in Section 5.4.1 and HONE used the pull-based data-shuffling approach. We increased the number of threads from 1 to 16 (by varying the thread pool sizes) while keeping the dataset size constant; the table shows running time in seconds and the strong scaling efficiency based on Equation (5.1). For iterative algorithms the reported values capture the running time of the first iteration. Speedup of HONE over Hadoop PDM is summarized in Table 5.6. These experiments used the large dataset condition for each application: for word count and inverted indexing, 8 GB; for k-means, 398M 3D points; for PageRank, the Wikipedia article link graph with 7.2M nodes; for LDA, 1 GB. For both HONE and Hadoop PDM, the number of

mappers was determined automatically based on the split size, which is 64 MB in our case. We used binary search to determine the optimal number of reducers, and figures for both HONE and Hadoop PDM are reported with optimal settings.

From these results we see that HONE is substantially faster than Hadoop PDM in terms of absolute running time. Keep in mind that our machine has only 8 physical cores, so the runs with 16 threads are taking advantage of hyperthreading. Overall, HONE exhibits much better strong scaling efficiency, outperforming Hadoop PDM in nearly all conditions. It is interesting to see that efficiency varies by application—for example, HONE achieves over 90% efficiency up to 8 threads on LDA, but for inverted indexing and PageRank, performance deteriorates substantially as we increase the thread count. From Table 5.6, we find no discernible trend on the relative performance of HONE compared to Hadoop PDM as the number of threads increases for the five applications.

5.5.2 Weak Scalability Analysis

In a weak scalability analysis, the problem size (i.e., workload) assigned to each processing element stays constant and additional processing elements are used to solve a larger overall problem. In this case, linear scaling is achieved if the running time stays constant while the workload is increased

in direct proportion to the number of processing elements.

If the amount of time to complete a work unit with one processing element is t_1, and the amount of time to complete N of the same work units with N processing elements is t_N, the weak scaling efficiency (WSE), as a percentage of linear, is given as follows:

$$\text{WSE} = \left(\frac{t_N}{t_1}\right) \times 100\% \tag{5.2}$$

Figure 5.6 shows running times for HONE with different numbers of threads and Table 5.7 provides the weak scalability efficiency computed from those results. In all cases, the experiments ran on the server described in Section 5.4.1. For these experiments, HONE used the pull-based data-shuffling approach and the number of reducers was tuned using binary search, as with the strong scalability analysis. For each application we increased the dataset size with the number of threads. For word count and inverted indexing, the dataset size varied from 128 MB to 2 GB; for k-means, from 12M to 204M points; for PageRank, from 0.4M nodes to 7.2M nodes; for LDA, from 125 MB to 1 GB. Note that for the LDA application we did not have sufficient data to run 16 threads, so that data point was omitted from this experiment. For iterative algorithms the reported values capture the running time of the first iteration.

Threads	WC	KM	II	PR	LDA
2	2.5×	4.5×	3×	6.5×	4×
4	3×	6×	3×	7×	4×
8	3.5×	6×	3×	7×	6×
16	3×	5×	3.5×	8.5×	8×

Table 5.6: **Speedup of HONE over Hadoop PDM based on the strong scalability results in Table 5.4 and Table 5.5.**

As with the strong scalability experiments, we see that efficiency varies by application. Again, since our machine has only 8 physical cores, it is no surprise that efficiency falls off dramatically with 16 threads, since they may be contending for physical resources. Inverted indexing and PageRank perform poorly, while the three remaining applications appear to scale better. In particular, both word count and LDA scale almost linearly up to the physical constraints of the hardware, and k-means scales well up to 4 threads. Note that this experiment used the pull-based approach to data shuffling for all data conditions, even though experiments below show that for larger datasets, a hybrid approach works better. This means that we can achieve even higher weak scaling efficiency if we dynamically adjust the data-shuffling approach.

5.5.3 Comparison of Hadoop Implementations

In the next set of experiments, we compared HONE with Hadoop PDM, our fully-distributed 16-node Hadoop cluster, and our Java implementation of Phoenix. Running times for the five sample applications on varying amounts

of data are shown in Figure 5.7. In the legend, "H-Pull", "H-Push" and "H-Hybrid" refer to the pull, push, and hybrid data-shuffling approaches in HONE (for the hybrid approach, each reducer works on two streams). "Hadoop PDM" refers to Hadoop pseudo-distributed mode. "Hadoop Cluster" refers to Hadoop running on the 16-node cluster. "Phoenix" refers to our implementation of Phoenix in Java. Note that HONE, Hadoop PDM, and our Java Phoenix implementation ran on the same machine; in all cases we fully utilized the machine with 16 threads. For iterative algorithms the reported values capture the running time of the first iteration. As before, we report results with the optimal reducer setting discovered via binary search.

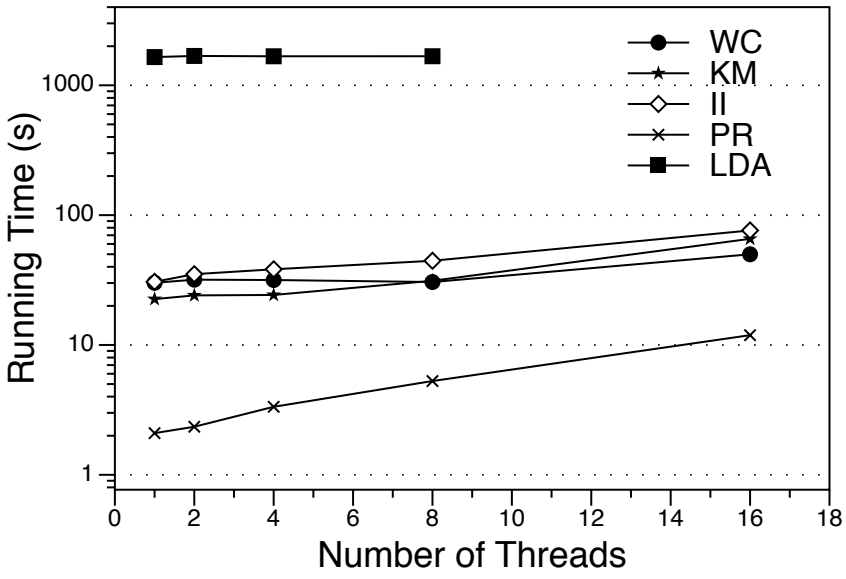


Figure 5.6: **Results of weak scalability experiments with HONE.**

We summarized the speedup comparing HONE to Hadoop PDM and the

Threads	WC	KM	II	PR	LDA
2	95%	94%	88%	90%	98%
4	95%	94%	76%	63%	99%
8	99%	61%	66%	40%	99%
16	61%	30%	41%	18%	-

Table 5.7: **Weak scaling efficiency based on Figure 5.6.**

Hadoop cluster as follows: For each data size category (small, medium, and large), we considered only the largest data condition for each application, as shown in Table 5.3. For example, with word count we considered the 512 MB, 2 GB, and 16 GB conditions. For each data condition, we selected the fastest from the {pull, push, hybrid} approaches and divided that running time by the running time of either Hadoop PDM or the Hadoop cluster. If the value is greater than one, indicating that HONE is slower, we take the negative; otherwise, we take the inverse, indicating that HONE is faster. These values are reported in Table 5.8.

For word count, as we can see from Figure 5.7a, the pull-based approach to data shuffling is the fastest for small to medium datasets. With large datasets, however, the hybrid approach appears to be the fastest. In all cases, the push-based approach is the slowest of the HONE configurations. HONE performs substantially faster than Hadoop PDM across all data conditions and is faster than the 16-node Hadoop cluster on small and medium datasets. The latter finding is not surprising since Hadoop jobs on a distributed cluster have substantial startup costs relative to the amount of data processed. We

find that our Java Phoenix implementation is very slow, sometimes by up to two orders of magnitude compared to HONE.

Results for k-means clustering and inverted indexing follow the same general trends as word count: pull-based data shuffling is faster on smaller datasets, but the hybrid approach is faster on bigger datasets. HONE is faster than Hadoop PDM across all dataset sizes, and it is faster than the 16-node Hadoop cluster on small to medium datasets. However, the Hadoop cluster is faster on the large datasets. For inverted indexing, the Java implementation of Phoenix is terrible, just like in word count. However, for k-means, the performance gap between Java Phoenix and HONE is substantially smaller—in some cases, the performance of Java Phoenix approaches HONE configurations. We believe that this is because k-means generates far less intermediate data than inverted indexing or word count, which confirms our thinking all along—that the optimization of intermediate data structures for data shuffling is the key to achieving high performance.

We notice a different pattern for PageRank and LDA: HONE is faster than both Hadoop PDM and the Hadoop cluster across all data conditions. This is perhaps due to the relatively small size of the datasets—the graph is relatively small, and the document collection for LDA is smaller than the Wikipedia articles used in word count and inverted indexing. Thus, these results appear to be consistent with the above findings. The pull-based

HONE vs. Hadoop PDM

	WC	KM	II	PR	LDA
small	6×	14×	4×	30×	7×
medium	4×	3×	5×	9×	8×
large	5×	3×	4×	6×	8×

HONE vs. Hadoop cluster

	WC	KM	II	PR	LDA
small	6×	7×	4×	15×	3×
medium	2×	2×	2×	5×	2×
large	−2×	−2×	−2×	3×	2×

Table 5.8: **Relative performance of HONE compared to Hadoop PDM (top) and the 16-node Hadoop cluster (bottom) for different data conditions. Negative values indicate that HONE is slower than the comparison system.**

approach outperforms all others for PageRank, but all three data-shuffling approaches are roughly equal for LDA.

Summarizing these results, our experiments suggest a few key takeaways. For small to medium datasets, the pull-based approach to data shuffling seems to be the fastest, but for large datasets, the hybrid approach can be faster—however, there are application-specific differences, such as with LDA. In our applications, we did not find a case where the push-based approach was convincingly better, which suggests that contention on the reducer streams and synchronization overhead significantly impacts performance.

In all these experiments, HONE is faster than Hadoop PDM, and in some cases, faster than the 16-node Hadoop cluster as well. For the cluster results, one can criticize that we have not used sufficiently large datasets to make distributed Hadoop worthwhile—but that's exactly the point we are trying

to make. As individual servers grow in memory capacity and core count, the sizes of datasets that can be handled in memory grows as well, and in these cases, a scale-up solution is perhaps preferable to a scale-out solution.

Finally, comparisons to our Java Phoenix implementation show that intermediate data structures for data shuffling need to be specifically designed for the execution environment. We adapted an approach that works well for C/C++, but translates into an inefficient design in Java. This shows, not surprisingly, that optimizations need to be targeted to the specific execution environment.

5.5.4 Comparison with Other Systems

Although Phoenix and Spark have very different designs compared to HONE, we believe that a comparison is still instructive. Here, we describe experiments on the word count and k-means clustering applications with varying amounts of data.

Phoenix2 [92] and Phoenix++ [82] are implemented in C/C++. Thus, a comparison against HONE gives us a sense of how much the choice of language matters. We downloaded both systems and ran an "out of the box" evaluation with default settings on word count and k-means clustering (both were existing implementations). We used the same server as all our other single-machine experiments. Relative performance is shown in Table 5.9 for

word count (top) and k-means (bottom): positive values indicate that HONE is faster and negative values indicate that the comparison system is faster.

The earlier system, Phoenix2, is actually slower than HONE on word count, and has scalability limitations—throwing segmentation faults beyond 2 GB.

Phoenix++, on the other hand, is 2–3× faster than HONE on word count up to 8 GB, but has roughly comparable performance at 16 GB. For k-means clustering, both Phoenix2 and Phoenix++ are substantially faster than HONE, by a factor of up to 7× on larger data. The symbol $\sim$ indicates that speed is roughly the same as HONE. Note that Phoenix++ performs "in-mapper combining" [58] and requires the developer to specify a data structure to store intermediate data (based on the application type), which gives it a performance advantage over Phoenix2 and HONE. However, this substantially alters the MapReduce model. These results show that there can be significant performance advantages to adopting C/C++, but at the cost of abandoning Hadoop compatibility.

In our evaluation of Spark, we loaded all data into memory (via the RDD abstraction) and allowed the system to fully utilize hardware resources (16 threads). These evaluations were performed on the same server as all the other single-machine experiments, and we used existing word count and k-means implementations. Results are shown in Table 5.9 for word count

Word Count

	128 MB	256 MB	512 MB	1 GB	2 GB	4 GB	8 GB	16 GB
Phoenix2	2×	2×	3×	2.4×	2.1×	-	-	-
Phoenix++	−2×	−3×	−2×	−2×	−2×	−3×	−2×	∼
Spark	3×	3×	3×	2×	2×	2×	∼	∼

K-means

	12M	26M	51M	102M	204M	398M
Phoenix 2	−4.5×	−3.5×	−4×	−4.5×	−6.5×	−6.4×
Phoenix++	−5×	−4×	−5×	−6×	−7.5×	−7×
Spark	4×	3×	3×	2×	2×	3×

Table 5.9: **Relative performance of HONE compared to Phoenix2, Phoenix++, and Spark for word count (top) and k-means (bottom). Negative values indicate that HONE is slower than the comparisons system. Note that Phoenix2 does not scale beyond 2 GB and terminates with a segmentation fault. The symbol ∼ indicates that speed is roughly the same as HONE.**

(top) and k-means (bottom). For word count, HONE is faster than Spark on datasets up to 4 GB and roughly comparable in terms of speed for larger datasets. For k-means, HONE is consistently faster than Spark for all dataset sizes. We are quick to emphasize that this is inherently an apples-to-oranges comparison because HONE and Spark are very different. Whereas Spark provides a general data processing model, HONE is limited to MapReduce. Spark was designed for distributed execution on a cluster, whereas HONE was specifically optimized for running on a single machine. However, since Spark and HONE both run on the JVM, the performance differences gives us a sense of the gains that might be attributed to careful engineering against the characteristics of the platform.

5.5.5 Effects of Input Split Size

In standard distributed Hadoop, input splits for the mappers are aligned with HDFS blocks so that tasks can be efficiently placed on machines where the blocks are held locally. Because HONE runs on a single machine, this constraint is not applicable and thus we have greater flexibility in tuning the split size. There are two considerations to balance when setting a value: Smaller splits lead to more mapper tasks and thus generate more opportunities to extract parallelism. On the other hand, if the split size is too small, the mapper threads don't have sufficient work to perform, and thus we waste time context switching.

We performed an experiment to empirically determine how these two factors play out. Figure 5.8 plots the execution time of word count and inverted indexing for different split sizes on the 512 MB dataset (using 16 threads). As we can see, for word count the optimal split size is 5 MB, whereas for inverted indexing, the we achieve the fastest running time with 64 MB.

5.5.6 Using HONE in a Multi-tenancy Cloud

HONE provides an interesting option in addition to other MapReduce execution frameworks (such as Hadoop PDM and Hadoop cluster) to a cloud

service provider for efficient execution of a set of MapReduce jobs. For our illustrative experiment, we consider three workload mixes having ten word count application jobs each; *mix*-1 consists of 60% small (125 MB input size), 20% medium (1 GB) and 20% large (15 GB) jobs; *mix*-2 consists of 20% small, 60% medium and 20% large jobs; *mix*-3 consists of 20% small, 20% medium and 60% large jobs. We consider two execution frameworks: HONE and a Hadoop 15-node cluster, and we compare the performance of naive execution frameworks, All-HONE and All-Hadoop, with smarter counterpart S-Scheme. All-HONE scheme chooses HONE as an execution framework for all the jobs, whereas All-Hadoop chooses 15-node Hadoop cluster for all jobs. S-Scheme uses HONE for small and medium data sizes, and the 15-node Hadoop cluster for the larger jobs. Table 5.10 shows that smarter execution selection schemes such as S-Scheme performs significantly better than the naive schemes. For workload *mix*-1,2,3 when compared to All-HONE, S-Scheme exhibits 3.5X, 3X and 4X speedup and when compared to All-Hadoop, it exhibits speedup of 2.3X, 2X and 1.3X.

Workload Mix	All-HONE	All-Hadoop	S-Scheme
mix-1	1249	859	367
mix-2	1345	951	463
mix-3	3546	1144	899

Table 5.10: **Selecting the right framework for the data size can lead to significant benefits. All numbers are in seconds.**

5.6 Synthetic Workload Results

To better understand the effects of different job characteristics on HONE, we built a synthetic workload generator that allows us to create different types of MapReduce jobs. Whereas our sample applications encode a specific set of fixed characteristics, the workload generator lets us vary those characteristics independently. We also used this tool to evaluate the impact of the off-heap memory allocation optimization.

5.6.1 Workload Generator

The basic structure of the synthetic job is similar to the word count application, but with a number of adjustable parameters, shown in Table 5.11. The job takes as input a collection of text documents to simulate actual data processing; the mapper tokenizes each document and processes each token in turn. What happens next depends on the workload parameters (details below), but intermediate (key, value) pairs are generated with an integer between 1 and 10,000 as the key and a random string as the value. The reducers simply count the number of values that are associated with each key and output the final counts.

Mapper emit distribution parameter (d_m) determines the number of intermediate (key, value) pairs emitted *per token* in the mapper. This param-

Parameter	Description
d_m	Mapper emit distribution; possible values are {*one-to-one*, *many-to-one*, *one-to-many*}.
λ	For d_m = *one-to-many*, the number of intermediate (key, value) pairs to emit per token is decided by drawing from a Poisson distribution with mean λ. Default value is 10.
ψ	For d_m = *many-to-one*, the probability to emit an intermediate (key, value) pair per token. Default value is 0.1.
d_i	Intermediate key distribution; possible values are {*uniform*, *Zipfian*, *biased*}.
$zipf_{skew}$	For d_i = *Zipfian*, the Zipfian skew parameter. Default value is 10.
α	For d_i = *biased*, probability an intermediate (key, value) pair will be sent to a single "special" reducer. Default value is 0.7.
ps	Payload size, the length of the randomly-generated string that serves as the intermediate value. Default value is 3.
b_{cpu}	Parameter to control CPU-intensiveness of the workload (in the mapper). For each token, value of π is calculated to b_{cpu} digits before generating intermediate data. Default value is 2.

Table 5.11: **Description of workload generator parameters.**

eter attempts to model the fact that some MapReduce algorithms generate more intermediate data than input data, while others generate less. The possible settings are {*one-to-one*, *many-to-one*, *one-to-many*}.

Implementing the *one-to-one* setting is straightforward: for each input token the job emits an intermediate (key, value) pair, based on constraints specified below. For the *one-to-many* setting, we need a mechanism to stochastically determine the number of (key, value) pairs to emit for each token. For this we draw from a Poisson distribution with a mean of λ (set

to 10 by default). In the *many-to-one* case, for every token, we generate an intermediate (key, value) pair with probability ψ. The default value of ψ is 0.1, which means that one intermediate pair is emitted every ten tokens on average.

Intermediate key distribution parameter (d_i) determines how intermediate (key, value) pairs are assigned to reducers. For example, word count exhibits a Zipfian intermediate key distribution since term occurrences are (roughly) Zipfian (i.e., lots of occurrences of common words and a long tail). Graph algorithms often behave similarly due to the presence of "supernodes", or nodes with many incoming edges. On the other hand, intermediate data in k-means clustering is uniformly distributed.

In order to capture these different characteristics, we provide a parameter d_i that can be set to {*uniform*, *Zipfian*, *biased*}. In all cases the intermediate key is an integer between 1 and 10,000, but the selection of the key is based on this setting:

- In the case of *uniform*, the integer is selected based on a uniform distribution.
- In the case of *Zipfian*, the key is drawn from a Zipfian distribution with skew parameter $zipf_{skew}$, with a default value of 10.
- In the case of *biased*, the intermediate key is selected such that each

emitted (key, value) pair is assigned to a "special" reducer with probability α, or is otherwise assigned to one of the other reducers with uniform probability. This means that if we have r reducers, one reducer will receive α fraction of all intermediate data, while the remaining data will be distributed evenly among the $r - 1$ other reducers. The default value of α is 0.7.

Payload Size (***ps***) determines the size of the intermediate value. In word count the payload is always an integer, but other MapReduce applications may emit bigger values that have complex internal structure. In the workload simulator, the value in the intermediate (key, value) pair is a randomly-generated string of length ps, with three as the default value.

CPU-intensiveness parameter (b_{cpu}) controls the amount of processing that is performed in the mapper in the simulated workload. To simulate workloads that are CPU-intensive to varying degrees, we compute π to b_{cpu} digits for each token before proceeding to generate intermediate output. The default value is two.

5.6.2 Summary of Findings

We have been exploring the performance of HONE and other systems under different workloads using the synthetic workload generator. This is the subject of ongoing explorations, but in Figure 5.9 we share a few of our initial

findings. In particular, we have been using this approach to examine the performance impact of the off-heap memory optimization discussed in Section 5.3.2; the is abbreviated "OH" in the figure legends. The basic setup is the same as in all the experiments above, comparing HONE (using the pull-based approach to data shuffling) with Hadoop PDM and the 16-node Hadoop cluster. As with before, all in cases we used binary search to find the optimal number of reducers, and report results based on those settings.

HONE gracefully handles skew: Figure 5.9a shows that if the intermediate key distribution is uniform, then HONE is faster than Hadoop PDM for all data conditions examined and HONE is faster than the full Hadoop cluster for smaller datasets. This finding is consistent with the results from Section 5.5. However, for non-uniform intermediate key distributions, HONE appears to be faster than Hadoop PDM and the full Hadoop cluster for the data conditions we examined; this is shown in Figures 5.9b, 5.9c, 5.9d, and 5.9e. Skew creates stragglers (tasks that take substantially more time than the others), which is a well-known issue for Hadoop in a distributed environment [52, 57], and the effects appear to carry over to Hadoop PDM as well. On the other hand, the design of HONE makes it more resistant to skew effects.

HONE is less sensitive to payload size: Increasing the payload size in-

creases disk activity and increases pressure on the communication channels for Hadoop PDM and the Hadoop cluster. On the other hand, HONE appears to be relatively insensitive to the payload size since everything is held in memory (provided, of course, that we have sufficient memory). This result is shown in Figure 5.9f.

HONE effectively utilizes CPU resources: As the workload becomes more CPU-intensive, HONE is able to effectively utilize available cores; see Figures 5.9g and 5.9h. Our single server has only 8 physical cores, so at some point we begin to saturate all available compute capacity—the full Hadoop cluster obviously has an advantage here because it has more cores. In Figure 5.9h, we make an interesting observation: for Hadoop PDM and the Hadoop cluster, increasing the CPU-intensiveness parameter (at least up to 10) does not have much of an impact on the overall running time, which suggests that there are bottlenecks elsewhere (e.g., I/O and skew issues). In this sense, HONE achieves a better balanced design.

HONE off-heap can be up to 2× faster than HONE on-heap: With HONE, offloading intermediate data to *off-heap* native direct memory consistently improves performance, compared to the default setting where all intermediate data are stored on the JVM heap.

5.7 Summary of Contributions

In this thesis, we propose to "scale down" Hadoop to run on shared-memory machines. We present a prototype runtime called HONE that is intended to be API with standard (distributed) Hadoop. That is, one can take an existing Hadoop jar and run it, without modification, on a multi-core shared memory machine using HONE. This allows us to take an implemented algorithm and find the most suitable runtime environment for execution on datasets of varying sizes—if the data fits into RAM, we can avoid network latency and significantly increase execution time in a shared-memory environment.

API and binary compatibility with Hadoop is the central tenant in our design. Although there have previously been alternative MapReduce implementations for shared-memory machines , taking advantage of them would require porting Hadoop code over to another custom API. In contrast, HONE is able to leverage existing implementations—we present experiments running a Hadoop-based Latent Dirichlet Allocation (LDA) implementation [93] on HONE and compare the performance of a single shared-memory machine with a 15-node Hadoop cluster.

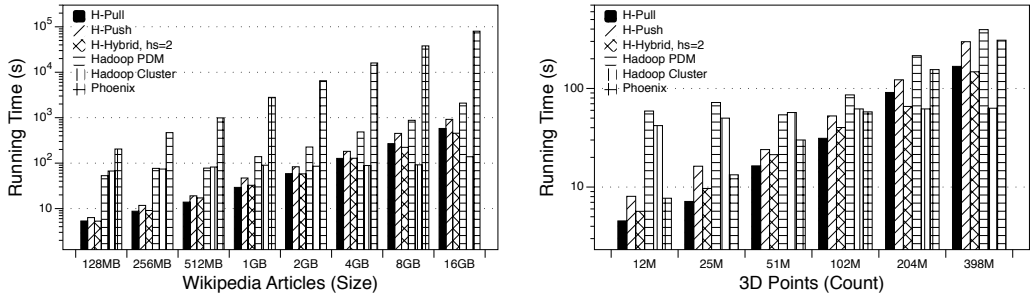
As others have suggested, we need to re-think scale-out vs. scale-up architectures as the amount of cores and memory on high-end commodity servers

continues to increase. There is no doubt that the *total* amount of data is also growing rapidly, but it is unclear if the datasets used in *typical* analytical tasks today are increasing as fast. The crux of the scale-out vs. scale-up debate hinges on these relative rates of growth: server capacities are (roughly) growing with Moore's Law, which should continue for at least another decade. If dataset sizes are growing at a slower rate, then scale-up architectures will become increasingly attractive.

Ultimately, the datacenter is likely to consist of a mix of scale-out and scale-up systems—we will continue to run large, primarily disk-based jobs to scan petabytes of raw log data to extract interesting features, but this work explores the interesting possibility of switching over to a multi-core, shared-memory system for efficient execution on more refined datasets. With HONE, this can all be accomplished without leaving the comforts of MapReduce: we simply select the most appropriate execution environment based on dataset size and other characteristics of the workload. This brings us to the biggest limitation of our current work and the subject of ongoing research—how to *a priori* determine the best HONE configuration in terms of the data-shuffling approach, split size, thread pool sizes, etc. In the future, we can imagine an optimizer that is able to examine a Hadoop workload and *automatically* decide what job to run where and the optimal parameter settings.

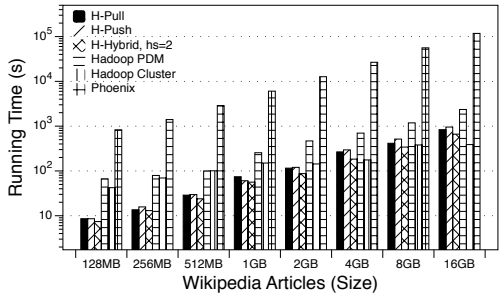
Our contributions in this work are:

- HONE is a scalable MapReduce implementation for multi-core, shared-memory machines. To our knowledge it is the first MapReduce implementation that is both Hadoop API compatible and optimized for scale-up architectures.
- We propose and evaluate different approaches to implementing the data shuffling stage in MapReduce, which is critical to high performance.
- We discuss key challenges in implementing HONE on the JVM, how we addressed them, and lessons we learned along the way.
- We evaluate HONE on a number of real-world applications, comparing it to Hadoop pseudo-distributed mode, a 16-node Hadoop cluster, and a few other systems.
- We share a synthetic workload generator for evaluating HONE that may be of independent interest for evaluating other systems.

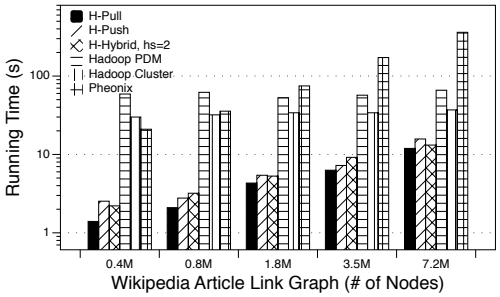


(a) Word Count

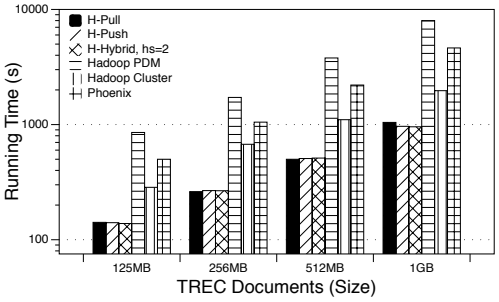
(b) K-means



(c) Inverted Indexing



(d) PageRank



(e) LDA

Figure 5.7: **Comparing HONE, Hadoop PDM, the 16-node Hadoop cluster, and our Java Phoenix implementation on five different applications with varying amounts of data. Note that the y-axis is plotted on a log scale.**

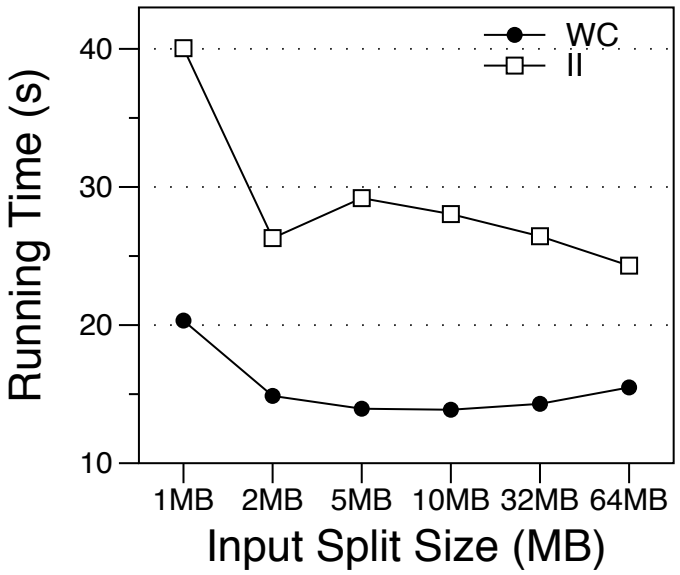


Figure 5.8: **Running time of word count and inverted indexing on the 512 MB dataset with different split sizes (running 16 threads).**

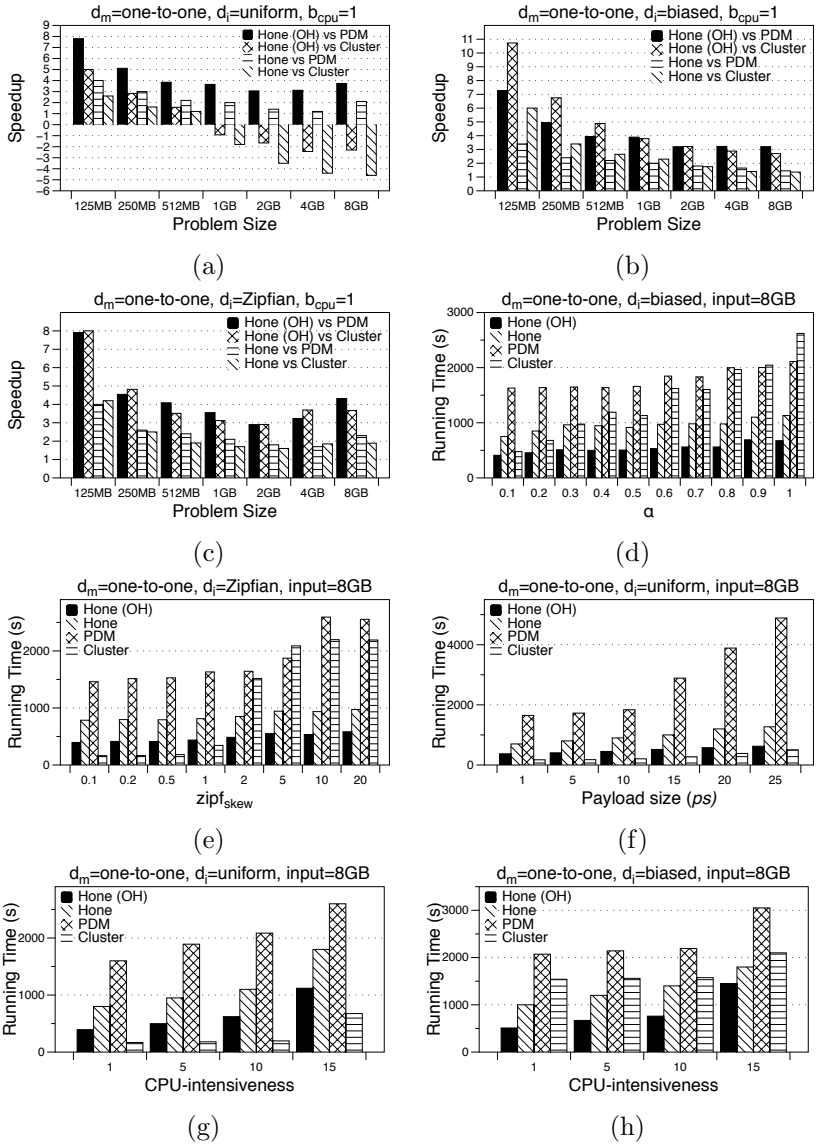


Figure 5.9: **Experimental results using our synthetic workload generator.**

Chapter 6

Conclusion and Future Directions

Overall, in this dissertation, we focus on the issue of resource inefficiency in scale-out architectures and we demonstrate that in these architectures workload consolidation can minimize overall resource consumption.

In the first part of the dissertation, we exploit the fact that most distributed environments need to use replication for fault tolerance, and we devise workload-driven replica selection and placement algorithms that attempt to minimize the *average query span*. We model a historical query workload trace as a *hypergraph* over a set of data items (which could be relation partitions, or file chunks), and formulate and analyze the problem of replica placement by drawing connections to several well-studied graph the-

oretic concepts. We use these connections to develop a series of algorithms to decide which data items to replicate, and where to place the replicas. We show effectiveness of our proposed approach by building a trace-driven simulation framework and by presenting results on a collection of synthetic and real workloads. Our experiments on analytical workloads show that careful data placement and replication can dramatically reduce the average query spans resulting in significant reductions in the resource consumption.

In the second part of the dissertation, we identify two key challenges in today's search architectures, first is high search costs that increase overall operating cost for the search service provider; second is, high load imbalance that degrades the search service performance. To address these challenges, we develop a system called as WAFEL, which stands for "Workload-Aware Framework for search cost and load Effective information retrievaL". These two objectives, minimizing search cost and load imbalance are often at odds with each other. To achieve first objective (minimizing search cost) we analyze search workload history to perform workload-aware web document partitioning to minimize the overall search cost by minimizing the average query span. Next, based on access frequencies of the web documents, we perform load-aware replication to create opportunity for load dispersion across the partitions at the time of routing. In order to take the advantage of workload-aware partitioning and load-aware replication, we introduce a smart replica-

tion scheme that can route queries to the physical partitions minimizing load imbalance while carefully trading the search cost. Our experiments on real dataset show effectiveness of our proposed approaches.

With an observation that unnecessary scaling-out can be costly in terms of resource consumption often degrading performance as well, in the context of Hadoop, we propose scaling down Hadoop to run on multi-core, shared-memory machines. We present a prototype runtime called HONE (Hadoop One) that is API compatible with Hadoop. With HONE, we can take an existing Hadoop application and run it efficiently on a single server. This allows us to take existing MapReduce algorithms and find the most suitable runtime environment for execution on datasets of varying sizes. For dataset sizes that fit into memory on a single machine, our experiments show that HONE is substantially faster than Hadoop running in pseudo-distributed mode. In some cases, HONE running on a single machine outperforms a 16-node Hadoop cluster.

Despite the overall improvements that the techniques introduced in this work bring about, there are number of limitations that need to be addressed. We leave these issues to be addressed as future work. These limitations are as follows:

- **Workload-aware data partitioning:** Our workload-aware partition-

ing approach to minimize query spans discussed in Chapter 3 and Chapter 4 assumes homogeneous partitions or physical machines. A typical data center may consist of heterogeneous machines. We note that partitioning on heterogeneous setup to achieve certain objective is a extremely hard problem and at this point it is unclear how to approach this problem.

- **Routing:** In this work, once we perform workload-aware data placement and replication, we perform setcover computation for routing queries to physical partitions to minimize the number of machines accessed by the queries. Performing setcover computation for each query can be expensive, instead it would be faster if we can employ an incremental setcover computation to speedup the routing. We leave this as future work.
- **In-memory MapReduce:** Our in-memory MapReduce system HONE is developed in Java, that means it inherits limitations of JVM. Single JVM process does not scale for large datasets in-memory. In this work, although we implement off-heap technique to scale the system for relatively bigger data, we cannot handle more than 16GB in-memory. As a future work, we wish to scale our system to much larger datasets on single machine.

This dissertation has highlighted the importance of resource minimization in Big Data platforms. The techniques introduced in this work focus on taking advantage of workload-awareness and consolidation of workloads to control resource wastage and also to increase performance.

Also, in this chapter, we discuss a few interesting broad set of future research directions related to the work presented in this dissertation.

6.1 Energy Efficient Computing

Today we are witnessing that energy costs are ever increasing. On the other hand, increasing demand for information processing have led to cheaper, faster and larger data management systems. This demand in turn requires more and more hardware to be employed to meet the service needs putting more pressure on energy costs. We notice that today most of the systems that are being built for the purpose of information processing usually are optimized for execution times. For example: Hadoop is a very popular data processing software employed primarily to crunch very large amounts of data (in the range of terabytes and petabytes). Hadoop cluster farms usually make use of thousands of cheap commodity server hardware. Main emphasis in development of MapReduce based software like Hadoop is high scalability and lower execution times. We find that emphasis on energy efficiency is completely missing in these systems. We feel that today there is a strong

need for energy and resource-aware scalable systems that only focus on high scalability and low execution times but also on lower energy and resource consumptions.

One open problem that we identify in this area is regarding improving energy efficiency of Hadoop. We identify several opportunities in Hadoop to improve its energy efficiency while maintaining its impressive scalability. First is in HDFS, where question is how to improve HDFS data placement policy so that data items can be co-located in an automated way? Secondly, imagine a hadoop cluster farm that has both cheap commodity and high performance expensive servers, then how to route the Hadoop jobs so that we achieve highest energy efficiency while we meet our performance deadlines? In both the issues, strong knowledge about workload history would help immensely. Our work in this thesis on workload-aware data placement and replication policies is relevant to these issues and lessons learnt from our work can be applied in a generic way to solve these problems.

6.2 In-Memory Computing

Previously, it used to be expensive to scale up computing hardware, so most of the large-scale systems used cheap commodity machines. But today scenario is changing where scaling-up of a single machine has become fairly inexpensive. In coming days, the data centers ultimately will consist of a mix

of scale-out and scale-up systems. We note that, today most of the systems developed to handle the problem of Big Data are optimized for scale-out settings. These design decisions often can lead to under utilization of the available resources. In order to judiciously make use of cheap processing power of commodity hardware together with large memories of scaled-up machines, we need systems that can also take advantage of large available memories.

One idea in this direction is to develop smart data processing techniques that can take advantage of both scale-out and scale-up optimized systems. For example in the scenario of Hadoop, let us say that we have developed a Hadoop compatible system like HONE that is optimized for scaled-up systems. Also let us say we perform a workload-aware data placement and replication in Hadoop. When jobs arrive at Hadoop then if job span is equal to 1 then we can send to scale-up optimized system like HONE and when a job with job span greater than 1 arrives at the system, it can be sent to vanilla Hadoop which uses scaled-our architecture with cheap commodity machines.

6.3 NUMA-aware Computing

Nowadays, as the number of cores to be placed on a single processor socket has hit the limits, multi-socket computing has become a norm where

multiple sockets with each socket employing multiple cores will carry out the processing. Multiple sockets share a common cache whereas multiple CPU core within a socket share a high speed cache. Accessing the data across the socket is orders of magnitude slower than the data access across the cores within a same socket, hence the problem of Non-Uniform Memory Access (NUMA).

As the many core computing becomes more and more popular, information processing systems need to utilize many cores for performance. NUMA will create a major performance hit for these systems. So we need NUMA-aware systems, that can make smart decisions about data access and scheduling and avoid high latency out-of-socket data accesses.

One open problem in this direction that we identify is in the context of multi-threaded map reduce systems. If we have multi-threaded map reduce system like HONE, then how can we achieve NUMA-awareness so that in the shuffling stage threads do not make too many out-of-socket accesses?

6.4 Multi-tenancy in Cloud Infrastructures

Cloud computing is gaining popularity where in order to reduce operating costs, multiple businesses or tenants share a common infrastructure owned by a cloud service provider. In coming days, to improve businesses multiple tenants may also share data among themselves while sharing common

infrastructure.

In a cloud setting, usually, each tenant has different workload requirements and service level agreements (SLA). Each tenant wants to make profits and also expects cloud service provider to meet his SLAs. Whereas cloud service provider's goal is to maximize performance for each tenant and minimize the overall cost of their service infrastructure. In other words, cloud service providers objective can be given as: $\mathbf{max}(\frac{performance}{cost})$. In order to meet both tenant and cloud provider's objective, we need techniques to perform multi-tenant workload consolidation.

One open problem in this direction is: given different workloads corresponding to multiple tenants with different SLAs, can we model these workloads together such that the workload-aware data partitioning helps meet multi-tenant SLAs as well as help maximize cloud service provider's objective.

Bibliography

[1] hMETIS: A hypergraph partitioning package, http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview.

[2] MLPart, http://vlsicad.ucsd.edu/gsrc/bookshelf/slots/partitioning/mlpart/.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, August 2009.

[4] N. Alon, P. D. Seymour, and R. Thomas. A separator theorem for graphs with an excluded minor and its applications. In *STOC*, 1990.

[5] C. J. Alpert. The ISPD98 circuit benchmark suite. In *Proc. of Intl. Symposium on Physical Design*, 1998.

[6] R. Aly, D. Hiemstra, and T. Demeester. Taily: Shard selection using the tail of score distributions. In *SIGIR*, pages 673–682, 2013.

[7] H. Amur, J. Cipar, V. Gupta, G. Ganger, M. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *SoCC*, 2010.

[8] J. Arguello, F. Diaz, J. Callan, and J.-F. Crespo. Sources of evidence for vertical selection. In *SIGIR*, pages 315–322, 2009.

[9] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. In *SWAT*, 1996.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 2003.

[11] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *CACM*, 51(12):77–85, 2008.

[12] R. B. Boppana. Eigenvalues and graph bisection: An average-case analysis. In *FOCS*, 1987.

[13] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.

[14] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Design and implementation of move-based heuristics for VLSI hypergraph partitioning. *J. Exp. Algorithmics*, 5:5, 2000.

[15] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *SIGIR*, pages 21–28, 1995.

[16] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 10(7):673–693, 1999.

[17] U. V. Catalyurek and C. Aykanat. Patoh: a multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533, 1999.

[18] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *PACT*, 2010.

[19] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, pages 98–109, 2011.

[20] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Supercomputing*, 2002.

[21] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[22] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[24] K. Devine, E. Boman, L. Riesen, U. Catalyurek, and C. Chevalier. Getting started with zoltan: A short tutorial. In *Proc. of 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*, 2009. Also available as Sandia National Labs Tech Report SAND2009-0578C.

[25] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[26] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3:515–529, September 2010.

[27] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, and Y. Li. A novel approach to improving the efficiency of storing and accessing small files on Hadoop: a case study by PowerPoint files. In *SCC*, 2010.

[28] Z. Du, J. Hu, Y. Chen, Z. Cheng, and X. Wang. Optimized qos-aware replica placement heuristics and applications in astronomy data grid. *Journal of Systems and Software*, 84(7):1224 – 1232, 2011.

[29] D. Economou, S. Rivoire, and C. Kozyrakis. Full-system power analysis and modeling for server environments. In *In Workshop on Modeling Benchmarking and Simulation (MOBS)*, 2006.

[30] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[31] U. Feige, G. Kortsarz, and D. Peleg. The dense k-subgraph problem. *Algorithmica*, 1999.

[32] H. Ferhatosmanoglu, A. S. Tosun, and A. Ramachandran. Replicated declustering of spatial data. In *PODS*, 2004.

[33] M. Garey and D. Johnson. *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*. 1979.

[34] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *ICDCS*, 2004.

[35] G. Graefe. Database servers tailored to improve energy efficiency. In *Proceedings of EDBT workshop on Software engineering for tailor-made data management*, 2008.

[36] S. Harizopoulos, M. A. Shah, J. Meza, and P. Ranganathan. Energy efficiency: The new holy grail of data management systems research. In *CIDR*, 2009.

[37] L.-Y. Ho, J.-J. Wu, and P. Liu. Optimal algorithms for cross-rack communication optimization in mapreduce framework. In *IEEE International Conference on Cloud Computing*, 2011.

[38] U. Holzle. Powering a google search. `http://googleblog.blogspot.com/2009/01/powering-google-search.html`. Created January 11, 2009.

[39] W. Jiang, V. T. Ravi, and G. Agrawal. A map-reduce system with an alternate api for multi-core environments. In *CCGRID*, 2010.

[40] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *IEEE VLSI*, pages 69–529, 1999.

[41] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):69–79, 1999.

[42] G. Karypis and V. Kumar. hmetis: A hypergraph partitioning package, version 1.5. 3. *user manual*, 23, 1998.

[43] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proc. of DAC*, pages 343–348, 1998.

[44] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[45] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *SIGMOD*, 2012.

[46] M. Koyutürk and C. Aykanat. Iterative-improvement-based declustering heuristics for multi-disk databases. *Information Systems*, 2005.

[47] A. Kulkarni and J. Callan. Document allocation policies for selective searching of distributed indexes. In *CIKM*, pages 449–458, 2010.

[48] A. Kulkarni and J. Callan. Document allocation policies for selective searching of distributed indexes. In *CIKM*, pages 449–458, 2010.

[49] K. Kumar, A. Quamar, A. Deshpande, and S. Khuller. Sword: workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal*, pages 1–26, 2014.

[50] K. A. Kumar, A. Deshpande, and S. Khuller. Data placement and replica selection for improving co-location in distributed environments. *CoRR*, abs/1302.4168, 2013.

[51] K. A. Kumar, J. Gluck, A. Deshpande, and J. Lin. Hone: "scaling down" hadoop on shared-memory systems. In *VLDB Demo*, 2013.

[52] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating skew in MapReduce applications. In *SIGMOD*, 2012.

[53] W. Lang and J. M. Patel. Towards eco-friendly database management systems. In *CIDR*, 2009.

[54] W. Lang and J. M. Patel. Energy management for mapreduce clusters. *PVLDB*, 3(1):129–139, 2010.

[55] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of hadoop clusters. *HotPower*, 2009.

[56] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In *SIGIR*, 2009.

[57] J. Lin. The curse of Zipf and limits to parallelization: A look at the stragglers problem in MapReduce. In *LSDS-IR Workshop*, 2009.

[58] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[59] D.-R. Liu and S. Shekhar. Partitioning similarity graphs: A framework for declustering problems. *Information Systems*, 1996.

[60] X. Liu, J. Han, Y. Zhong, C. Han, and X. He. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *CLUSTER*, 2009.

[61] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing mapreduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, May 2010.

[62] McObject LLC. In-memory database systems: Myths and facts, 2010.

[63] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *J. Parallel Distrib. Comput.*, 69(9), 2009.

[64] N. Mitchell and G. Sevitsky. Building memory-efficient Java applications: Practices and challenges. In *PLDI Tutorial*, 2009.

[65] T. A. Neves, L. M. de A. Drummond, L. S. Ochi, C. Albuquerque, and E. Uchoa. Solving replica placement and request distribution in content distribution networks. *Electronic Notes in Discrete Mathematics*, 36:89–96, 2010.

[66] K. Y. Oktay, A. Turk, and C. Aykanat. Selective replicated declustering for arbitrary queries. In *Euro-Par*, 2009.

[67] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[68] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[69] F. Pellegrini. Scotch: Static mapping, graph, mesh and hypergraph partitioning, and parallel and sequential sparse matrix ordering package, 2007. *Cited on*, page 3.

[70] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *Supercomputing*, 2004.

[71] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 430–441, New York, NY, USA, 2013. ACM.

[72] K. Ranganathan and I. Foster. Identifying dynamic replication strategies for a high-performance data grid. In *GRID*, 2001.

[73] K. Ranganathan, A. Iamnitchi, and I. Foster. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In *CCGRID*, 2002.

[74] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, 2007.

[75] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using Hadoop on a cluster. In *Hot Topics in Cloud Data Processing*, 2012.

[76] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization, 2005.

[77] N. Selvakkumaran and G. Karypis. Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(3):504–517, 2006.

[78] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3r: increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12), Aug. 2012.

[79] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Adaptive popularity-driven replica placement in hierarchical data grids. *The Journal of Supercomputing*, 51:374–392, 2010.

[80] L. Si and J. Callan. Relevant document distribution estimation method for resource selection. In *SIGIR*, pages 298–305, 2003.

[81] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM J. Sci. Comput.*, 18(5):1436–1445, 1997.

[82] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *MapReduce*, 2011.

[83] D. Thain and M. Livny. Building reliable clients and servers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[84] P. Thomas and M. Shokouhi. Sushi: Scoring scaled samples for server selection. In *SIGIR*, pages 419–426, 2009.

[85] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2:1626–1629, August 2009.

[86] A. A. Tosun and H. Ferhatosmanoglu. Optimal parallel I/O using replication. In *ICPP*, 1997.

[87] A. S. Tosun. Replicated declustering for arbitrary queries. In *ACM symposium on Applied computing*, 2004.

[88] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *SIGMOD*, 2010.

[89] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1st edition, June 2009.

[90] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM TODS*, 22:255–314, 1997.

[91] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM TODS*, 16:181–205, March 1991.

[92] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *IISWC*, 2009.

[93] K. Zhai, J. Boyd-Graber, N. Asadi, and M. Alkhouja. Mr. LDA: A flexible large scale topic modeling package using variational inference in MapReduce. In *WWW*, 2012.

[94] L. Zhang, H. Tian, and S. Steglich. A new replica placement algorithm for improving the performance in cdns. *Int. J. Distrib. Sen. Netw.*, 5:35–35, January 2009.