

Automating Memory Management in Data Analytics

by

Mayuresh Kunjir

Department of Computer Science
Duke University

Date: _____

Approved:

Shivnath Babu, Supervisor

Jun Yang

Kamesh Munagala

Benjamin C. Lee

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2019

ABSTRACT

Automating Memory Management in Data Analytics

by

Mayuresh Kunjir

Department of Computer Science
Duke University

Date: _____

Approved:

Shivnath Babu, Supervisor

Jun Yang

Kamesh Munagala

Benjamin C. Lee

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2019

Copyright © 2019 by Mayuresh Kunjir
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

Recent years have seen unprecedented growth in the volume, velocity, and variety of the data managed by data analytics platforms. At the same time, the skilled IT staff required to develop and operate the datacenters are growing at a much smaller pace. This trend suggests a big interest in making the data analytics platforms more automatic (or, more popularly, self-driving). There are, however, several major challenges in this task. Firstly, multiple ‘one-size’ systems need to co-exist and co-operate in order to support a variety of computation needs such as log processing, business predictions, and real-time analysis. Secondly, cluster resources are managed at multiple levels exhibiting complex interactions between the many distributed system components. Finally, multiple tenants share a cluster, each with specific performance expectations restricting opportunities for optimal use of resources.

We have built an integrated management platform, called THOTH, that provides a data-centric view over the data analytics system environment. This platform is used to develop multiple auto-tuning algorithms to help systems meet their performance goals. We specifically focus on memory-based data analytics considering the growing sizes of—and effectively, more aggressive use of—memory in data processing systems. Our first contribution is a cache manager targeted at multi-tenant cluster setups. It supports a novel fairness model providing guarantees to tenants on the performance speedups experienced by their workload.

Our second contribution is automatic tuning of memory management decisions

taken at multiple levels during an application execution. This problem is approached in two ways: (i) A black-box modeling assisted with system internal knowledge, and (ii) An empirically-driven white-box approach. The two algorithms that we have developed significantly improve the state-of-the-art tuning techniques, while exhibiting different trade-offs between the convergence guarantees and the speed of optimization.

We expect the work presented here act as a major step towards building self-driving data processing systems, motivating further work in automating components such as physical design of data storage and root cause analysis of performance problems.

Contents

Abstract	iv
List of Tables	x
List of Figures	xii
Acknowledgements	xv
1 Introduction	1
1.1 Trends in Data Analytics	1
1.2 Research Overview	3
1.3 Outline	7
2 Thoth Framework	9
2.1 Introduction	9
2.2 Multi-System Management Platform	12
2.2.1 Thoth Apps	13
2.2.2 Thoth Data Manager	15
2.2.3 Other Approaches	16
2.3 Demonstration	17
2.3.1 Database-Centric View	18
2.3.2 Data Layout Recommender	20
2.3.3 Memory Management in Thoth	21

3	Multi-tenant Cache Sharing	23
3.1	Introduction	23
3.1.1	Contributions	28
3.2	Related Work	29
3.3	Fairness Properties and Policies for Single Batch	30
3.3.1	Fairness and Randomization	30
3.3.2	Basic Fairness Desiderata	32
3.3.3	Envy-freeness and the Core	36
3.3.4	Discussion	39
3.4	Approximately Computing PF and MMF Allocations	41
3.4.1	Proportional Fairness	43
3.4.2	Max-min Fairness	47
3.4.3	Fast Heuristics	48
4	ROBUS Cache Planner	51
4.1	The ROBUS Platform	51
4.2	Related Work	55
4.3	Evaluation	58
4.3.1	Setup and Methodology	59
4.3.2	Performance gains due to batching	62
4.3.3	Evaluation of batched caching policies	64
4.3.4	Discussion	71
4.4	Conclusion	74
5	Importance of Memory Tuning	76
5.1	Introduction	77
5.2	Memory Management Options	79

5.2.1	Application Tuning Approaches	83
5.3	Understanding impact and interactions	85
5.3.1	Containers per Node	86
5.3.2	Task Concurrency	89
5.3.3	Cache and Shuffle memory	90
5.3.4	Interactions with GC settings	92
5.3.5	Manually tuning an application	96
5.4	Statistics Generation	97
6	Guided Bayesian Optimization Approach	101
6.1	Introduction	101
6.1.1	AutoTuning Techniques	102
6.1.2	Our Contributions	103
6.1.3	Related Work	103
6.2	Guided Bayesian Optimization	105
6.3	White-box Model for Memory Pools	108
6.3.1	Statistics Generation	109
6.3.2	Utility Evaluation	109
6.4	Evaluation	111
6.4.1	Setup	111
6.4.2	Convergence	114
6.4.3	Quality of White-box Model	117
6.5	Discussion and Future Work	118
7	RelM White-box Approach	119
7.1	Introduction	120
7.2	Related Work	121

7.3	RelM Tuner	123
7.4	Initializer	125
7.4.1	Arbitrator	127
7.5	Evaluation	130
7.5.1	Setup	130
7.5.2	Quality of Results	134
7.5.3	Training Overheads	137
7.5.4	Sensitivity to initial profile	139
7.5.5	Ranking by utility scores	140
7.6	Need for Database Performance Data Scientists	142
8	Conclusion	143
A	BigFrame Benchmarking Service	146
A.1	Introduction	146
A.2	A Sample Benchmark	147
A.2.1	Structured Data	148
A.2.2	Semi-Structured and Unstructured Data	149
A.2.3	Graph Data	150
A.2.4	The Benchmark DAW	150
	Bibliography	152
	Biography	167

List of Tables

3.1	Utilities of cached views to tenants	27
3.2	Every tenant gets utility from a different view	33
3.3	Every tenant gets utility from the same view	34
3.4	Two tenants benefit from the same view and one benefits from a different view	35
3.5	All tenants except one get utility from the same view	36
3.6	Fairness properties of cacheing mechanisms	41
4.1	Summary of cache management policies in data analytics clusters . .	57
4.2	Test cluster setup	60
4.3	Query inter-arrival rates for different setups	68
4.4	Query time statistics on stateless cache optimizations	70
5.1	Overview of Memory Management Options	82
5.2	Test suite used in empirical analysis	86
5.3	Test cluster setup for empirical analysis	86
5.4	Changes to improve runtime and reliability of PageRank	96
5.5	Statistics derived from an application profile	98
6.1	Statistics derived from an application profile to be used in GBO . . .	108
6.2	Test cluster setup for GBO evaluation	112
6.3	Test suite used in GBO evaluation	112
6.4	Sample space used in <i>Exhaustive Search</i>	112

7.1	Test suite used in ReIM evaluation	131
7.2	Test cluster setup for ReIM evaluation	131
7.3	Default memory configuration options	131
7.4	Sample space used in <i>Exhaustive Search</i>	132
7.5	LHS configurations used in initialization	132
7.6	Comparing recommendations made by tuning policies.	135
7.7	Analysis of a GPR run for SVM.	135
7.8	Unique recommendations made by ReIM	140

List of Figures

1.1	Summary of Contributions	4
2.1	Data OS Multi-System Management	10
2.2	Thoth Multi-System Management	13
2.3	Categorization of Profiled Data	16
2.4	A workflow in BigFrame [1]	18
2.5	Visualization of BigFrame in Thoth’s workflow monitor	20
2.6	Visualization of Thoth’s data layout recommender	21
3.1	Performance comparison of caching alternatives	24
3.2	Performance as a function of fraction of input data partitions cached	26
4.1	ROBUS platform	52
4.2	Input data size distribution	61
4.3	Workload generation in ROBUS	62
4.4	Workload throughput on different cache granularities	63
4.5	E2E latency comparison of cache policies	64
4.6	Effect of data sharing changes on four equi-paced tenants	66
4.7	Effect of the variance in query arrival rates	68
4.8	Effect of changing number of tenants	69
4.9	Effect of batch size on four equi-paced tenants setup	70
4.10	Fraction of time popular views cached by algorithms	72
4.11	Fairness index as a function of number of batches	73

4.12	Comparing Batching Delay and Execution Time when Batch Size=10 seconds	73
5.1	Motivation example showing interplays in memory management . . .	78
5.2	Node memory managed by Resource Manager	80
5.3	Container memory managed by JVM	80
5.4	Heap managed by application framework	80
5.5	Impact of number of containers on performance	87
5.6	Understanding application failures	87
5.7	Impact of Task Concurrency on performance	89
5.8	Impact of Cache Capacity and Shuffle Capacity on performance . . .	91
5.9	Interactions between NewRatio and Cache Capacity on K-means . . .	93
5.10	Impact of NewRatio on GC overheads	93
5.11	Memory usage timelines explaining a failure	94
5.12	Interaction between NewRatio and Shuffle Capacity	95
6.1	Workflow of Guided Bayesian Optimization (GBO)	105
6.2	Accuracy of White-box model estimates for K-means	109
6.3	Accuracy of White-box model estimates for Sort	110
6.4	Training overheads in GBO	114
6.5	Speed of convergence of tuning policies	115
6.6	Impact of inaccurate statistics on predictions	117
7.1	Application tuning process in ReIM	124
7.2	Working of ReIM's <i>Arbitrator</i> algorithm	128
7.3	Performance comparison of tuning policies	134
7.4	Comparison of TPC-H Queries run using <i>MaximizeResourceAllocation</i> policy and using ReIM.	136
7.5	Training overheads of black-box and white-box tuning.	137

7.6	Understanding sensitivity of ReIM recommendations to the initial profile	138
7.7	Analyzing sensitivity of statistics to initial profile	141
7.8	Evaluating accuracy of configurations recommended by ReIM	141
A.1	Data model of our benchmark DAW	148

Acknowledgements

I am sincerely and heartily grateful to my advisor, Shivnath Babu, for his support and guidance during my graduate studies at Duke University. I especially admire him for so many thought-provoking discussion sessions which greatly helped me evolve as a researcher. I would also like to extend my gratitude to my co-advisor, Jun Yang, for mentoring and motivating me throughout. I am also grateful to my other committee members, Kamesh Munagala and Benjamin C. Lee for their support and advice during this work.

I would like to acknowledge all my colleagues at Duke University for their friendship and big support. It was a great pleasure to be part of an inspired data management team including Abhishek Dubey, Prajakta Kalmegh, Seunghyun Lee and Yuzhang Han during the beginning stages of my doctoral studies. I am equally grateful to my esteemed collaborators during this time: Eric Lo, Andy He, and Malu Castellanos for exposing me to big data analytics; Brandon Fain and Kamesh Munagala for strengthening my work with their expertise in theory; Janardhan Kulkarni for engaging discussions in resource management as well as for helping me adapt to life in Durham; Sudipto Das for being a great mentor during my internship at Microsoft Research. I would also like to offer my sincere gratitude to Marilyn Butler who has been a constant pillar of support during my time at Duke University.

This work would not be complete without a great support and many sacrifices of my family. I am sincerely grateful to my parents who always motivated to follow my

dreams. Last but not the least, I would like to extend my eternal love to my wife Myrah who has strongly stood by me through thick and thin and always encouraged me to give my best.

Introduction

1.1 Trends in Data Analytics

Recent years have seen a dramatic increase in data collected from various sources and in different formats. The global digital data footprint is expected to grow from 33 Zettabytes (One zettabyte=A trillion gigabyte) in 2018 to 175 Zettabytes by 2025 [2]. Rapid growth in social networking, mobile devices, facilities, equipment, R&D, simulations, and physical infrastructure are all contributors to this increase in data flow [3]. It is important to assess the impact of these trends while developing software systems for modern data analytics.

No-one-size-fits-all: The “no-one-size-fits-all” philosophy [4] of system design has led to a variety of systems being developed in recent years. Examples include MapReduce, NoSQL systems, column-stores, data stream analytics, in-memory databases, and many others. A traditional DBMS usually has one execution engine and one storage engine. Typically, these two engines are coupled tightly with each other and are not usable individually. In contrast, a modern data analytics cluster contains multiple “one-size” compute and storage systems internally. Consequently, it has

now become imperative to provide an integrated management solution that provides a database-centric view of the underlying multi-system environment. An application developer is burdened with the task of choosing an *appropriate* system configuration based on resource availability and changing application requirements. This entails considerable manual effort in tuning each system individually based on available resources.

Memory-centric designs: Jim Gray in 2006 predicted that “Memory is the new disk”. True to his prediction, in-memory data analytics is increasingly the focus in recent data processing platforms. The growing popularity of systems like Apache Spark, SAP HANA, RocksDB, Apache Arrow, and Druid highlight these trends. Memory-based processing has been shown to provide up to two orders of performance improvement over disk-based processing [5]. Increasing memory sizes not only mean that more data will be cached but also that distributed computations will use memory more aggressively. As a result, the performance of data analytics workload is more dependent on memory management than before. Its implication in a multi-system architecture, where the resource management decisions are taken at multiple levels, is that the impact and the interactions among the many levels of memory management decisions needs to be systematically modeled.

Multi-tenant nature: Current data analytics systems process huge volumes of data collected on a daily basis through transactions, clickstream logs, social media, etc. As it is expensive to move data at such a large scale, the systems move computations to the data instead. Popular analytics systems, *e.g.*, Hadoop and Spark, support system designs wherein multiple tenants share hardware resources as well as data typically residing in a decoupled distributed file system such as HDFS. Such multi-tenant designs allow for a high resource utilization by removing a need to provision for the

peak load of each tenant. Multi-tenancy brings in the challenge of bringing fairness to workload management since each tenant expects certain performance guarantees expressed in the form of Service Level Agreements (SLAs).

Autonomic Computing: While the amount of data managed in enterprise datacenters is growing by a rapid pace, the number of skilled IT staff—such as application developer and system administrators—to develop and operate the datacenters are increasing by a much smaller pace [6]. The implication of this trend is clear: the system-to-staff ratios are increasing very quickly. High system-to-staff ratios are only possible through extensive automation of system management tasks. Significant progress has been made to automate certain administrative tasks such as backups, storage defragmentation, index rebuilds [7]. But some other tasks such as root cause discovery and configuration tuning still lack sufficient automation to make the systems truly autonomic (or self-driven) [8,9].

1.2 Research Overview

A management layer on top with a database-centric view of systems can use a learning-based approach to recommend the right system to use for an application. This is a functionality of database management systems that will be hard to provide with a purely OS-centric approach. In [10], a case was made for building a unified DBMS⁺ system in a shared-cluster environment that understands an application's requirements and finds the *best fit* system configuration for executing the application. To think of database-as-a-service is more convenient for system administrators and application developers for better resource provisioning and data layout. This also relieves them of significant effort in tuning each system separately, allocating resources, and most importantly choosing a system most suitable for an application.

In order to build DBMS⁺ systems, Herodotou [11] proposes a *profile-predict-*

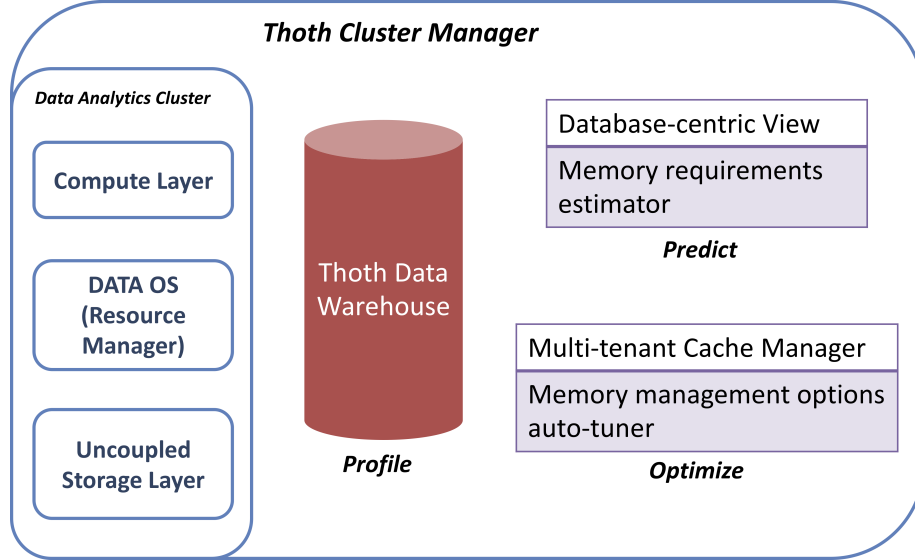


FIGURE 1.1: Summary of Contributions

optimize approach. This approach entails: (a) collecting dynamic profiling data in order to learn run-time behavior of complex workloads; (b) deploying cost-based or black-box models to estimate the impact of hypothetical tuning choices on workload performance; and (c) using efficient search strategies to find tuning choices leading to efficient performance.

In this dissertation, by following the *profile-predict-optimize* approach, we design a prototype for DBMS⁺: THOTH [12]. THOTH provides a data-driven platform to develop various cluster manageability applications with a vision to provide a database-centric view for integrated management of resource provisioning and data layout as well as for development of a multi-system aware optimizer.

Figure 1.1 provides a summary of the contributions made in this dissertation. The backbone of the THOTH cluster management platform is a data manager which collects data in the form of system logs, runtime statistics, resource profiles, metadata and configuration options from data processing systems in a non-invasive manner. This rich data is consolidated to a warehouse through which it is made available to various cluster management *apps*.

Memory management is critical to the performance of data analytics workloads as suggested by the trends in hardware and software developments. We have used the THOTH platform for memory management in data analytics clusters under the premise that modern data analytics will increasingly be done in memory on shared-nothing clusters. Some key characteristics of such clusters include: (a) Container-based isolation, (b) Memory intensive computations, (c) Support to multi-tenancy, and (d) Java Virtual Machine (JVM) dependence.

In [13], we identify a major research challenge on bringing *fairness* to the allocation of memory in the presence of multi-tenant workloads. Data analytics platforms need to support multi-tenant workloads in order to make the most efficient use of computational resources. Under such commercial setups supporting multi-tenancy, memory used for caching data, unlike other resources like CPU and disk, can be shared by multiple tenants at the same time. Unlike traditional database systems which use caches only as a buffer pool [14], modern data analytics systems also expose data cache through external APIs [15]. Since cache is a limited resource, policies to decide *what* and *when* to cache are essential to make the best use of the resource.

In the light of these challenges, we design allocation policies for a shared resource like cache that provide a novel fairness model incorporating not only the more standard notions of Pareto-efficiency and sharing incentive, but also envy-freeness via the notion of *core* from cooperative game theory. The policies are built-in to a system prototype we developed on Apache Spark, called ROBUS [13]. ROBUS analyzes tenant workloads in small online batches to find savings for a cache-based optimization. By systematically planning cache allocation, ROBUS provide fairness guarantees to the tenants of the data analytics cluster of the speedups provided to the performance of their workloads.

Another major memory management challenge faced by the data processing platforms arises from the fact that the memory allocation decisions are made at multiple

levels (viz. the resource-management level, within a container, at the application level, and inside the JVM) with complex interplays involved amongst the decisions and performance metrics. We first identify the space of multi-level memory management options in common data analytics systems and establish the importance of tuning them [16, 17]. Using the profiling data available in THOTH, we develop two contrasting approaches to auto-tuning: (1) A relatively black-box approach assisted with system internal knowledge, and (2) An empirically-driven, rule-based, white-box approach.

Our first solution, Guided Bayesian Optimization (GBO) [18], augments a powerful black-box model of Bayesian Optimization with a simple approximate white-box model capable of separating good configurations from bad ones. Using the white-box model as a guide during the exploration of configuration space, the tuning process is shown to speed up significantly across a variety of application workloads.

The second solution to tuning uses an empirically-based white box approach. The solution, called ReIM, provides fast tuning of memory management options for a reliable and a resource-efficient execution of applications in JVM-based data analytics systems. ReIM first uses a profile run to understand the requirements of various memory pools used at multiple levels during application execution. The profile data is used by a set of rule-based white-box models to come up with an assignment of configuration options expected to provide a reliable and the most resource-efficient execution. The quality of results of the ReIM tuner is comparable to state-of-the-art black-box optimization approaches which often take significantly longer time.

The two approaches we use towards memory management options auto-tuning are both shown to be much faster and more robust compared to the existing solutions. Between the two, they exhibit a trade-off between the convergence guarantees and the tuning overhead making a case for co-existence in autonomous data processing systems.

1.3 Outline

This dissertation starts off with an overview of THOTH multi-system cluster management platform in Chapter 2. We provide system implementation details as well as a demonstration using BigFrame benchmarking service [1] that we have developed. The BigFrame service is used in monitoring and identifying performance bottlenecks in a multi-system application pipeline. Details on the development of the BigFrame service are given in Appendix A.

Chapter 3 introduces the problem of cache allocation in presence of multi-tenant workloads and develops cache allocation policies that bring fairness to the optimization choices. We propose two randomized policies based on economics and game theory, *viz.*, Max-min Fair and Proportional Fair. Furthermore, we develop approximate algorithms to implement these policies. Chapter 4 builds a system framework for cache management, called ROBUS, that processes multi-tenant workload in a batched manner using the cache allocation policies developed in the previous chapter. A system prototype of ROBUS is developed in Apache Spark, a popular data analytics platform. An evaluation is carried out using synthetically created industry-standard use cases showcasing how the system brings fairness to tenants' performance.

Chapter 5 shifts focus to hierarchical memory management in data analytics systems. It identifies the impact and interactions of memory management options across multiple levels during an application execution. A detailed empirical study is carried out to put forth guidelines in tuning. Chapter 6 develops a relatively black-box solution to automatically tune memory management options. Using the findings of the empirical study, it builds a simple white-box model to be used during exploration thereby optimizing the process of tuning. Chapter 7 also uses the empirical study carried out in Chapter 5 in order to develop a faster white-box auto tuner

for memory management options. The algorithm we develop provides a reliable and resource-efficient configuration using statistics obtained from a profile run. Evaluation results compare our approach to industry-standard robust policies as well as to a popular black-box approach.

Finally, Chapter 8 concludes with our contributions and future directions.

Thoth Framework

2.1 Introduction

A data-driven enterprise today needs systems for advanced computations like click-stream log analysis, getting real-time insights into streaming data, business forecasting, and near-interactive experience for large volumes of data, etc. Organizations like Facebook, Netflix, LinkedIn, and Yahoo are adopting many ‘one-size’ systems that cater to these requirements. Different ‘one-size’ systems serve different processing needs such as batch processing, bulk-synchronous MPI processing, key-value stores, in-memory analytics, graph computations, and columnar storages.

Following the ‘no one size fits all’ philosophy, active research in big data platforms is focussing on creating an environment for multiple ‘one-size’ systems to co-exist and co-operate in the same cluster. Sharing a cluster across multiple systems is desired in order to save on huge data migration costs involved in dataflow pipelines. Some of the solutions existing today include Apache Yarn [19], Mesos [20], Facebook’s Corona, and Google’s Omega [21]. The approach taken by all these systems is to provide a management layer capable of managing resource allocation across systems.

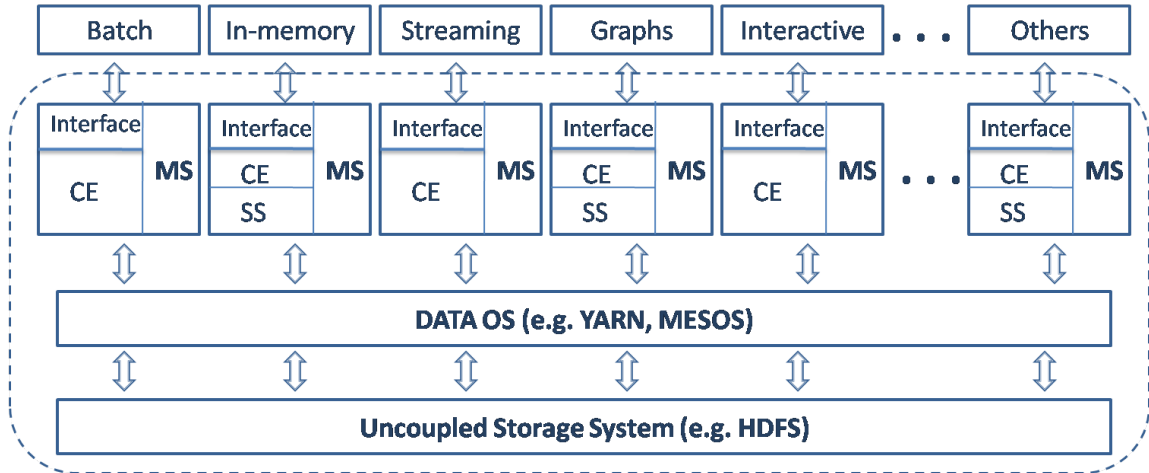


FIGURE 2.1: Data OS Multi-System Management

We call this layer *Data OS* since it is functionally closer to an operating system for data. Figure 2.1 shows a broad architectural view of these systems.

However, OS functionalities are incapable of managing database-specific needs of systems. The ‘no one size fits all’ philosophy poses many challenges for system administrators and application developers. They are faced with the challenge of finding a system most suitable for their applications since each system may serve many overlapping classes of problems. For instance, log analysis can be performed on parallel databases like Teradata/Greenplum, batch systems like Hadoop, column-oriented systems for efficient OLAP analysis like Vertica, or in-memory analytics with systems like SAP HANA. An application developer is burdened with the task of choosing an *appropriate* system based on resource availability and changing application requirements. This entails considerable manual effort in tuning each system individually based on available resources.

A management layer on top with a database-centric view of systems can use a learning-based approach to recommend the right system to use for an application. This is a functionality of database management systems that will be hard to provide with a purely OS-centric approach. In [10], we made a case for building a unified

DBMS⁺ system in a shared-cluster environment that understands an application’s requirements and finds the *best fit* system for executing an application. To think of *database-as-a-service* is more convenient for system administrators and application developers for better resource provisioning and data layout. This also relieves them of significant effort in tuning each system separately, allocating resources, and most importantly choosing a system most suitable for an application.

In this work, we introduce a prototype for DBMS⁺: **Thoth**. **Thoth** provides a platform to develop various multi-system manageability applications (called *apps* in short). The applications could range from a dashboard to monitor running applications to a multi-system optimizer. We first list down some of the major research challenges that drive our vision:

- **Database-centric view:** Create a view over multi-system cluster by integrating monitoring data from all the systems and all the layers of software stack for users to be able to easily track and understand the performance of their applications without a need to understand the internals of each system.
- **Auto-tuning:** Build a multi-system aware optimizer capable of picking right system for execution along with settings for its tuning knobs.
- **Dynamic Resource Provisioning:** Find best resource allocation strategy across systems by studying historical workflow traces. Use it to allocate resources on-the-fly to meet applications’ demands.
- **Data Layout Management:** Provide recommendations on data format, storage engine, partitioning and materialization based on usage patterns of workloads.

Contributions

- Proposal of a multi-system management architecture: $DBMS^+$ and design of a prototype: *Thoth*. *Thoth* provides a data driven platform for various cluster management *apps*.
- Data management utility for *Thoth*: *Thoth DM*. It provides a unified view over data collected from multiple systems running on cluster for *apps* to make system management decisions.
- A dataflow visualization *app* and a data layout management *app* developed in *Thoth*. While dataflow visualizer allows real-time monitoring of applications, data layout manager processes data profiled by *Thoth DM* to recommend changes to data layout.

Outline

The rest of the chapter is organized as follows: Section 2.2 describes how *apps* in *Thoth* contribute towards building a multi-system management layer. It also illustrates the methodology for collection and management of data. Section 2.3 provides a demonstration of two *apps*: 1. Dataflow visualizer, and 2. Data-layout recommender.

2.2 Multi-System Management Platform

We propose a prototype of $DBMS^+$ architecture, a multi-system management platform, called *Thoth*. Let's first look at some of the important design goals of *Thoth*.

1. **Extensible:** *Thoth* should provide an extensible platform for the development of manageability features that cannot be provided by *Data OS*.
2. **Data-driven:** *Thoth* should enable data-driven decisions which means an ability to collect and process data generated by systems is required.

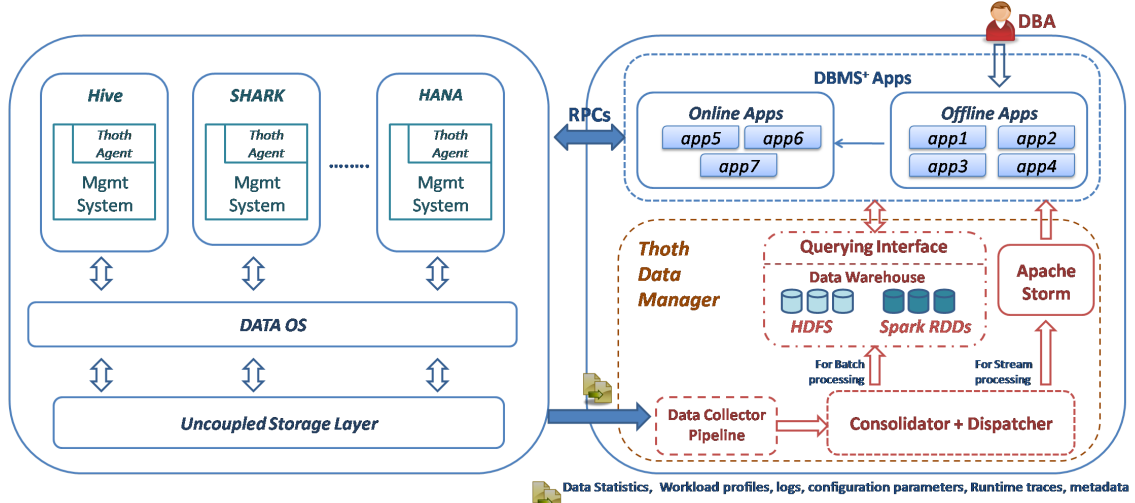


FIGURE 2.2: Thoth Multi-System Management

3. **Real-time:** Thoth should support real-time and frequent decision making.
4. **Online and offline modes:** Some of the manageability decisions are more of recommendations to administrators while some other may change the state of systems. Thoth should support both modes of functioning.
5. **Co-operative development:** Thoth should interact with the systems and *Data OS* layers through well-defined APIs so that Thoth *apps* can be developed side-by-side as the systems and *Data OS* evolve.

2.2.1 Thoth Apps

Thoth provides a platform for the development of various multi-system management applications, *apps* in short. These *apps* help Thoth achieve easy extensibility. Figure 2.2 shows the Thoth platform; the backbone of Thoth is a data manager which enables data-driven decision making. Thoth DM collects data in the form of system logs, runtime statistics, resource profiles, metadata and configuration options from a multi-system cluster. It consolidates and stores the data in a warehouse through which it is made available to *apps*. We discuss the Thoth DM design in Section 2.2.2.

DBMS⁺ *apps* register themselves with Thoth DM to receive a subset of instrumented data. The *apps* can be of two types: (a) *offline apps* and (b) *online apps*. The *offline apps* analyze data to supply recommendations on system design or system monitoring data to database administrators. DBAs can use this information to better design applications or to better tune systems. Since these *apps* do not take any action on systems by themselves, they are termed *offline apps*. The *online apps*, on the other hand, are capable of executing an action that may change system design. To invoke these actions, we add *agents* inside systems. The *online apps* communicate with the agents using a RPC mechanism. This allows for development *apps* without intruding system space much, one of the foremost goals of DBMS⁺ design.

We list down a few small *apps* to showcase the potential of DBMS⁺ platform. A discussion on building a fully functional multi-system optimizer prototype is left out of the scope of this document. In section 2.3, we give a demo for *app1* and *app2* from the list provided below. Later chapters provide detailed solutions for two more apps from the list, viz. *app6* and *app7*.

- *offline apps*

app1 Dashboard for administration and monitoring a.k.a. Database-Centric View

app2 Data layout recommender

app3 Multi-tenant resource allocation recommender

app4 Prediction for system tuning parameters

- *online apps*

app5 Data layout auto-tuner

app6 Multi-tenant resource allocation auto-tuner

app7 Cluster Configuration Parameters auto-tuner

2.2.2 *Thoth Data Manager*

Thoth DM is a single entry point for all application logs and profile data. Its functionalities include: (a) consolidating data to a system-agnostic form; (b) maintaining a warehouse for data; and (c) supplying relevant data to various *apps* of DBMS⁺. Key challenges in Thoth DM are the choice of data collection granularity and a need of system-agnostic traces. We describe how Thoth DM addresses these challenges next.

Query execution on any distributed system can be imagined as being carried out in three levels – query, job, and task. A query is executed as a multi-stage process. Each stage, alternatively called a job, is split into multiple tasks distributed across the cluster. We collect data at all three granularities. The data includes configuration settings, metadata, runtime statistics, and profiles. We use lightweight agents for system instrumentation keeping in tune with our goal of building Thoth manageability functionalities without intruding systems space much. Thoth DM consolidates the collected data into multiple categories shown in Figure 2.3. This allows for optimizing storage and supplying only the relevant data to *apps*.

Figure 2.2 shows the Thoth DM workflow. Data is generated by various events in life-cycle of queries. It is collected by the data collector module of Thoth DM and passed on to a data consolidator, which is implemented as an ‘interceptor’ in Apache Flume. Once consolidated, data can be dispatched to consumers. A consumer can either be a streaming *app* or Thoth data warehouse. Real-time processing *apps* require data to be sent over a stream in Apache Storm. The *apps* doing a batch processing, on the other hand, require data to be stored in the data warehouse. Thoth DM’s supports data to be stored either on HDFS or in Spark RDDs as per

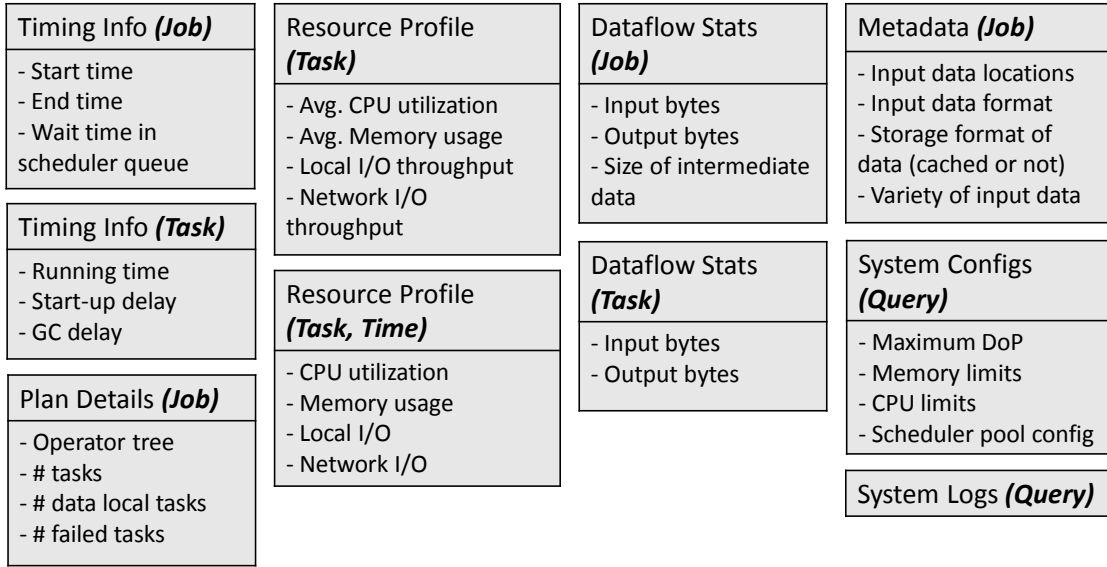


FIGURE 2.3: Categorization of Profiled Data

app requirements. To extract required information from data warehouse, *apps* use a querying interface provided by Thoth DM.

Data stored in warehouse exhibits a large variety; for example the query logs are highly unstructured, the resource profiles adhere to a fixed structure, and the query plans have hierarchical/nested structure. Thoth *apps* may favor one storage system over other based on the type of data they need. This points to an interesting research challenge in finding right data layout for Thoth data warehouse which we plan to address in future.

2.2.3 Other Approaches

Apache Chukwa [22] is a data collection system that enables monitoring of distributed systems. Its scope is limited to large scale log collection on Hadoop HDFS. It also provides a reporting and monitoring framework for the near real-time analysis of the cluster. Our solution of Thoth DM enables but is not limited to log collection on HDFS. Another key difference is that Thoth DM allows streaming of data to

Apache Storm for efficient real-time processing of events. Netflix’s adoption of Chukwa in Suro [23] provides a support for not only arbitrary data formats but also real-time processing of data. Suro provides central data management for Netflix applications running in different clusters. Our vision, however, is to contribute a database management layer for a multi-system cluster.

Splunk [24] provides a real-time platform for proactive monitoring of large volumes of machine data. It collects data from various sources like logfiles, messages, config files, tickets, messages, etc and makes it available for further analysis. Splunk also promotes the notion of apps and categorizes them into application management, business analytics, etc. thus enabling more advanced analytics. Our solution is motivated by Splunk but highly differs in our goal, that of providing an integrated management solution for the multi-system environment. Our prototype of Thoth DM collects and consolidates data relevant for provisioning of core DBMS functionalities in the form of *apps*.

2.3 Demonstration

The purpose of this demonstration is to get the vision of DBMS⁺ across to community. Thoth provides a rich platform to develop various management *apps* with a database-centric view of underlying systems. It is made possible by Thoth DM utility that collects data (statistics, profiles, configurations, etc.), consolidates, and supplies it in real-time to Thoth *apps*.

A demo cluster is set up on Amazon EC2 cloud with two systems: (a) Hive over Hadoop, and (b) Shark over Spark. Two Thoth *apps* are developed. First *app* processes data in real-time and streams it to a dashboard which supports monitoring execution flow and understanding performance. The second *app* analyzes historical workload traces to recommend dynamic changes in data layouts.

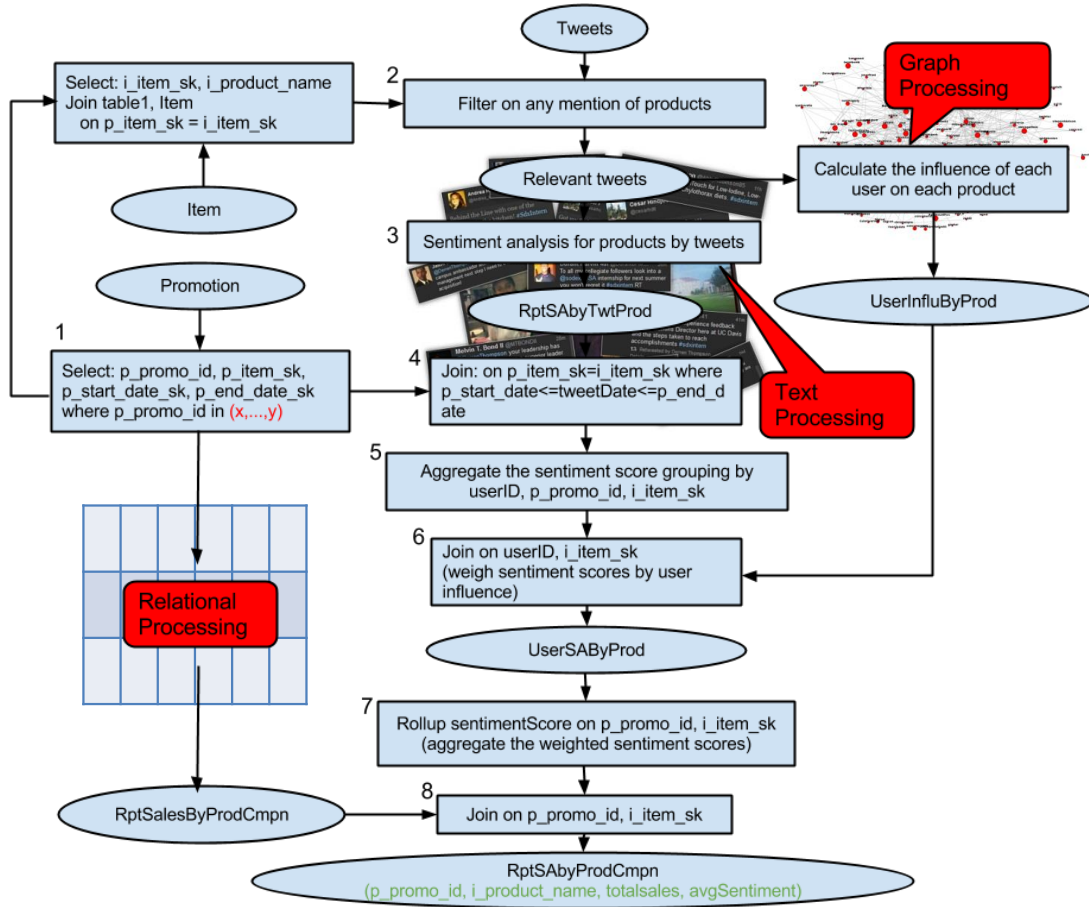


FIGURE 2.4: A workflow in BigFrame [1]

2.3.1 Database-Centric View

Many cluster monitoring tools are available to assist DBAs, e.g. Ganglia [25]. These tools have a *system-centric* view of the cluster which means they treat database applications like any other processes using hardware resources. Such a narrow view is good enough for many *Data OS* management decisions such as provisioning resources in case of failures. But for a DBMS^+ management layer, what is needed is a *database-centric* view of systems. An understanding of the intricacies of database systems is essential for admins to better tune systems or better design applications. e.g. To diagnose a query failure, an understanding of the behavior of jobs during execution

of the query would help pinpoint whether the failure was due to: a sub-optimal query plan; an inferior data layout; or insufficient resource allocation. This is where **Thoth** helps with its management of data collected from system instrumentation and logs. Another important contribution of **Thoth** is to provide a consistent view of multiple systems sharing the cluster. An admin managing a multi-system cluster may not be equipped with the expertise of each system but rather interested in a high level easy-to-understand information that can assist her find best settings for applications.

We use BigFrame benchmarking service [1] to exhibit **Thoth**'s role in managing a multi-system cluster. BigFrame is our effort targeted towards creating a *benchmarking-as-a-service* solution for big data analytics. Unlike the existing benchmarks, e.g TPCDS, HiveBench, etc., which are either micro-benchmarks or benchmarks in very specific domains, making them not fit into the big data environment today, BigFrame can instead generate the benchmark tailored to a specific set of data and workload requirements. For example, one category of enterprises may be grappling with increasing data volumes; the variety and the velocity of their data not being pressing concerns. Another type of enterprises may be interested in benchmarking systems that can handle larger volumes, with the volume of unstructured data dominating that of the structured data. Third category of enterprises may be interested in understanding reference architectures for data analytics applications that need to deal with large rates of streaming data (i.e., a high velocity of data). More details on BigFrame design are provided in Appendix A.

One application workflow provided by the BigFrame benchmarking service is shown in Figure 2.4 as an example. The workflow exhibits a variety of use-cases: (a) SQL queries with relational operators; (b) Statistical analysis and machine learning algorithms from NLP applied to tweets to extract user sentiments; (c) Iterative graph computations to compute user influence that weights their sentiment scores. When presented with such a workflow and with multiple possible paths of execution,

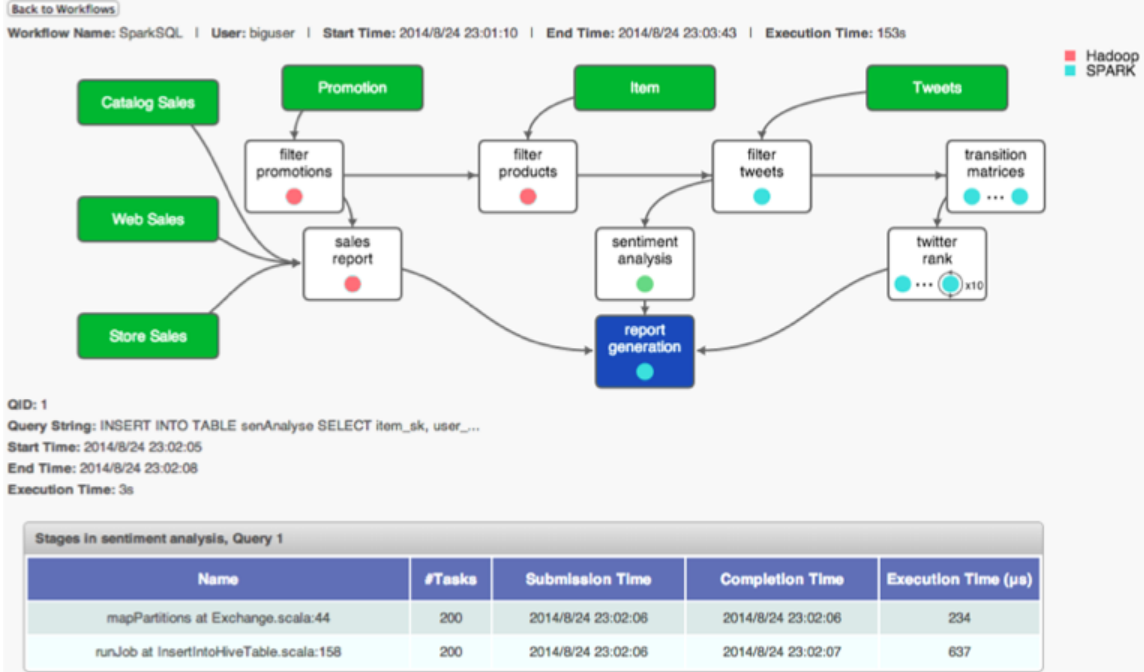


FIGURE 2.5: Visualization of BigFrame in Thoth’s workflow monitor

not just making the *right* choice of systems for execution but also understanding the behavior of systems under such a workload in itself is a big challenge for system admins. We demonstrate how Thoth dashboard *app* (*app1*) assists admins on BigFrame workloads.

The dashboard *app* receives data from Thoth DM in form of a stream sent over Apache Storm. It builds visualizations that show various features like query plans, resource utilization profiles, dataflow statistics, etc. Each of the visualization includes tuning options to pick a set of statistics to be included, the granularity for summarization, and so on. A sample visualization is shown in Figure 2.5.

2.3.2 Data Layout Recommender

We demonstrate another *app* in Thoth that provides workload-aware recommendations for data layout (*app2*). The purpose is to showcase how workload traces can be used in building powerful models for tuning data layouts across multiple systems.

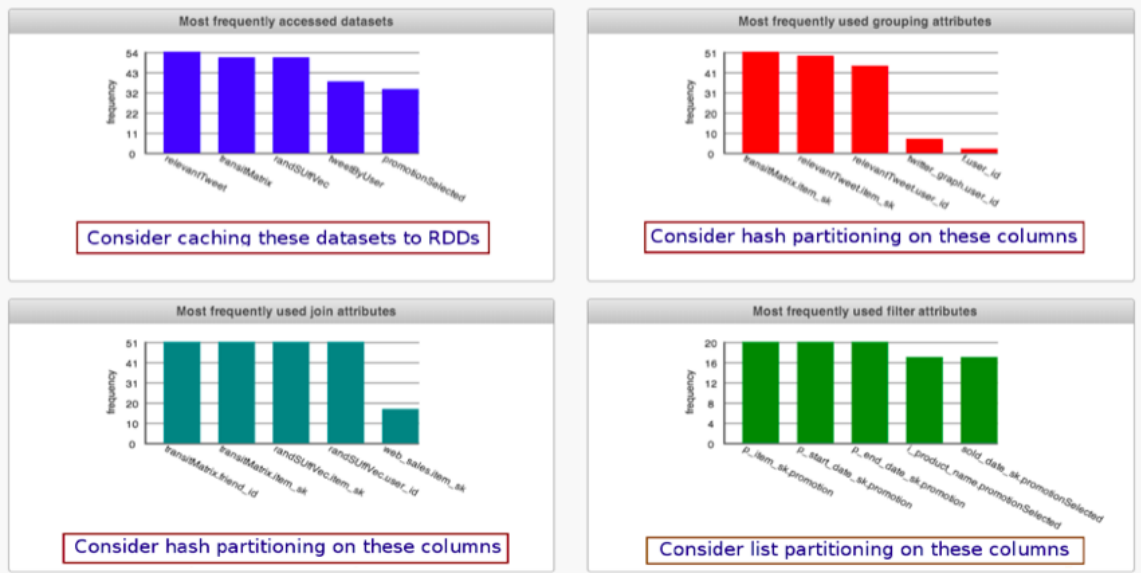


FIGURE 2.6: Visualization of Thoth’s data layout recommender

According to a study carried out in [26], analytical workloads exhibit interesting temporal computational patterns that call for dynamic changes in system design. The flow of *app2* looks like the following: Trace is periodically queried from Thoth warehouse; Models are invoked over the trace to get recommendations on physical design changes such as data format, materialization, or partitioning; The recommendations are relayed to DBA who can use them in re-structuring of data. Figure 2.6 presents an example visualization providing recommendations to user based on a profile of their workload execution.

2.3.3 Memory Management in Thoth

The rest of the document focuses on building memory management in Thoth platform. The particular contributions can be categorized as the following two *apps*:

1. Multi-tenant resource allocation auto-tuner
2. Cluster configuration auto-tuner

Chapter 3 and Chapter 4 develop a cache allocation manager for multi-tenant

analytical workloads. Therein, we develop a framework which supports a *fair* usage of cache by tenants running data analytics workloads in an automated manner.

Chapter 5 outlines the importance of configuration parameters relating to memory management in data analytics applications through empirical analysis. Following which, Chapter 6, and Chapter 7 provide solutions to automatically tune the configuration parameters in order to reliably and efficiently run user applications.

Multi-tenant Cache Sharing

3.1 Introduction

Two recent trends in data processing are: (i) the aggressive use of memory to speed up processing by caching datasets [15,27], and (ii) the use of multi-tenant clusters for analyzing large and diverse datasets [19]. The growing popularity of systems like Apache Spark [15] and SAP HANA [27] highlight these trends. The challenges in designing a caching policy on a multi-tenant distributed analytics platform are multi-fold:

Caching has become ubiquitous.

Traditionally, caches were only used internally in database systems as a buffer pool [14] for recently accessed data pages from disk. Modern data analytics systems additionally expose APIs for data caching to application developers and data analysts.

```

1 // read sales data (id, year, product, city, sales)
2 val sales = sc.textFile("sales.txt").map(_.split(","))
3 // find sales of the current year and cache
4 val salesThisYear = sales.filter(_(1)="2016").cache
5 // find sales in the city "SF" this year
6 salesThisYear.filter(_(3)="SF").map(_(4).toInt).sum
7 // find sales of the product "iCar" this year
8 salesThisYear.filter(_(2)="iCar").map(_(4).toInt).sum

```

Listing 3.1: Example Spark program showing user-directed caching

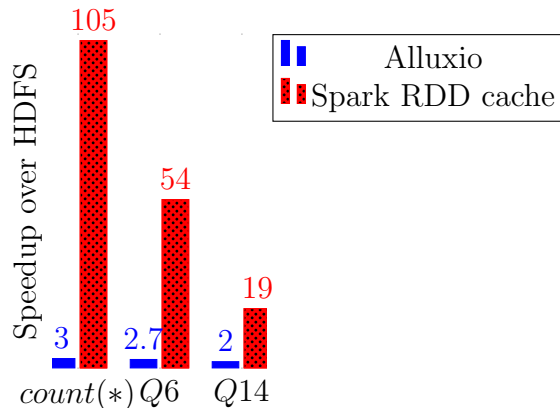


FIGURE 3.1: Comparing performance of different caching alternatives for TPC-H queries

For example, Spark introduces an abstraction called *Resilient Distributed Dataset* (*RDD*) to represent any data relevant to modern analytics: files (on a local or distributed file-system), tables (horizontally or vertically partitioned), vertices or edges of graphs, statistical models learned from data, etc. A user can create an RDD directly from data residing on a local or distributed file-system, or by applying a transformation to one or more other RDDs. The user can then direct the system to cache the RDD in memory. Listing 3.1 gives an example.

As we show in Figure 3.1, computations done on RDDs cached in memory run one to two orders of magnitude faster than when the data resides on disk [15]. The experiment in Figure 3.1 was carried out on a 10-node cluster using three queries of varying complexity on TPC-H data stored in text format on the Hadoop Distributed FileSystem (HDFS). Two different architectures for caching were considered in Figure 3.1: (i) caching using Spark’s internal RDD cache store, and also (ii) caching RDDs on an external memory store called Alluxio [28]. Spark’s RDD cache provides much bigger speedups because it stores data in an optimized columnar format, whereas Alluxio simply mimics the data layout on disk (which, in this case, is row-based) in memory.

Caching under multi-tenancy is nontrivial.

Cache is always a limited resource since the total size of memory in a cluster is usually orders of magnitude smaller than the data sizes stored and queried in the cluster. As we saw in Figure 3.1, in a multi-tenant cluster that has multiple users, massive performance speedups will be experienced by tenants who get to use the in-memory cache. Thus, it is highly desirable in multi-tenant clusters that low-priority tenants should not be able to hog the available cache, and prevent other tenants from getting the performance benefits they deserve. Unlike a resource like a CPU core which is used by one tenant at a time, a cached data item can simultaneously benefit a high-priority and a low-priority tenant. Furthermore, different tenants will have different utilities for objects that could be placed in the cache.

Sensitivity of parallel computations to the fraction of cached input:

Recent studies have shown that the speedups in parallel computations are very sensitive to how much of the input data is cached [29]. Since the performance of a parallel job is determined by the slowest of its parallelly-running tasks, the job's performance does not improve much unless data partitions needed by all parallelly-running tasks are found in cache [29]. Figure 3.2 shows this behavior for a *K-means* clustering program run in Spark by varying the fraction of input data partitions cached. (Partition is the smallest unit of data Spark tasks can process.) Notice how quickly the performance speedup vanishes once some part of the input data is not found in the cache.

When faced with such challenges, traditional cache allocation policies can lead to user dissatisfaction, poor or unpredictable performance, and low resource utilization. We will illustrate the problems and opportunities through an example.

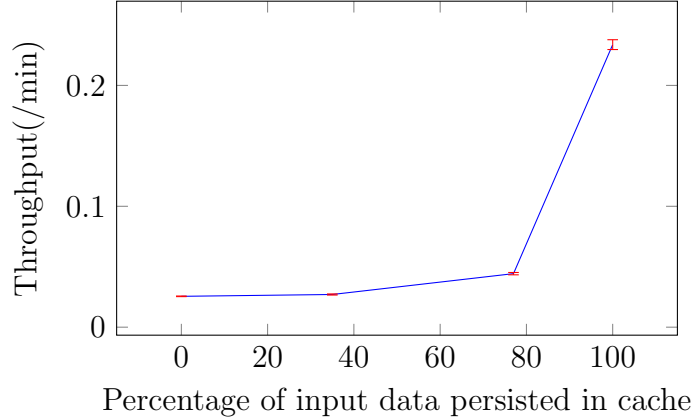


FIGURE 3.2: Performance of K-means clustering program in Spark when the fraction of input data persisted in Spark’s RDD store is varied.

Multiple tenants: The predominant practice in the industry is to group similar users—e.g., all users submitting batch jobs—into queues (or, pools). Each queue forms a tenant in the cluster. Our example uses three tenants: (i) *Analyst*, the business analysts in the company, (ii) *Engineer*, the developers in the company who build data-driven applications such as recommendation models, and (iii) *VP*, the top-level management in the company such as the CEO and the Chief Security Officer who look at hourly and daily reports.

Cacheable entities: These three tenants will benefit from caching one or more of three *views*— R , S , and P —each of size M bytes. Throughout this paper, “view” refers to any data item that can be cached to give a performance benefit. For SQL workloads, a view corresponds to a SQL expression, like any candidate view generated by a materialized view selection algorithm [30–33]. For broader data analytics—e.g., graph processing—a view corresponds to a dataset on which the user has put a cache directive (recall the example Spark program from Figure 3.1).

Utilities: The matrix in Table 3.1 shows the *utility* that each tenant gets if the corresponding view were to be cached in memory. A simple definition of utility we will use in this paper is the savings in I/O because data is read from the in-memory

Table 3.1: Utilities of cached views to tenants

Tenant	R	S	P
Analyst	2	1	0
Engineer	2	1	0
VP	0	1	2

cache versus disk. For example, if view R is cached in memory, then tenant *Analyst* will get a utility of two units. One common pattern in multi-tenant clusters that we bring out in Table 3.1 is that view R could be the detailed logs that business analysts and developers access quite often; view P could be a table that only the top-level management has access to; while view S could be a materialized view with aggregated information shared by all tenants.

Scenario 1: Suppose the in-memory cache has a total size of M bytes. If a static and equal partitioning of the cache is used, each tenant will be entitled to $\frac{M}{3}$ bytes of cache memory. Recall that each of the views R , S , and P are M bytes each; so none of them will fit in their $\frac{M}{3}$ bytes of cache. If, instead, the cache is kept unpartitioned, and a *Least Recently Used (LRU)* policy is used for cache allocation, the view R will likely remain cached for the most time on account of it being the most used. Thus, the Analyst and Engineer tenants will see performance speedups. However, the VP tenant’s workload will not see any benefits.

Scenario 2: To satisfy the VP tenant, suppose the cluster admin decides to give the VP tenant 50% higher priority than the other tenants. So, she assigns weights to the Analyst, Engineer, and VP tenants in the ratio 1 : 1 : 1.5. The cache is now allocated based on the weighted utility of the tenants. But, even with this change, view R will be the only one cached since it has the highest weighted utility of 4 ($= 2 \times 1 + 2 \times 1$); higher than view S ’s ($1 \times 1 + 1 \times 1 + 1 \times 1.5 = 3.5$), and view P ’s ($2 \times 1.5 = 3$).

Scenario 3: To improve the poor performance seen by the VP tenant, the cluster

admin now decides to double the size of cache memory. Now two views will fit in the $2M$ -sized cache. However, even after this massive investment, the VP tenant will only see a minor increase in performance compared to the Analyst and Engineer tenants: views R and S will now be cached since they together have the highest weighted utility of 7.5 (4 for R + 3.5 for S); higher than 7 for R and P , and 6.5 for S and P .

Better scenarios: Let us consider what would have been an *ideal* cache allocation. An alternative in Scenarios 1 and 2 is to cache view S instead of R . While S has a slightly lower weighted utility, all three tenants will benefit from caching S . An alternative in Scenario 3 is to cache R and P which will also give performance benefits to all three tenants while only being slightly lower in overall weighted utility than caching R and S . In particular, the VP tenant will now see major benefits from doubling the cache size.

3.1.1 Contributions

- Our example shows key tradeoffs which create the need to make principled choices during memory allocation for caching in multi-tenant clusters. In this work, we develop cache allocation algorithms that can speed up a multi-tenant workload while guaranteeing *fairness* in terms of the tenants' performance.
- Section 3.3 considers shared resource allocation within a batch, and enumerates properties desired from any allocation scheme. We show that the notion of *core* from cooperative game theory captures the fairness properties in a succinct fashion. We show that when restricted to randomized allocation policies within a batch, a *proportionally fair* policy generates an allocation which satisfies fairness properties in expectation for that batch.
- The policies we construct are based on convex programming formulations of

exponential size. Nevertheless, in Section 3.4, we show that these policies admit to arbitrarily good approximations in polynomial time. We present implementations of two fair policies: max-min fairness and proportional fairness. We also present faster and more practical heuristics for computing these solutions.

- In Chapter 4, we provide ROBUS system details to optimize multi-tenant workloads in an *online* manner using the cache. ROBUS groups queries in small time-based batches and employs randomized cache allocation on each batch.
- We describe our implementation of ROBUS on a multi-tenant Spark cluster. Motivated by practical use cases, we develop a workload generator to create various scenarios. Implementation details and evaluation are provided in Chapter 4. Results show that our policies provide desirable throughput and fairness across a comprehensive spectrum of scenarios.

3.2 Related Work

The proportional fairness algorithm is widely studied in Economics [34–36] as well as in scheduling theory [37–43]. In the context of resource partitioning problems (or exchange economies) [44, 45], it is well-known that a convex program, called the Eisenberg-Gale convex program [35] computes prices that implement a Walrasian equilibrium (or market clearing solution). Our shared resource allocation problem is different from allocation problems where resources need to be partitioned, and it is not clear how to specify prices for resources (or views) in our setting. Nevertheless, we show that there is an exponential size convex program using configurations as variables for which solutions implement proportional fairness in a randomized sense.

In scheduling theory, the focus is on analyzing delay properties [39–41] assuming jobs have durations. Our focus is instead on utility maximization, which has also been considered in the context of wireless scheduling in [42, 43]. The latter work focuses

on *long-term* fairness for partitioned resources, where utility of a tenant is defined as sum of discounted utilities across time. The resulting algorithms, though simple, only provide guarantees assuming job arrivals are ergodic and tenants exist forever. They do not provide per-epoch guarantees. In contrast, we focus on obtaining per-epoch fairness in a randomized sense without ergodic assumptions, and on defining the right fairness concepts when resources are shared.

We finally note that [46] presents dynamic schemes for achieving envy-freeness across time; however, these techniques are specific to resource partitioning problems and do not directly apply to our shared resource setting.

3.3 Fairness Properties and Policies for Single Batch

We study various notions of fairness when restricted to view selection for queries from a single batch. We consider policies that compute allocations that simultaneously provide large utility to many tenants, and enforce a rigorous notion of fairness between the tenants. Since this is very related to other resource allocation problems in economics [36, 47, 48], we draw heavily on that work for inspiration. However, the key difference from standard resource allocation problems is that in our setting, the resources (or views) are simultaneously shared by tenants. In contrast, the resource allocation settings in economics have typically considered *partitioning* resources between tenants. As we shall see below, this leads to interesting differences in the notions of fairness.

3.3.1 Fairness and Randomization

It is well-known in economics [49] that the combination of fairness and indivisible resources (in our case, the cache and views) necessitates randomization. To develop intuition, we present two examples.

First consider a simple fair allocation scheme that for N tenants simply allows

each tenant to use $\frac{1}{N}$ of the total cache for her preferred view(s). It is plausible that some tenants prefer a large view that does not fit in this partition but does fit in the cache. Therefore, letting tenants have $\frac{1}{N}$ probability of using the whole cache can have arbitrarily larger expected utility than the scheme which with probability 1 lets them use $\frac{1}{N}$ fraction of the whole cache.

Next, consider a batch wherein two tenants each request a different large view such that only one can fit into the cache. In this case, there can be no deterministic allocation scheme that does not ignore one of the tenants. Using randomization, we can easily ensure that each tenant has the same utility in expectation. In fact, utility in expectation will be the per batch guarantee we seek, which over the long time horizon of a workload will lead to deterministic fairness.

Notation for Single Batch Since our view selection policy works on individual batches at a time, the notation and discussion below is specific to queries within a batch. Let N denote the total number of tenants. Define:

Definition 1. A configuration S is the set of views feasible in that the sum of the view sizes $\sum_{S_i \in S} |S_i|$ is at most the cache size.

$U_i(S)$ denotes the utility to tenant i that would result from caching S , which is defined as the sum over all queries in i 's queue of the utility for that query.

ROBUS generates a set Q of configurations which by definition can fit in the cache, and assigns a probability x_S to cache each configuration $S \in Q$. Define the vector of all such probabilities as:

Definition 2. An Allocation \mathbf{x} is the vector corresponding to probabilities x_S of choosing configuration S normalized so that $\sum_{S \in Q} x_S = 1$. We define the seminorm $\|\cdot\|$ on an allocation \mathbf{x} as $\|\mathbf{x}\| = \sum_{S \in (Q \setminus \emptyset)} x_S$.

In other words, $\|\mathbf{x}\|$ is the total quantity of probability mass that allocation \mathbf{x} places on “desireable” (nonempty) configurations. We denote $U_i(\mathbf{x}) = \sum_{S \in Q} x_S U_i(S)$ as the expected utility of tenant i in allocation \mathbf{x} . ROBUS implements allocation \mathbf{x} by sampling a configuration from the probability distribution.

For each tenant i , let $U_i^* = \max_S U_i(S)$ denote the maximum possible utility tenant i can obtain if it were the only tenant in the system. For allocation \mathbf{x} , we define the *scaled utility* of i as $V_i(\mathbf{x}) = \frac{U_i(\mathbf{x})}{U_i^*}$. We will use this concept crucially in defining our fairness notions.

3.3.2 Basic Fairness Desiderata

The first question to ask when designing a fair allocation algorithm is what properties define fairness. In classical economic theory, a *fair* allocation is one that is *Pareto-efficient* and *envy-free* [50]. Also, there has been much recent work in economics and computer science on heterogeneous resource allocation problems that consider these properties and the notion of *Sharing Incentive* [37, 38, 46]. Note that because we work within a randomized model, all of these properties are framed in terms of expected utility of tenants.

- **Pareto Efficiency (PE):** An allocation is Pareto-efficient if no other allocation simultaneously improves the expected utility of at least one tenant and does not decrease the expected utility of any tenant.
- **Sharing Incentive (SI):** This property is termed *individual rationality* in Economics. For N tenants, each tenant should expect higher utility in the shared allocation setting than she would expect from simply having access to all of the resources with probability $\frac{1}{N}$. More formally, allocation \mathbf{x} satisfies SI if for all allocations \mathbf{y} with $\|\mathbf{y}\| \leq \frac{1}{N}$ and for tenants i , $U_i(\mathbf{x}) \geq U_i(\mathbf{y})$. In other words, $V_i(\mathbf{x}) \geq \frac{1}{N}$ for all tenants i , where $V_i(\mathbf{x})$ is the scaled utility function defined above.

The above desiderata omit a formal definition of envy-freeness (informally, that no tenant should prefer the allocation to another tenant) which is something we revisit later. One property that is widely studied in other resource allocation contexts is strategy-proofness on the part of the tenants (the notion that no tenant should benefit from lying) [37, 51]. In our case, since the queries are seen by the query optimizer, strategy-proofness is not an issue.

We now consider a progression of view selection mechanisms on a single batch from very simple to more sophisticated. As a running example, suppose there is a cache of capacity 1. There are three views R , S , or P that are demanded by N tenants. Each view has unit size, so that we can cache only one view any time. Note that this is a drastically simplified example setup only intended to build intuition about why certain view selection algorithms might fail or are superior to others; our results and experiments do not only have unit views, are not limited to three tenants, and may have arbitrarily complex utilities compared to these examples.

We can summarize the input information our view selection might see in a given batch in a table (e.g., Table 3.2) where the numbers represent utilities tenants get from the views. An allocation here is a vector \mathbf{x} of three dimensions and $\|\mathbf{x}\| = 1$ that gives the probabilities in our randomized framework x_R, x_S, x_P for selecting the views.

Table 3.2: Every tenant gets utility from a different view

Tenant	R	S	P
A	1	0	0
B	0	1	0
C	0	0	1

Static Partitioning (STATIC) Static partitioning is the algorithm that deterministically allows each of the N tenants to use $\frac{1}{N}$ of the shared resource. This algorithm

does not take advantage of randomization. For the example in Table 3.2, this algorithm cannot cache anything because each user only gets to decide on the use of $\frac{1}{3}$ of the cache. The algorithm is thus trivially not Pareto efficient. As mentioned previously, such examples motivate the randomization framework to start with.

Random Serial Dictatorship (RSD) Random serial dictatorship (RSD) is widely considered [47, 48] for problems like house allocation and school choice. We order the tenants in a random permutation. Each tenant sequentially computes the best set of views to cache (in the residual cache space) to maximize its own utility. In the example in Table 3.2, each tenant gets a $\frac{1}{3}$ chance of picking her preferred resource so the allocation is $\langle x_R = \frac{1}{3}, x_S = \frac{1}{3}, x_P = \frac{1}{3} \rangle$, where each tenant has the same utility in expectation.

Table 3.3: Every tenant gets utility from the same view

Tenant	R	S	P
A	2	1	0
B	0	1	0
C	0	1	2

It is easy to prove that RSD is always SI: Each tenant has $\frac{1}{N}$ chance of being first in the random ordering, so its scaled utility is at least $\frac{1}{N}$. However, RSD fails to capture the shared nature of our problem. Consider Table 3.3; RSD computes the same allocation as in the example in Table 3.2, $\langle x_R = \frac{1}{3}, x_S = \frac{1}{3}, x_P = \frac{1}{3} \rangle$. On this example, although RSD is SI, it is not Pareto-efficient (PE). Tenants A and C have expected utility of 1 (a $\frac{1}{3}$ chance of getting 2 if they come first in the permutation and a $\frac{1}{3}$ chance of getting 1 if B does) and tenant B has expected utility of $\frac{1}{3}$ with this allocation. However, if we used allocation $\langle x_R = 0, x_S = 1, x_P = 0 \rangle$ then tenants A, B, and C all have utility 1, which is strictly better for tenant B and as good for tenants A and C. RSD fails to capture the fact that while each tenant may have

different top preferences, many tenants may share secondary preferences.

Utility Maximization Mechanism (OFTP) Next, consider the mechanism that simply maximizes the total expected utility of an allocation, *i.e.*, $\arg \max_{\mathbf{x}} \sum_i U_i(\mathbf{x})$. It is easy to check that this mechanism can ignore tenants who do not contribute enough to the overall utility. In other words, it cannot be SI.

Consider the example given in Table: 3.4. Here, the allocation which maximizes total utility is simply $\mathbf{x} = \langle x_R = 0, x_S = 1, x_P = 0 \rangle$ with expected total utility of 2. But tenant C has an expected utility of 0. Thus, performance optimization can give a Pareto optimal allocation but in general need not be fair in terms of sharing incentive because it only considers the total utility without consideration for individual tenants.

Table 3.4: Two tenants benefit from the same view and one benefits from a different view

Tenant	R	S	P
A	0	1	0
B	0	1	0
C	0	0	1

Max-min Fairness (MMF) In this algorithm we optimize performance subject to fairness constraints to get a mechanism that is both SI and PE. For allocation \mathbf{x} , let $\mathbf{v}(\mathbf{x}) = (V_1(\mathbf{x}), V_2(\mathbf{x}), \dots, V_N(\mathbf{x}))$ denote the vector of scaled utilities of the tenants. We choose an allocation \mathbf{x} so that the vector $\mathbf{v}(\mathbf{x})$ is lexicographically max-min fair. This means the smallest value in $\mathbf{v}(\mathbf{x})$ is as large as possible; subject to this, the next smallest value is as large as possible, and so on. We present algorithms to compute these allocations in Section 3.4.

Theorem 1. *The MMF mechanism is both PE and SI.*

Proof. The RSD mechanism guarantees scaled utility of at least $\frac{1}{N}$ to each tenant. Since the MMF allocation is lexicographically max-min, the minimum scaled utility it obtains is at least the minimum scaled utility in RSD, which is at least $\frac{1}{N}$. To show PE, note that if there were an allocation that yielded at least as large utility for all tenants, and strictly higher utility for one tenant, the new allocation would be lexicographically larger, contradicting the definition of MMF. \square

Table 3.5: All tenants except one get utility from the same view

Tenant	R	S
T_1	1	0
T_2	1	0
...
T_N	0	1

Consider the example in Table 3.5. It is easy to see that the MMF value is $\frac{1}{2}$ and can be achieved with the allocation $\langle x_R = \frac{1}{2}, x_S = \frac{1}{2} \rangle$. This allocation is both SI and PE.

3.3.3 *Envy-freeness and the Core*

SI and PE are necessary conditions for fair solutions, but are not sufficient. The above discussion omits the notion of *envy free*, meaning no tenant envies how the allocation treats another tenant. Defining envy free is straightforward in settings where resources are partitioned between tenants: No tenant should derive higher utility from the allocation to another tenant. However, in our setting, resources (views) are shared between tenants and thus envy freeness is ill-defined a priori. We want to reconsider envy in the public goods setting as a notion of proportionality and stability, a sort of group sharing-incentive. In order to develop the intuition behind this, we use an analogy to public projects.

The tenants are members of a society, who contribute equal amount of tax. The

total tax is the cache space. Each view is a public project whose cost is equal to its size. Users derive utility from the subset of projects built (or views cached). In a societal context, users are envious if they perceive an inordinate fraction of tax dollars being spent on making a small number of users happy. In other words, if they perceive a *bridge to nowhere* being built. Let us revisit the example in Table 3.5. Here, the MMF allocation sets $\mathbf{x} = \langle x_R = \frac{1}{2}, x_S = \frac{1}{2} \rangle$ and ignores the fact that an arbitrarily large number of tenants want R , compared to just one tenant who wants S . If we treat R as a school and S as a park, an arbitrarily large number of users want a school compared to a park, yet half the money is spent on the school, and half on the park. This will be perceived as unfair on a societal level, intuitively because the outcome is highly disproportional to the interests of the agents.

Randomized Core

In order to formalize this intuition, we borrow the notion of core from cooperative game theory and exchange market economics [44, 45, 52, 53]. We treat each user as bringing a *rate endowment* of $\frac{1}{N}$ to the system. If they were the only user in the system, we would produce an allocation \mathbf{x} with $\|\mathbf{x}\| = \frac{1}{N}$ and maximize their utility. An allocation \mathbf{x} over all tenants lies in the *core* if no subset of tenants can deviate and obtain better utilities for all participants by pooling together their rate endowments. More formally,

Definition 3. *An allocation \mathbf{x} is said to lie in the (randomized) **core** if for any subset T of N tenants, there is no feasible allocation \mathbf{y} such that $\|\mathbf{y}\| = \frac{|T|}{N}$, for which $U_i(\mathbf{y}) \geq U_i(\mathbf{x}), \forall i \in T$ and $U_j(\mathbf{y}) > U_j(\mathbf{x})$ for at least one $j \in T$.*

It is easy to check that any allocation in the core is both SI and PE, by considering sets T of size 1 and N respectively. In the above example (Table 3.5), the allocation $\mathbf{x} = \langle x_R = \frac{N-1}{N}, x_S = \frac{1}{N} \rangle$ lies in the core. Tenant T_N gets its SI amount of utility and cache space. The probability that the more demanded view R is cached is increased

by a proportionally larger amount. In societal terms, each user perceives his tax dollars as being spent fairly.

In the context of provisioning public goods, there are two solution concepts that are known to lie in the core: The first, termed a Lindahl equilibrium [54,55] attempts to find per-tenant prices that implement a Walrasian equilibrium, while the second, termed ratio equilibrium [56] attempts to find per-tenant ratios of cache-shares. However, these concepts are shown to exist using fixed-point theorems, which don't lend themselves to efficient algorithmic implementations. We sidestep this difficulty by using randomization to our advantage, and show that a simple mechanism finds an allocation in the core.

Proportional Fairness (PF)

Definition 4. An allocation \mathbf{x} is **proportionally fair (PF)** if it is a solution to:

$$\text{Maximize } \sum_{i=1}^N \log(U_i(\mathbf{x})) \text{ subject to: } \|\mathbf{x}\| \leq 1 \quad (3.1)$$

We show the following theorem using the KKT (Karush-Kuhn-Tucker) conditions [57]. The proof also follows easily from the classic first order optimality condition of PF [34]; however, we present the entire proof for completeness. Subsequently, in Section 3.4, we show how to compute this allocation efficiently.

Theorem 2. *Proportionally fair allocations satisfy the core property.*

Proof. Let \mathbf{x} denote the optimal solution to (PF). Let d denote the dual variable for the constraint $\|\mathbf{x}\| \leq 1$. By the KKT conditions, we have:

$$x_S > 0 \implies \sum_i \frac{U_i(S)}{U_i(\mathbf{x})} = d$$

$$x_S = 0 \implies \sum_i \frac{U_i(S)}{U_i(\mathbf{x})} \leq d$$

Multiplying the first set of identities by x_S and summing them, we have

$$d = d\left(\sum_S x_S\right) = \sum_i \frac{\sum_S x_S U_i(S)}{U_i(\mathbf{x})} = \sum_i 1 = N$$

This fixes the value of d . Next, consider a subset T of users, with $|T| = K$, along with some allocation \mathbf{y} with $\|\mathbf{y}\| = \frac{K}{N}$. First note that the KKT conditions implied:

$$\sum_i \frac{U_i(S)}{U_i(\mathbf{x})} \leq N \quad \forall S$$

Multiplying by y_S and summing, we have:

$$\sum_i \frac{U_i(\mathbf{y})}{U_i(\mathbf{x})} \leq N \sum_S y_S = K$$

Therefore,

$$\sum_{i \in T} \frac{U_i(\mathbf{y})}{U_i(\mathbf{x})} \leq K$$

Therefore, if $U_i(\mathbf{y}) > U_i(\mathbf{x})$ for some $i \in T$, then there exists $j \in T$ for which $U_j(\mathbf{y}) < U_j(\mathbf{x})$. This shows that no subset T can deviate to improve their utility, so that the (PF) allocation lies in the core. \square

3.3.4 Discussion

Our notion of core easily extends to tenants having weights. Suppose tenant i has weight λ_i . Then an allocation \mathbf{x} belongs to the core if for all subsets T of tenants, there does not exist \mathbf{y} with $\|\mathbf{y}\| = \frac{\sum_{i \in T} \lambda_i}{\sum_{i=1}^N \lambda_i}$ such that for all tenants $i \in T$, $U_i(\mathbf{x}) \leq U_i(\mathbf{y})$, and $U_j(\mathbf{x}) < U_j(\mathbf{y})$ for at least one $j \in T$. The proportional fairness algorithm is modified to maximize $\sum_i \lambda_i \log U_i(\mathbf{x})$ subject to $\|\mathbf{x}\| \leq 1$.

Utilities under MMF and PF

We present results showing that (PF) has larger utility than MMF in certain canonical scenarios. Our first scenario defines the following *grouped* instance: There are k views, $1, 2, \dots, k$ each of unit size. The cache also has size 1. There are k groups of tenants; group i has N_i tenants all of which want view i .

Lemma 1. *The total utility of (PF) is at least the total utility of MMF for any grouped instance.*

Proof. On grouped instances, MMF sets rate $1/k$ for each tenant, yielding a total utility of N/k for N tenants. The (PF) algorithm sets rate $x_i = N_i/N$ for all tenants in group i . This yields total utility of $\sum_i N_i^2/N$. Next note that

$$\frac{\sum_{i=1}^k N_i^2}{k} \geq \left(\frac{\sum_{i=1}^k N_i}{k} \right)^2$$

Noting that $\sum_i N_i = N$, it is now easy to verify that (PF) yields larger utility. \square

In fact, the ratio of the utilities of MMF and PF is precisely the Jain's index [58] of the vector $\langle N_1, N_2, \dots, N_k \rangle$. By setting $k = N/2 + 1$, and $N_2 = N_3 = \dots = N_k = 1$, this shows that (PF) can have $\Omega(N)$ times larger total utility than MMF. Our next scenario focuses on arbitrary instances with only two tenants.

Lemma 2. *For two tenants, the total utility of (PF) is at least the total utility of MMF.*

Proof. Let the utilities of the two tenants be a, b in (PF) and A, B in MMF. Assume $a \leq b$. Since MMF maximizes the minimum utility, we have $a \leq \min(A, B)$. Let $\alpha = A/a$ and $\beta = B/b$, so that $\alpha \geq 1$. Since $\log(a) + \log(b) = \log(ab)$ is maximized

by definition of PF and log is an increasing function, we have $ab \geq AB$, so $\alpha\beta \leq 1$. Since $\alpha \geq 1$, this implies $1/\beta \geq \alpha \geq 1$. Therefore

$$b - B = B(1/\beta - 1) \geq a(1/\beta - 1) \geq a(\alpha - 1) = A - a$$

This shows $a + b \geq A + B$ completing the proof. □

Summary of Fairness Properties In summary, Table 3.6 shows the fairness properties that hold for all of our candidate algorithms. We abbreviate the properties SI for sharing incentive and PE for Pareto efficiency. Based on this theoretical analysis, we suggest that proportional fairness is likely to be a preferable view selection algorithm for our ROBUS framework. The theoretical properties of proportional fairness suggest that it should perform fairly and efficiently.

Table 3.6: Fairness properties of cacheing mechanisms

Algorithm	SI	PE	CORE
STATIC	✓		
RSD	✓		
OPTP		✓	
MMF	✓	✓	
PF	✓	✓	✓

3.4 Approximately Computing PF and MMF Allocations

In this section, we show that the PF and MMF allocations can be computed to arbitrary precision. We then present fast heuristic algorithms for approximately computing PF and MMF allocations, which we implement in our prototype.

One key issue in computation is that the number of configurations is exponential in the number of views and tenants, so that the convex programming formulations have exponentially many variables. Nevertheless, since the programs have $O(N)$

constraints, we use the multiplicative weight method [59, 60] to solve them approximately in time polynomial in N and accuracy parameter $1/\epsilon$. These algorithms assume access to a *welfare maximization* subroutine that we term WELFARE.

Definition 5. *Given weight vector \mathbf{w} , WELFARE(\mathbf{w}) computes a configuration S that maximizes weighted scaled utilities, i.e., solves $\arg \max_S \sum_{i=1}^N w_i V_i(S)$.*

The scaled utilities are computed using the tenant utility model described in our evaluation section. In our presentation, we assume WELFARE solves the welfare maximization problem exactly. Our algorithms will make polynomially many calls to WELFARE.

Multiplicative Weight Method We first detail the multiplicative weight method, which will serve as a common subroutine to all our provably good algorithms. This classical framework [59, 60] uses a Lagrangian update to decide feasibility of linear constraints to arbitrary precision.

We first define the generic problem of deciding the feasibility of a set linear constraints: Given a convex set $P \in \mathbf{R}^s$, and an $r \times s$ matrix A ,

$$\boxed{\text{LP}(A, b, P): \exists x \in P \text{ such that } Ax \geq b?}$$

Let $y \geq 0$ be an r dimensional dual vector for the constraints $Ax \geq b$. We assume the existence of an efficient ORACLE of the form:

$$\boxed{\text{ORACLE } C(A, y) = \max\{y^t Az : z \in P\}}.$$

The ORACLE can be interpreted as follows: Suppose we take a linear combination of the rows of Ax , multiplying row $a_i x$ by y_i . Suppose we maximize this as a function of $x \in P$, and it turns out to be smaller than $y^T b$. Then, there is no feasible way to satisfy all constraints in $Ax \geq b$, since the feasible solution x would make

$y^T Ax \geq y^T b$. On the other hand, suppose we find a feasible x . Then, we check which constraints are violated by this x , and increase the dual multipliers y_i for these constraints. On the other hand, if a constraint is too slack, we decrease the dual multipliers. We iterate this process until either we find a y which proves $Ax \geq b$ is infeasible, or the process roughly converges.

More formally, we present the Arora-Hazan-Kale (AHK) procedure [59] for deciding the feasibility of $\text{LP}(A, b, P)$. The running time is quantified in terms of the WIDTH defined as:

$$\rho = \max_i \max_{x \in P} |a_i x - b_i|$$

Algorithm 1 AHK Algorithm

```

1: Let  $K \leftarrow \frac{4\rho^2 \log r}{\delta^2}$ ;  $y_1 = \vec{1}$ 
2: for  $t = 1$  to  $K$  do
3:   Find  $x_t$  using ORACLE  $C(A, y_t)$ .
4:   if  $C(A, y_t) < y_t^T b$  then
5:     Declare  $\text{LP}(A, b, P)$  infeasible and terminate.
6:   end if
7:   for  $i = 1$  to  $r$  do
8:      $M_{it} = a_i x_t - b_i$  ▷ Slack in constraint  $i$ .
9:      $y_{it+1} \leftarrow y_{it}(1 - \delta)^{M_{it}/\rho}$  if  $M_{it} \geq 0$ .
10:     $y_{it+1} \leftarrow y_{it}(1 + \delta)^{-M_{it}/\rho}$  if  $M_{it} < 0$ .
11:   ▷ Multiplicatively update  $y$ .
12:   end for
13:   Normalize  $y_{t+1}$  so that  $\|y_{t+1}\| = 1$ .
14: end for
15: Return  $x = \frac{1}{K} \sum_{t=1}^K x_t$ .
```

This procedure has the following guarantee [59]:

Theorem 3. *If $\text{LP}(A, b, P)$ is feasible, the AHK procedure never declares infeasibility, and the final x satisfies:*

$$(a_i x - b_i) + \delta \geq 0 \quad \forall i$$

3.4.1 Proportional Fairness

Our algorithm uses the AHK algorithm as a subroutine and considers dual weights to find an additive ϵ approximation solution. The primary result is the following

theorem:

Theorem 4. *An approximation algorithm computes an additive ϵ approximation to (PF) with $O(\frac{4N^4 \log^2 N}{\epsilon^2})$ calls to WELFARE, and polynomial additional running time.*

Proof For allocation \mathbf{x} , let $B(\mathbf{x}) = \sum_i \log V_i(\mathbf{x})$. Let $Q^* = \max_{\mathbf{x}} B(\mathbf{x})$ denote the optimal value of (PF), and let \mathbf{x}^* denote this optimal value. We first present a Lipschitz type condition, whose proof we omit from this version.

Lemma 3. *Let \mathbf{y} satisfy $B(\mathbf{y}) \geq Q^* - \epsilon$ for $\epsilon \in (0, 1/6)$. Then, for all i , $V_i(\mathbf{y}) \geq V_i(\mathbf{x})/2$.*

The proof idea is to use the concavity of the log function to exhibit a convex combination of \mathbf{x} and \mathbf{y} whose value exceeds Q^* , which is a contradiction. It is therefore sufficient to find Q^* to an additive approximation in order to achieve at least half the welfare of (PF) for all tenants. Towards this end, for a parameter Q , we write (PF) as a feasibility problem PFFEAS(Q) as follows:

Definition 6. PFFEAS(Q) *decides the feasibility of the constraints*

$$(F) = \left\{ \sum_S x_S V_i(S) - \gamma_i \geq 0 \forall i \right\}$$

subject to the constraints:

$$(P1) = \left\{ \sum_S x_S \leq 1, x_S \geq 0 \forall S \right\}$$

$$(P2) = \left\{ \sum_i \log \gamma_i \geq Q, \gamma_i \in [1/N, 1] \forall i \right\}$$

The above formulation is not an obvious one, and is related to *virtual welfare* approaches recently proposed in Bayesian mechanism design [61, 62]. The key idea is to connect expected values (utility) to their realizations in each configurations via expected value variables, the γ_i . The constraints (P2) and (P1) are over expected values, and realizations respectively. The ORACLE computation in the multiplicative weight procedure will decouple into optimizing expected value variables over (P2), and optimizing WELFARE over (P1) respectively, and both these problems will be easily solvable.

We note that (P2) has additional constraints $\gamma_i \in [1/N, 1] \forall i$. These are in order to reduce the width of the constraints (F). Note that otherwise, γ_i can take on unbounded values while still being feasible to (P2), and this makes the width of (F) unbounded. The lower bound of $1/N$ on γ_i is to control the approximation error introduced. We argue below that these constraints do not change our problem.

Lemma 4. *Let Q^* denote the optimal value of the proportional fair allocation (PF). Then, $\text{PFFEAS}(Q)$ is feasible if and only if $Q \leq Q^*$.*

Proof. In the formulation $\text{PFFEAS}(Q)$, the quantity γ_i is simply the scaled utility of tenant i . Consider the proportionally fair allocation \mathbf{x} . For this allocation, all scaled utilities lie in $[1/N, 1]$ since the allocation is SI. Therefore, \mathbf{x} is feasible for $\text{PFFEAS}(Q^*)$. On the other hand, if \mathbf{y} is feasible to $\text{PFFEAS}(Q)$ for $Q > Q^*$, then \mathbf{y} is also feasible for (PF), contradicting the optimality of \mathbf{x} . \square

We will therefore search for the largest Q for which $\text{PFFEAS}(Q)$ is feasible. Since each $\gamma_i \in [1/N, 1]$, we have $Q \in [-N \log N, 0]$. Therefore, obtaining an additive ϵ approximation to Q^* by binary search requires $O(\log N)$ evaluations of $\text{PFFEAS}(Q)$ for various Q , assuming constant $\epsilon > 0$.

Solving PFFEAS(Q). We now fix a value Q and apply the AHK procedure to decide the feasibility of PFFEAS(Q). To map to the description in the AHK procedure, we have $b = 0$, and A is the LHS of the constraints (F). We have $r = N$. Since any $V_i(S) \leq 1$, and $\gamma_i \in [1/N, 1]$, the width ρ of (F) is at most 1. Finally, for small constant $\epsilon > 0$, we will set $\delta = \frac{\epsilon}{N^2}$. Therefore, $K = \frac{4N^4 \log N}{\epsilon^2}$.

For dual weights \mathbf{w} , the oracle subproblem $C(A, \mathbf{w})$ is the following:

$$\text{Max}_{\mathbf{x}, \gamma} \sum_i (w_i V_i(\mathbf{x}) - \gamma_i)$$

subject to (P1) and (P2). This separates into two optimization problems.

The first sub-problem maximizes $\sum_i w_i V_i(\mathbf{x})$ subject to \mathbf{x} satisfying (P1). This is simply WELFARE(\mathbf{w}). The second sub-problem is the following:

$$\text{Minimize} \sum_i w_i \gamma_i$$

subject to \mathbf{w} satisfying (P2). Let L denote the dual multiplier to the constraint $\sum_i \log \gamma_i \geq Q$. Consider the Lagrangian problem:

$$\text{Minimize} \sum_i (w_i \gamma_i - L \log \gamma_i)$$

subject to $\gamma_i \in [1/N, 1]$ for all i . The optimal solution sets $\gamma_i(L) = \max(1/N, \min(1, L/w_i))$, which is a non-decreasing function of L . We check if $\sum_i \gamma_i(L) < Q$. If so, we increase L till we satisfy the constraint with equality. This parametric search takes polynomial time, and solves the second sub-problem.

The AHK procedure now gives the following guarantee: Either we declare PFFEAS(Q) is infeasible, or we find (\mathbf{x}, γ) such that for all i , we have:

$$\sum_S x_S V_i(S) \geq \gamma_i - \epsilon/N^2 \geq \gamma_i(1 - \epsilon/N)$$

Since $\sum_i \log \gamma_i \geq Q$, the above implies:

$$B(\mathbf{x}) = \sum_i \log V_i(\mathbf{x}) \geq Q - \sum_i \log(1 - \epsilon/N) \geq Q - \epsilon$$

so that the value $Q - \epsilon$ is achievable with the allocation \mathbf{x} .

Binary Search. To complete the analysis, since $\text{PFFEAS}(Q^*)$ is feasible, the procedure will never declare infeasibility when run with $Q = Q^*$, and will find an \mathbf{x} with $B(\mathbf{x}) \geq Q^* - \epsilon$, yielding an additive ϵ approximation. This binary search over Q takes $O(\log N)$ iterations.

Thus, we arrive at the result of theorem 4.

3.4.2 Max-min Fairness

We present an algorithm `SIMPLEMMF` that computes an allocation \mathbf{x} maximizing $\min_i V_i(\mathbf{x})$. The MMF allocation can be computed by applying this procedure iteratively as in [63]; we omit the simple details from this version. We note that the idea of applying the multiplicative weight method to compute max-min utility also appeared in [64].

We write the problem of deciding feasibility as `SIMPLEMMF`(λ):

$$(F) = \left\{ \sum_S V_i(S)x_S \geq \lambda \forall i \right\}$$

subject to the constraints:

$$(P) = \left\{ \sum_S x_S \leq 1, x_S \geq 0 \forall S \right\}$$

We have $\lambda^* \in [1/N, 1]$, where $\lambda^* = \max_{\mathbf{x}} \min_i V_i(\mathbf{x})$. Therefore, the width $\rho \leq 1$. Further, we can set $\delta = \epsilon/N$. We can now compute K from the AHK procedure, so

that $K = \frac{4N^2 \log N}{\epsilon^2}$ in order to approximate λ^* to a factor of $(1 - \epsilon)$. The procedure is described in Algorithm 2.

Algorithm 2 Approximation Algorithm for SIMPLEMMF

```

1: Let  $\epsilon$  denote a small constant  $< 1$ .
2:  $T \leftarrow \frac{4N^2 \log N}{\epsilon^2}$ 
3:  $\mathbf{w}_1 \leftarrow \frac{1}{N}$  ▷ Initial weights
4:  $\mathbf{x} \leftarrow \mathbf{0}$  ▷ Probability distribution over set of views
5: for  $k \in 1, 2, \dots, T$  do
6:   Let  $S$  be the solution to WELFARE( $\mathbf{w}_k$ ).
7:    $w_{i(k+1)} \leftarrow w_{ik} \exp(-\epsilon \frac{U_i(S)}{U_i^*})$ 
8:   Normalize  $\mathbf{w}_{k+1}$  so that  $\|\mathbf{w}_{k+1}\| = 1$ .
9:    $x_S \leftarrow x_S + \frac{1}{T}$  ▷ Add  $S$  to collection
10: end for

```

In order to compute MMF allocations, we use a similar idea to decide feasibility, except that we have to perform $O(N^2)$ invocations. This blows up the running time to $O\left(\frac{4N^4 \log N}{\epsilon^2}\right)$ invocations of WELFARE.

The algorithm gives the following result:

Theorem 5. *An approximation algorithm for SIMPLEMMF (Algorithm 2) finds a solution \mathbf{x} such that $\min_i V_i(\mathbf{x}) \geq \lambda^*(1 - \epsilon)$ using $\frac{4N^4 \log N}{\epsilon^2}$ calls to WELFARE.*

3.4.3 Fast Heuristics

In this section, we present heuristic algorithms that directly work with the exponential size convex programs. We directly implement these algorithms in software to gather our experimental results.

Configuration Pruning For $M = O(N^2)$, generate M random N -dimensional unit vectors $w_k, k = 1, 2, \dots, M$. For each w_k , let S_k be the configuration corresponding to WELFARE(w_k). Denote this set of configurations by \mathcal{S} . We restrict the convex programming formulations of PF and MMF to just the set of configurations \mathcal{S} , and solve these programs directly, as we describe below. The intuition behind doing this

pruning step is the following: The approximation algorithms for PF and MMF find convex combinations of configurations that are optimal for $\text{WELFARE}(w)$ for some w 's that are computed by the multiplicative weight procedure. Instead of this, we generate random such Pareto-optimal configurations, giving sufficient coverage so that each tenant has a high probability of having the maximum weight at least once.

We compared two algorithms for SIMPLEMMF, one using the multiplicative weight procedure (Algorithm 2), and the other solving the linear program (Program (3.3) below) restricted to random optimal configurations. When run on 200 batches with five tenants, using 5 weight vectors gives a 10.4% approximation to the objective of SIMPLEMMF. With 25 random weight vectors, the approximation error is 1.4%, and using 50 random weights, the approximation error drops to 0.6%. This shows that a small set \mathcal{S} of configurations that are optimal solutions to $\text{WELFARE}(w)$ for random vectors w is sufficient to generate good approximations to our convex programs. In our implementation, we set \mathcal{S} to be the union of these configurations along with the configurations generated by the SIMPLEMMF algorithm (Algorithm 2).

Proportional Fairness We first note that (PF) is equivalent to the following; the proof of equivalence follows from Theorem 2, where the dual variable corresponding to the constraint $\sum_{\mathcal{S}} x_{\mathcal{S}} = 1$ is precisely N .

$$\text{Max } g(\mathbf{x}) = \sum_{i=1}^N \log(V_i(\mathbf{x})) - N\|\mathbf{x}\| \quad \text{s.t.: } \mathbf{x} \geq 0 \quad (3.2)$$

Given a configuration space \mathcal{S} , we can solve the program (3.2) using gradient descent, as shown in Algorithm 3. As precomputation, for each configuration $S \in \mathcal{S}$, we precompute $V_i(S)$. Then $V_i(\mathbf{x}) = \sum_{S \in \mathcal{S}} V_i(S)x_S$.

Algorithm 3 Proportional Fairness Heuristic

- 1: Let $M = |\mathcal{S}|$. Set $t = 1$.
 - 2: Let $\mathbf{x}_1 = (1/M, 1/M, \dots, 1/M)$.
 - 3: **repeat**
 - 4: $\mathbf{y} = \nabla g(\mathbf{x})$ evaluated at $\mathbf{x} = \mathbf{x}_t$.
 - 5: $r^* = \arg \max_r (g(\mathbf{x}_t + r\mathbf{y}))$
 - 6: $\mathbf{x}_{t+1} = \mathbf{x}_t + r^*\mathbf{y}$
 - 7: Project \mathbf{x}_{t+1} as: $x_d = \max(x_d, 0)$ for all dimensions $d \in \{1, 2, \dots, M\}$.
 - 8: **until** \mathbf{x}_t converges
-

Max-min Fairness Using the precomputed configuration space \mathcal{S} , we solve SIMPLEMMF using the following linear program:

$$\max \left\{ \lambda \mid \sum_{S \in \mathcal{S}} V_i(S)x_S \geq \lambda \forall i, \mathbf{x} \geq \mathbf{0} \right\} \quad (3.3)$$

This can be solved using any off-the-shelf LP solver (our implementation uses the open source lpsolve package [65]). In order to compute the MMF allocation, we iteratively compute the lexicographically max-min allocation using the above LP. The details are standard; see for instance [63]. Briefly, in each iteration a value of λ is computed. All tenants whose rate cannot be increased beyond λ without decreasing the rate of another tenant are considered saturated and the rate of λ for these tenants is a constraint in the next iteration of the LP. The solution to the final LP for which all tenants are saturated is the MMF solution.

ROBUS Cache Planner

Chapter 3 built fair cache allocation solution theoretically including development of two algorithms. This chapter provides the architectural details of the system followed by detailed evaluation of a system prototype.

4.1 The ROBUS Platform

ROBUS (Random Optimized Batch Utility Sharing), shown in Figure 4.1, is the cache management platform we have developed for multi-tenant data-parallel workloads.

Multi-tenant cluster

ROBUS is designed to easily fit in modern data analytics systems like Hadoop and Spark. These systems support multi-tenancy. Each tenant submits its workload in an online fashion to a designated queue which is characterized by a weight indicating the tenant's *fair* share of system resources. A fair share scheduler [37,66] is typically employed to schedule tasks (smallest unit of work) for the workload.

Memory allocated to an application is divided into two parts: a heap space for run-time objects and a data cache. While the heap is divided across all concurrently running tasks and is allocated by the task scheduler (e.g., Hadoop fair scheduler [66],

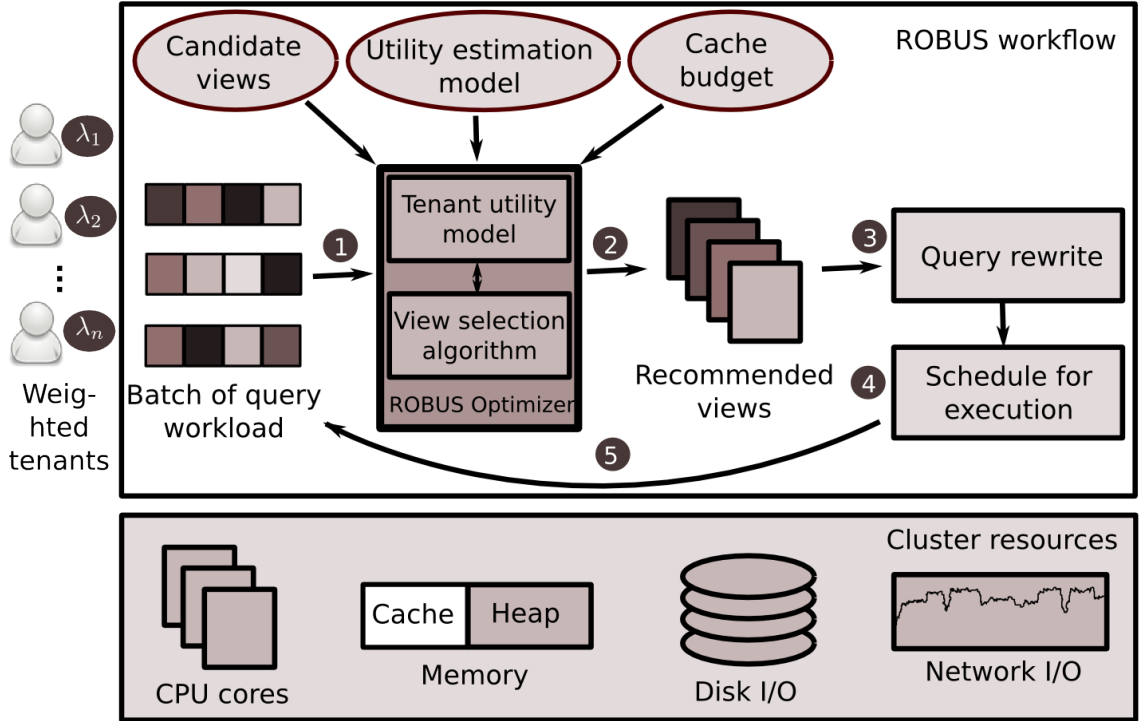


FIGURE 4.1: ROBUS platform

DRF [37]), the cache is shared by all queries in execution simultaneously and is managed by ROBUS. Referring back to Figure 3.1, ROBUS uses the Spark RDD cache architecture to implement the shared cache because of its superior performance.

ROBUS workflow

ROBUS is designed as a middleware between application tenants and the computation platform. Queries submitted by tenants to queues are processed by ROBUS in *batches* of a fixed time interval. Queries within a batch are optimized together and are scheduled for execution at the same time. This framework is motivated from prior multi-query optimization works such as Shared Scans [67] where a job/query is allowed to wait for a maximum time bounded by a preset *batch size* before it gets scheduled. ROBUS uses a similar notion where queries will never have to wait for more than the batch interval before being scheduled. Furthermore, ROBUS distinguishes between *interactive tenants* and *batch tenants*. Interactive tenants run short

queries for which they want to see immediate response. On the other hand, batch tenants are more throughput-sensitive. That is, they would like to see their workload finished as quickly as possible, and care less about how soon individual queries in the workload are finished. If a tenant declares itself as interactive to ROBUS, then ROBUS sets the batch interval to zero for that tenant. Thus, queries from this tenant will be scheduled immediately and will not be considered by ROBUS's optimization algorithm. We will discuss a more sophisticated batching model later in Section 4.3.4.

Figure 4.1 illustrates ROBUS workflow. ROBUS carries out the following four steps in a repeatedly-running loop. Step 1 removes a batch of queries from the tenants' queues. Step 2 runs an algorithm over this entire batch to select a set of views to cache from a predetermined set of candidate views. (The set of candidate views can be computed separately using any state-of-the-art algorithm for materialized view selection from a query workload [32, 33]; and this computation is not part of ROBUS's repeatedly-running loop.) This algorithm to select cached views from the candidate views simultaneously optimizes performance and fairness. Designing this algorithm is the main focus of this paper. Step 3 brings the views recommended in Step 2 to the cache (if they are not already in the cache) before rewriting queries to use the views. This step also directs the system to *uncache* any previously cached views not required for this batch. The entire batch of the rewritten queries is then submitted for execution in Step 4.

Life-cycle of a query in this workflow consists of the following three phases broadly:

- **Batching Delay:** Time from the query arrival to ROBUS submitting it for execution; Includes both the wait time before Step 1 as well as the time spent in ROBUS optimization.
- **Scheduling Delay:** The wait the query experiences in a scheduler queue

before the first task of it is launched.

- **Execution Time:** Time span from the first task start to the last task completion for the query.

The batch size configuration drives the trade-offs in Batching Delay and Execution Time. For example, a very low setting of batch size ensures a lower Batching Delay, but at the same time, lowers the chance of savings in Execution Time. This happens because a smaller batch results in many cache updates in order to get locally-optimal views for each batch into the cache; thereby making most of the queries disk-bound. We evaluate these tradeoffs and also give some guidelines in tuning the batch size in Section 4.3.

Views and Utilities

Step 2 takes three inputs: (i) a set of candidate views, (ii) a utility estimation model for cached views, and (iii) total cache budget (i.e., memory available for caching). The candidate view generation in ROBUS is a pluggable module. By default, the candidate views for a SQL query are the base tables accessed by the query. The ROBUS prototype we evaluate in Section 4.3 uses a predetermined set of candidate views consisting of different vertical projections of the input tables. This set can be made richer using the pluggability of the module. For workloads like machine learning and graph processing, the candidate views are the datasets on which the user has put a cache directive. A view management algorithm from the database literature [32,33] can be employed to dynamically build and maintain a set of candidate views on disk to suit the workload. Such an algorithm, if employed, would run in the background. ROBUS also supports a pluggable Step 3 where a query is rewritten to use the views selected for caching in Step 2. In future, we plan to extend Step 3 to support re-optimization of the query based on the cached views, which could change the query plan entirely.

ROBUS view selection module employs a pluggable utility estimation model to estimate the utility provided to a query by any cached view. ROBUS makes polynomial (in the number of tenants) calls to this model while optimizing a batch. ROBUS currently models these utilities as savings in disk I/O costs if the view were to be read off of the in-memory cache versus disk. This approach keeps the models simple and widely applicable. It could be extended to richer utility models required to support more complex candidate view selection algorithms from literature that consider interactions among views [68]. Total utility of a cache configuration for a tenant is computed by adding up estimated utilities of the queries submitted by the tenant. The tenant utilities thus computed are used in recommending an optimal set of views to cache. The view selection policy attempts to come up with a cache configuration that provides a weight-proportional performance speedup to the tenants' workload. The guarantees provided are in expectation, i.e., over a number of batches. This is due to the fact that a cache directive is specified over an entire view. In other words, a view is treated indivisible at the program interface level. While individual partitions of a view on a node are managed by a standard LRU cache manager by the system underneath, ROBUS operates at a higher level of data abstraction.

4.2 Related Work

Multi-tenant database-as-a-service architectures

Traditionally, the notion of multi-tenancy in databases deals with consolidating DBMS resources, namely, hardware, process, schema, among tenants [69–72]. Commercial database engines use adaptive memory managers to split server memory dynamically among various pools including the buffer pool [73, 74]. Buffer pool is used to cache recently accessed data pages for performance gains. However, the model of multi-tenancy used here supports exclusive ownership of data by tenants and therefore, no consideration is given to a possibility of data sharing among ten-

ants. Emerging multi-tenant big data architectures, on the other hand, allow for the entire cluster data to be shared among tenants representing different teams within an enterprise [75, 76]. This sharing of data is exploited by ROBUS in making the best use of cache.

A recent work on buffer pool management proposes a multi-tenancy aware buffer pool page replacement algorithm [77]. Techniques developed here could potentially apply to big data analytics setups, e.g., Spark’s RDD store can be treated as a pool of data partitions/blocks. ROBUS, though, manages cache at the application layer because: 1. Newer programming interfaces like Spark allow cache directives to be specified only on the objects/views accessible through programs, 2. Cache “all-or-nothing” property observed in data-parallel workloads [29] calls for decision-making on an entire view, especially for iterative machine learning workloads, and 3. While hit ratio metering required in [77] is supported in mature database-as-a-service installations, it would require big infrastructure upgrades in an emerging data analytics framework like Spark.

Infrastructure-as-a-Service solutions also allow sharing memory across tenants [78] and solutions are available to automatically change the memory allocation to tenants, e.g. ballooning [79].

Multi-tenant big data analytics architectures

Big data analytics platforms support a model of multi-tenancy where large volumes of data is accessed by analysts, data scientists, and developers alike for functionalities ranging from periodic report generation to business intelligence [15, 80]. A critical component of such platforms is a fair scheduler or a resource allocator [37, 66, 81]. These schedulers do not differentiate the cache resource from the heap resource and, as a result, partition cache among tenants. As seen in our work, partitioned/isolated cache setups severely reduce optimization opportunities.

Table 4.1: Summary of cache management policies in data analytics clusters

Policy	Frequency	Data abstraction	Multi-tenant	Multi-tenancy properties			
				Pareto efficient	Strategy free	Fair shares	Fair speedups
Isolated caches(LRU)	Online	Block	✗	-	-	-	-
MT-LRU [77]	Online	Block	✓	✓	does not apply	✓	✗
Shared offline [85]	Offline	File/MV/RDD	✓	✗	does not apply	✗	✗
Greedy online(LRU)	Online	File/MV/RDD	✓	✓	✗	✗	✗
Fair Ride [51]	Online	File	✓	near-optimal	✓	✓	✗
ROBUS	Batched	File/MV/RDD	✓	✓	does not apply	near-optimal	near-optimal

Most data analytics platforms support data objects to be cached either in application memory or in an external caching service (e.g. Alluxio [28] and DDM [82]). ROBUS optimizer fits naturally in both scenarios: While data sharing comes naturally with the use of external cache store, the application memory is also shared among multiple jobs submitted through a job server [75, 76].

Memory-based analytics infrastructures such as SAP Hana [27], RAMClouds [83] allow fast processing of memory-resident data. However, many commercial clusters today have a considerably large amount of storage on disk than total memory (e.g., Facebook [84]). Choice of cache directives is critical for performance and fairness in such setups and ROBUS is designed to provide an automated solution to this problem.

Table 4.1 summarizes how ROBUS compares to existing caching policies. LRU policies typically operate at a lower granularity of data on individual nodes. Among the policies used to manage data at a higher granularity, i.e., a file, a materialized view, or an RDD, an offline policy could be used to materialize views that are expected to maximize benefits with some prior knowledge on data access [85]. This policy, however, does not provide any fairness guarantees. On the other extreme, an online policy could greedily materialize every cache directive using LRU for evictions which results in performance inefficiencies when tenants have big variability in data access. Additionally, this policy is not strategy proof as pointed out in FairRide [51] with a phenomenon of ‘free-ride’: a tenant getting *free* access to a data object brought in cache by another tenant. FairRide brings in strategy-proofness to this setup by

probabilistically blocking access to cached files. However, FairRide does not give any guarantees on expected speedup to tenants, which is an important metric from fairness standpoint. Furthermore, it operates on files cached in an external store unlike ROBUS, which supports materialization of any view of data in the application memory as well. As ROBUS operates on batches of workload already submitted by tenants, the question of strategic tenants does not arise. Finally, ROBUS supports multiple algorithms to optimize a workload batch which allow for different trade-offs in the performance and the fairness.

Physical design tuning and Multi-query optimization

Classical view materialization algorithms in databases [30–32, 86, 87] treat entire workload as a set and optimize towards one or more of the flow time, space budget, and view maintenance costs. Online physical design tuning approaches [33, 68, 88, 89], on the other hand, adapt to changes in query workload by modifying the physical design. None of the afore-mentioned approaches support multi-tenant workloads and therefore cannot be used in selecting views for caching. However, some of the techniques used, in particular, candidate view enumeration, view matching, and query rewrite, can be applied in ROBUS through the pluggability provided in the framework (Section 4.1).

Batched optimization of queries was proposed in [90] and is used in many work sharing approaches [67, 91–93]. ROBUS employs batched query optimization likewise, but crucially also ensures that each tenant gets their fair share of utility from the cache.

4.3 Evaluation

ROBUS is motivated by emerging multi-tenant architectures wherein memory is treated as a first class citizen. The primary goal of evaluation is to test different cache

allocation policies on a variety of practical setups of multi-tenant analytics clusters. We evaluate ROBUS caching policies on a variety of practical setups of multi-tenant analytics clusters. The setups may differ in the number of tenants, workload arrival patterns, data access patterns, etc. Some of the practical multi-tenant setups listed below.

- Analysts: Tenants correspond to various BI analysts in an enterprise that all run a similar workload. A few datasets are frequently accessed by all tenants suggesting a good opportunity for shared optimization.
- ETL+Analysts: All analysts have similar data access patterns as above. But additionally, a tenant runs ETL workload that may touch different datasets.
- Workflow+Engineering: Engineering workload is of bursty nature. Depending on the time of day, engineering queues have small bursts of work whereas workflow queues have a pre-scheduled well-balanced workload.

We replicate various combinations of these setups on a small-scale Spark cluster detailed next.

4.3.1 Setup and Methodology

Figure 4.1 has presented the architecture of ROBUS. We use Apache Spark [15] to build a system prototype. Spark is a natural choice for the evaluation since it supports distributed memory-based abstraction in the form of Resilient Distributed Datasets (RDDs). A long-running Spark context is shared among multiple queues, each queue corresponding to a tenant. The Spark context has access to the entire RDD cache in the cluster. Spark’s internal fair share scheduler is configured with a pool dedicated to each queue; the fair share properties of the pool are set proportional to the weight of the corresponding queue.

Table 4.2 presents our test cluster setup. We generate two types of data: (a) A

Table 4.2: Test cluster setup

Spark version	1.6.1
Number of worker nodes	10
Total number of cores	80
Total executor memory	100GB
Memory used for storage (Cache)	10GB

set of 30 datasets with varying sizes each matching schema of the “sales” tables—*store_sales*, *catalog_sales*, and *web_sales*—from TPC-DS benchmark [94], and (b) All datasets from TPC-H benchmark [95] generated at scale 10.

The first category of data represents raw fact/log data that comes into the cluster from the OLTP/operational databases in a company. This data is processed by synthetically-generated ETL and exploratory SQL queries, each performing scans and aggregations over a dataset. We refer to this category of queries as the **Sales** workload. The total size of **Sales** data on disk is 600GB. We treat a vertical projection of each dataset consisting of the most popularly accessed columns as a candidate view. Sizes of these views when loaded to cache range from 118MB to 3.6GB, as shown in Figure 4.2.

The second category of data, TPC-H, represents a typical warehouse. This data is queried by standard TPC-H benchmark queries which involve more complex operations, such as joins, compared to the **Sales** workload. Like in the case of the **Sales** workload, we support materialization of a vertical projection on each dataset. Sizes of the cached views range from 10MB to 3GB.

We set the cache size to 10GB, 10% of the total executor memory, leaving aside the rest as the heap space for execution objects.

The tenant utility model we use to estimate the utility of a cache configuration is a simple approximation to the I/O savings. If a view in a cache configuration can answer a query, the utility for the query is set equal to the size of the dataset on disk from which the view is created. The utility for a batch of queries is the sum of utilities

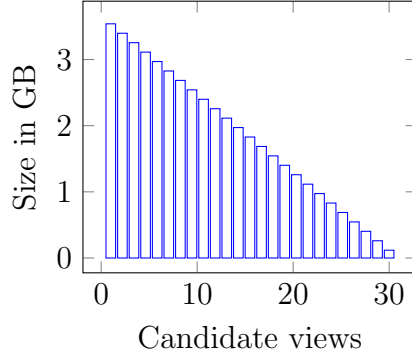


FIGURE 4.2: Cache size estimates of candidate views in Sales workload

for each query within the batch minus the cost of building the configuration. The dataset sizes are stored in a hashtable to speed up the utility evaluation. Note that, we do not take into account any possible filtering in input data due to optimizations such as filter pushdowns in our utility estimation.

ROBUS workflow is described in Section 4.1 already. The cache update phase in our prototype first *uncaches* any views not required for the current batch and then materializes each newly selected view in the configuration by running a *count(*)* query over it. While the materialization queries are running, we also submit the queries from the batch which are not dependent on any view being materialized in order to make full use of resources. Here we want to add the fact that the cache update phase in our evaluation setup only marks datasets for caching or uncaching using Spark’s cache directives. Spark lazily updates the cache when the first query requesting cached data from the batch is scheduled for execution.

Workload Arrival and Data Access

Figure 4.3 shows our workload generation process. Several studies have established that query arrival times follow a Poisson distribution [96, 97]. We use the same in our prototype. Previous studies have also indicated that the data accessed by analytical workloads follows a Zipf distribution [8, 96]: A small number of datasets are more popular than others, while there is a long tail of datasets that are only sporadi-

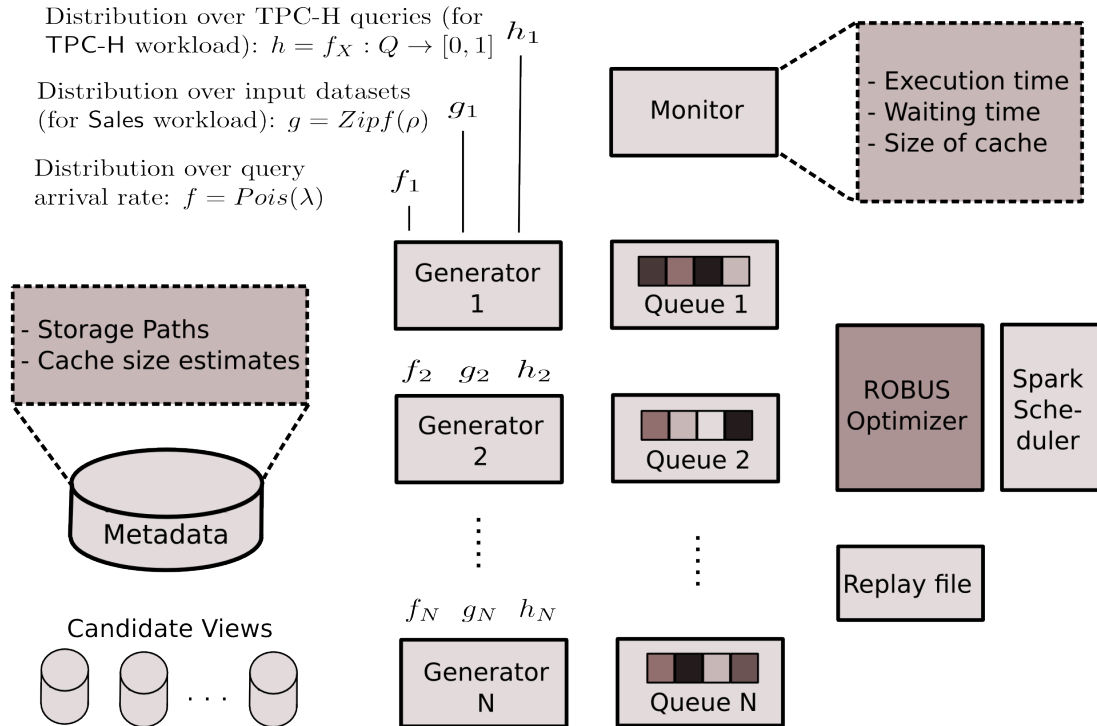


FIGURE 4.3: Workload generation in ROBUS

cally accessed. To replicate such data access, our synthetic Sales workload generator picks a dataset from a Zipfian distribution provided at the time of configuration and adds grouping and aggregation predicates from a probability distribution over a set of predicates defined over the datasets at the time of configuration. The TPC-H workload generator, on the other hand, picks a benchmark query from a probability distribution over the 22 benchmark queries provided at the time of configuration. Queries used in the evaluation are all submitted using SparkSQL APIs.

4.3.2 Performance gains due to batching

We first analyze the performance obtained on different granularity of data caching. We compare three policies here:

1. **BASELINE:** The case where no view is ever cached.
2. **GREEDY ONLINE:** Each query greedily materializes a view. The LRU policy at

Throughput (Queries served per minute)

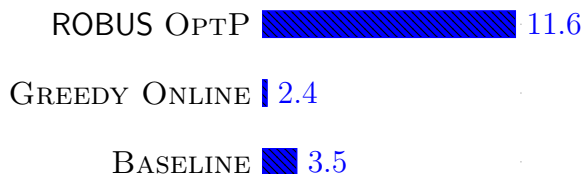


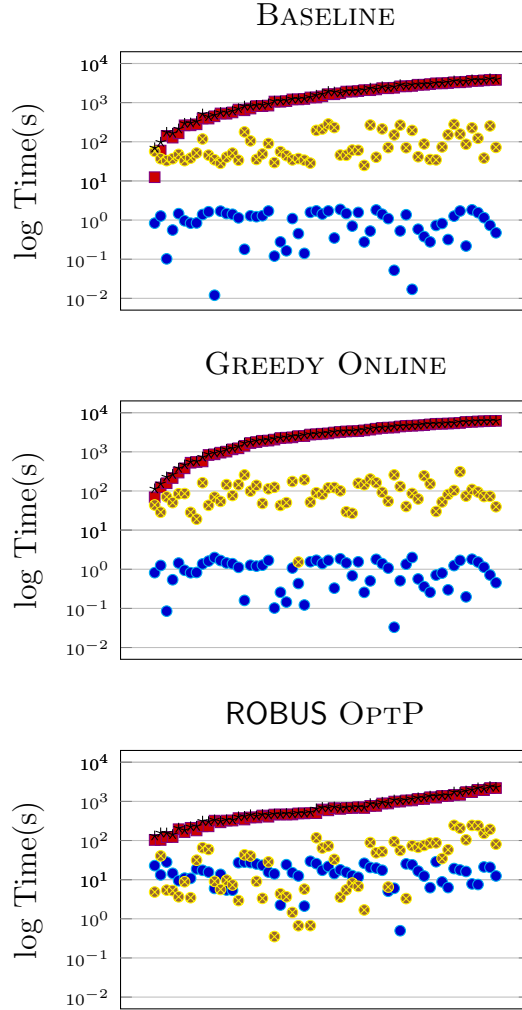
FIGURE 4.4: Workload throughput on different cache granularities

data partition level takes care of cache management.

3. ROBUS OTP: Queries in a fixed time batch are analyzed to materialize a *configuration* (refer Definition 1) that provides maximum utility to the query batch.

We use a setup with four tenants generating queries with mean inter-arrival time of 10 seconds each. While three of the tenants generate queries over TPC-H data, the fourth tenant is configured with **Sales** workload generator. The query generators are kept active for 600 seconds while the ROBUS OTP policy is configured to use 40 second batches. Figure 4.4 compares the policies based on the overall throughput. Figure 4.5 plots for all queries from a tenant the end-to-end (E2E) latency and the time breakup of the three phases of query life-cycle (as defined in Section 4.1).

Surprisingly, GREEDY ONLINE caching fares worse than BASELINE. Due to a large variability in the data accessed by tenants' workload, greedily caching views leads in too many cache transitions making most queries I/O bound. Furthermore, the extra overhead in running cache queries results in a longer Scheduling Delay as can be seen in Figure 4.5(b). While the batched caching approach used in ROBUS causes an increase in Batching Delay by a few seconds (capped by 40s in this case), the execution times, as seen in the graph, drop by one or two orders of magnitude for most of the queries. This triggers a significant drop in the Scheduling Delay as well. The overall throughput gain, as seen in Figure 4.4, is over 3x.



• Batching Delay ■ Scheduling Delay ♦ Execution Time * E2E Latency

FIGURE 4.5: Comparing cache policies in terms of E2E latency of queries including the breakup across the three phases of query lifetime

4.3.3 Evaluation of batched caching policies

The batched caching policy evaluated above is designed with a goal of utility maximization (OPTP) and is unfair in theory (Section 3.3). Here we compare it with max-min fair and proportional fair policies, and also with a commonly used static partitioning.

- **STATIC:** Cache is partitioned in proportion to weights of the tenants with each partition optimized individually.
- **MMF:** Max-min fairness implementation described in Section 3.4.
- **FASTPF:** Proportional fairness implementation described in Section 3.4.
- **OPTP:** Workload from a batch is treated as belonging to a single tenant – a special case of either MMF or FASTPF.

In order to compare these policies, we vary the following parameters independently in our experiments.

1. Data sharing among tenants
2. Workload arrival rate
3. Number of tenants
4. Batch size and Cache state

We first list the performance metrics we gathered.

1. Throughput Gain. Throughput is simply the number of queries served per unit time. The *throughput gain* for a cache setup corresponds to the ratio of the throughput of the setup to the throughput of BASELINE, wherein no cache is used, under the same workload.

2. Fairness Index. For job schedulers, a performance-based fairness index is defined in terms of variance in slowdowns of jobs in a shared cluster compared to a baseline case where every job receives all the resources [81]. As our work is about speeding up queries, we use relative speedups across queries while deriving fairness. The baseline again is the case of not using the cache. Here, X_i is the mean speedup for tenant i , and λ_i is the weight of tenant i .

$$\text{Fairness index} = \frac{(\sum_{i=1}^n \frac{X_i}{\lambda_i})^2}{n \sum_{i=1}^n (\frac{X_i}{\lambda_i})^2} \quad (4.1)$$

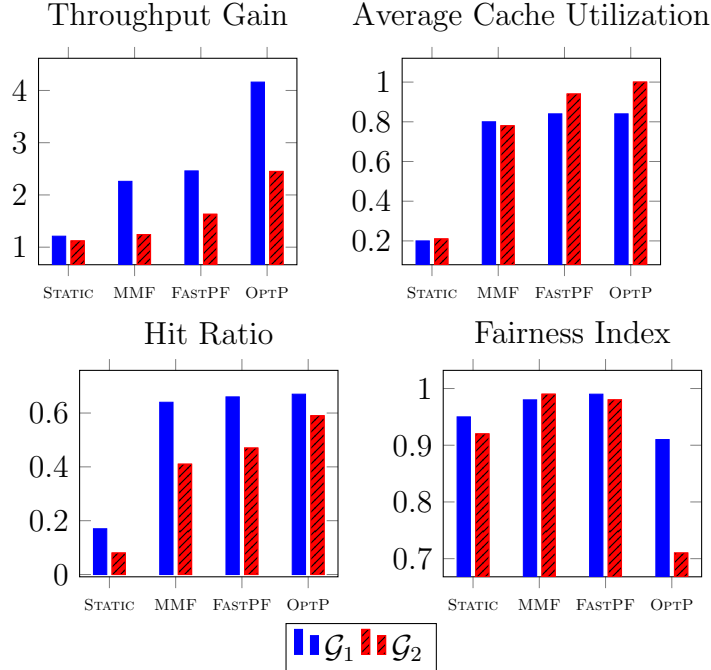


FIGURE 4.6: Effect of data sharing changes on four equi-paced tenants

3. Average Cache Utilization. This is simply the average fraction of cache in use during workload execution.

4. Hit Ratio. The fraction of queries served off cached views in a workload.

We emphasize that these metrics are computed over long time horizons. Some of the other metrics we collect include flow time, mean execution time, mean wait time, and wait time fairness index. These, however, are not presented due to space constraints.

Effect of data sharing among tenants: To study the impact of different data sharing patterns on the performance of algorithms, we create two different setups with four tenants. In setup \mathcal{G}_1 , all four tenants generate queries from a single probability distribution over Sales datasets. Setup \mathcal{G}_2 , however, showcases a big heterogeneity among the tenants: Two of the tenants generate queries over TPC-H data; The other two generate Sales queries with datasets picked from different probability distribu-

tions. Each tenant generates queries with mean query inter-arrival time set to 10 seconds. Batch size is set to 30 seconds. We run 30 batches of workload for every data point.

Figure 4.6 shows how algorithms perform on the two setups. The homogeneous setup shows a throughput gain upwards of 2 on all algorithms except STATIC. OPTP, in particular, gives a high throughput by always caching views on the most frequently accessed datasets. All algorithms, including OPTP, score high on the fairness index as well. The heterogeneous setup distinguishes the algorithms better. Firstly, throughput goes down with heterogeneity in all cases. The gap in throughput of STATIC with the shared cache policies is small despite the shared policies caching significantly more data. This is due to the frequent updates to cache configuration per batch in this setup. MMF, in particular, showcases very small gain in throughput. We revisit this phenomenon later in Section 4.3.4. OPTP scores poorly on fairness index in \mathcal{G}_2 : It gives preferential treatment to the two TPC-H tenants due to data sharing between them while denying cache space to the other two tenants. Both MMF and FASTPF get an almost perfect fairness score irrespective of data sharing pattern among tenants.

Effect of the variance in query arrival rates: To replicate the bursty tenants scenarios, we vary query inter-arrival rates of tenants in a two-tenant setup. We create two setups, \mathcal{G}_3 and \mathcal{G}_4 , as described in Table 4.3. While the first setup leads to a roughly equal number of queries from each tenant in a batch, the second setup generates three queries of tenant 2 for every query of tenant 1 on average. Here, the first tenant is configured to generate TPC-H workload while the second one generates Sales workload.

Figure 4.7 shows the impact of variance in query arrival. Throughputs are much better in setup \mathcal{G}_4 because of a greater proportion of queries over Sales workload:

Table 4.3: Query inter-arrival rates for different setups

Setup	\mathcal{G}_3	\mathcal{G}_4
Number of tenants	2	2
Batch size (s)	36	36
Mean query inter-arrival rates (s)	(6, 6)	(12, 4)

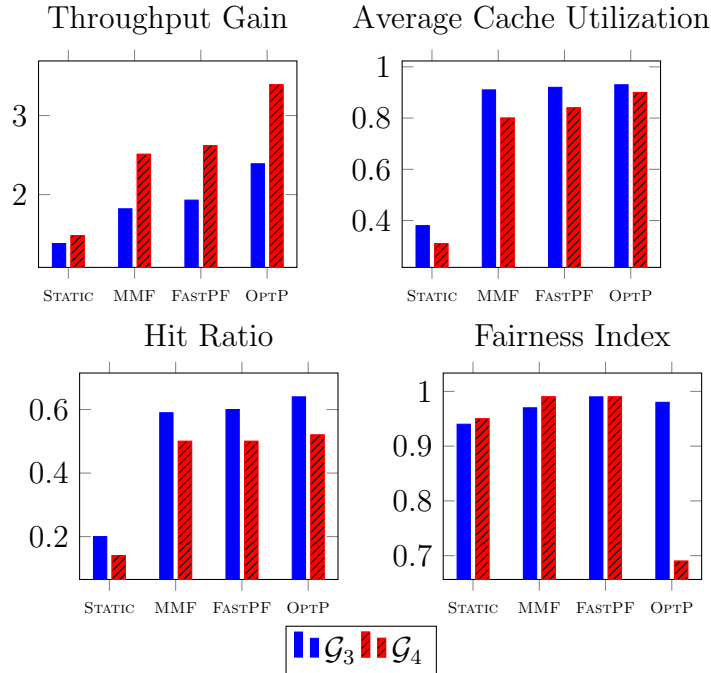


FIGURE 4.7: Effect of the variance in query arrival rates

These queries are all scan bound and get big speedups due to caching compared to some complex TPC-H queries that are not as heavily dependent on scans.

The average cache utilization and hit ratio, however, go down in the second setup. This, too, is due to the nature of the workload wherein the most popular views for the Sales workload submitted by tenant 2 are smaller in size compared to the most popular views for the TPC-H workload. OPTP favors the faster tenant in the second setup *unfairly*: It serves 8x queries off cache for the faster tenant compared to the queries served off cache for the slower tenant. Both MMF and FASTPF serve only about 3x queries off cache for the faster tenant which is in proportion to the number of queries submitted.

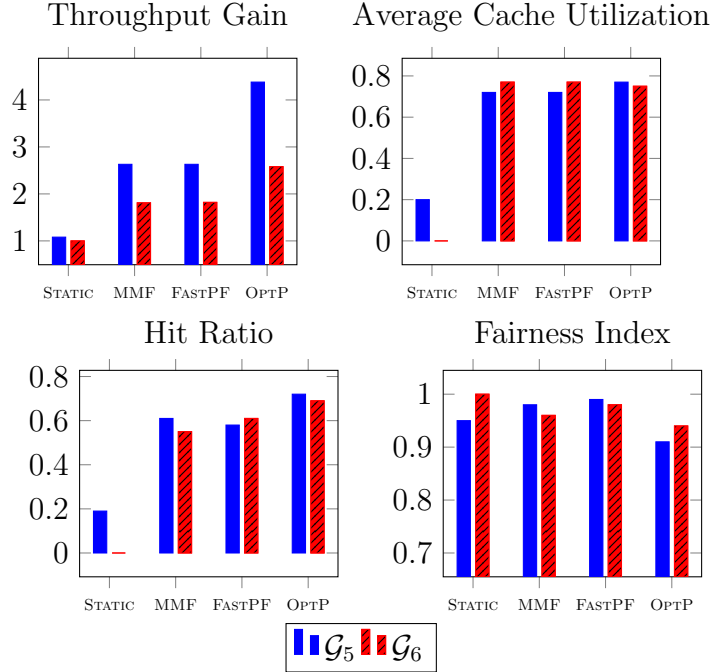


FIGURE 4.8: Effect of changing number of tenants

Effect of the number of tenants: We present results on two setups: \mathcal{G}_5 with four tenants and \mathcal{G}_6 with 8 tenants. Each tenant generates `Sales` queries from a single probability distribution over datasets under both the setups. We try to keep the number of queries per batch the same by doubling the query inter-arrival rate with the number of tenants, batch size remaining the same. 30 batches of workload are run in each case. Figure 4.8 shows the results.

As the number of tenants goes up, both the average cache utilization and the hit ratio under `STATIC` drop sharply, whereas the average cache utilization of the other algorithms remains largely stable. The fairness index stays high for all algorithms in both setups since the tenants show a great deal of homogeneity in data access. Throughput gains by `MMF` and `FASTPF` are about 2x over those by `STATIC`, while `OPTP` improves it further by only about 30%.

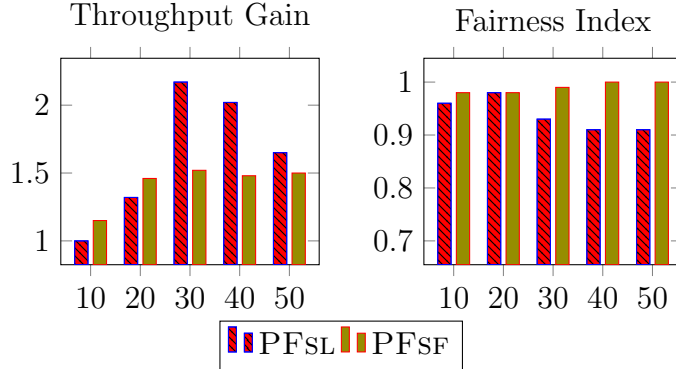


FIGURE 4.9: Effect of batch size on four equi-paced tenants setup

Batch Size and Cache State

Our batched policies introduce additional optimization choices. Primarily, there are two tunable parameters in a view selection algorithm: 1. Batch size, and 2. State of cache across batches.

The first option needs no elaboration. The second option decides whether the cache is to be treated as *stateful* or as *stateless* when optimizing a batch. In the former case, the estimated benefit of views already in cache is boosted by a factor $\gamma > 1$ to make it more likely for these views to stay in cache. The latter case ignores the state of the cache when considering the next batch. All results presented so far have used stateless cache.

Table 4.4: Query time statistics on stateless cache optimizations

Batch size	0(BASELINE)	10	20	30	40	50
Mean Batching Delay	0s	5s	10s	15s	20s	25s
Mean Execution Time	83s	64s	67s	44s	48s	56s

We empirically compared how the algorithms react to these parameters. Figure 4.9 shows the effect of a change in batch size on two versions of FASTPF: One treating the cache as stateless(PFSL), and the other treating it as stateful(PFSF), with $\gamma = 2$. The two versions are compared on a four tenant setup. Mean Batching Delay and Mean Execution Time for the stateless setups are included in Table 4.4.

The best performance is seen on the stateless cache with batch sizes of 30 and 40 seconds. With queries arriving at a mean inter-arrival time of 10 seconds, each batch contains 10-18 queries here. Upon inspecting the cache configurations across batches, it is noticed that the cache fits 5-6 views on average. Each of the cached views, in most cases, is observed to serve at least two queries thus amortizing the cost of data loading well. Smaller batch sizes (10s and 20s) result in too many cache transitions adversely affecting performance speedups. Stateful cache optimizations help here by retaining the more popular views but only by a small margin. Larger batch sizes (50s and up) also show lower performance gains because they allow a relatively smaller fraction of queries the cache benefits, the cache budget remaining the same.

PFSL setups are observed to score low on fairness index in comparison to PFSF. This is due to an implementation choice we made: Do not evict a view requested for a new batch if it is already cached. While the stateful optimization accounts for such views during utility estimation, the stateless optimization does not. This creates a small gap in the estimated utilities and the actual performance speedups for tenants in the stateless optimization which is reflected in the observed fairness index.

4.3.4 Discussion

Comparing caching policies

Our experiments show that among all setups, FASTPF policy shows the most consistent numbers both in terms of throughput and fairness. However, in many cases, MMF is also observed to do just as well. We believe this is a second order difference that a more precise cost model and implementations of the exact algorithms (for instance, the algorithm in Section 3.4 for proportional fairness) will bring out. However, even given similar empirical results, PF has the advantage of the core property as a succinct and easy to explain notion of fairness and should, therefore, be used as the default policy. The following example of a four tenant setup further illustrates

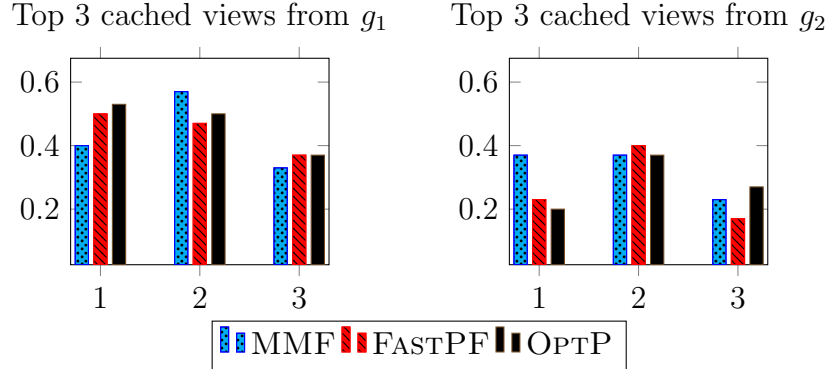


FIGURE 4.10: Fraction of time popular views cached by algorithms

the difference among the policies.

We set the query generator for three tenants to a distribution g_1 over Sales datasets while the fourth tenant is made to generate queries from a different distribution g_2 . Recollecting the example presented in Table 3.5, MMF tries to share the cache (probabilistically) equally between the two sets of tenants, producing an inefficient allocation outside of the core. We include a chart showing the duration for which the most popular views were cached by MMF, FASTPF, and OPTP in Figure 4.10. The top three views in each of g_1 and g_2 serve 25%, 13%, and 8% of the queries respectively. While MMF caches the topmost view from the distributions roughly equally, FASTPF and OPTP favor the topmost view from g_1 more since it is shared by three tenants. MMF tries to compensate the three tenants by caching their second best view more, but this view has a lower utility both due to lower access frequency and smaller size. So MMF ends up favoring the fourth tenant. OPTP favors top views from g_1 unreasonably more thus being unfair. FASTPF provides more favorable tradeoffs here.

We next note that the **running time** of our algorithms is polynomial in number of tenants. In most typical industry setups, there is only a handful number of tenants. Therefore, we expect our algorithms to be fast even in the wild. Just to quantify the batch processing times, we observed them to be the order of tens of milliseconds

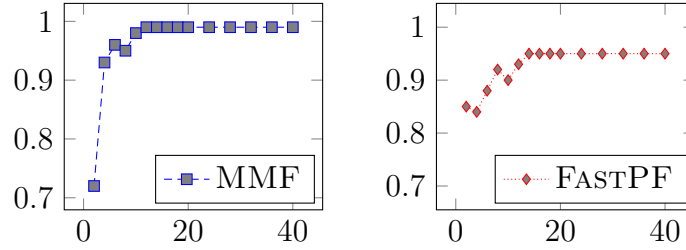


FIGURE 4.11: Fairness index as a function of number of batches

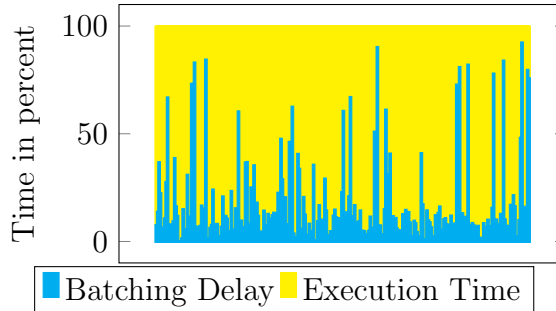


FIGURE 4.12: Comparing Batching Delay and Execution Time when Batch Size=10 seconds

consistently.

Convergence Properties

As our algorithms are randomized, it is important to study how long they take to converge to fair solutions. In our experiments, we find that the number of batches to achieve this convergence is very small, to the tune of 10-15. In Figure 4.11, we present results of a 8 tenant workload with 40 batches, optimized once using MMF and once using FASTPF. The fairness index was computed after every 2 batches. It can be seen that both algorithms converge to their respective optimal values in less than 15 batches. We plan to systematically study which parameters define the rate of convergence of the algorithms as a future work.

Batch Size

The analysis presented earlier shows that the best performance is achieved when a query batch contains a sufficient number of queries so that a cached view can help two or more of the queries. A small profiling run could help learn the characteristics of

query arrival in order to set the right batch size. Having said this, the Batching Delay could be a bigger factor in determining the batch size in practical ROBUS setups. Our evaluation so far focussed on the throughput gains over a given workload wherein queries arrived for a set time period. Caching policies were compared on how fast (and on how fairly) they processed this query workload. We consistently noticed that scheduling delays dominated batching delays (refer Figure 4.5) so much so that the Batching Delay was a very small factor in the E2E latency. A long running industrial cluster, however, may want to enforce certain latency sensitive SLAs [98] on the queries which would bound the Batching Delay.

We first attempt to understand the impact of Batching Delay in our current setup. We compare the Batching Delay with Execution Time on a workload processed in fixed time batches of 10 seconds in Figure 4.12. The Scheduling Delay is not considered here as it is not a factor ROBUS can control directly. One can notice a number of queries spending a significant proportion of time in Batching Delay. Most of these are short running queries even when disk-bound, and therefore, the Batching Delay harms their E2E latency. Going forward, we would like ROBUS platform to incorporate SLAs on individual query wait times.

Code Base

The code base of ROBUS has been open-sourced [99] and our entire experimental setup can be replicated by following a simple set of instructions provided with the code.

4.4 Conclusion

Emerging Big data multi-tenant analytics systems complement an abundant disk-based storage with a smaller, but much faster, cache in order to optimize workloads by materializing views in the cache. The cache is a shared resource, i.e., cached data can be accessed by all tenants. In this work, we presented ROBUS, a cache

management platform for achieving both a fair allocation of cache and a near-optimal performance in such architectures. We defined notions of fairness for the shared settings using randomization in small batches as a key tool. We presented a fairness model that incorporates Pareto-efficiency and sharing incentive, and also achieves envy-freeness via the notion of core from cooperative game theory. We showed a proportionally fair mechanism to satisfy the core property in expectation. Further, we developed efficient algorithms for two fair mechanisms and implemented them in a ROBUS prototype built on a Spark cluster. Our experiments on various practical setups show that it is possible to achieve near-optimal fairness, while simultaneously preserving near-optimal performance speedups using the algorithms we developed.

Our framework is quite general and applies to any setting where resource allocations are shared across agents. As future work, we plan to explore other applications of this framework.

Importance of Memory Tuning

Allocation and usage of memory in modern data-processing platforms is based on an interplay of algorithms at multiple levels: (i) at the resource-management level across containers allocated by resource managers like Mesos and Yarn, (ii) at the container level among the OS and processes such as the Java Virtual Machine (JVM), (iii) at the framework level for caching, aggregation, data shuffles, and application data structures, and (iv) at the JVM level across various pools such as the Young and Old Generation as well as the heap versus off-heap. We use THOTH, our data-driven platform for multi-system cluster management, to build a deep understanding of different interplays in memory management options. Through multiple memory management apps built in THOTH, we demonstrate how THOTH can deal with multiple levels of memory management as well as auto-tune them.

This chapter stresses the importance of tuning memory management options using a detailed empirical study to establish the space of problems. Following this, Chapter 6 and Chapter 7 develop alternative auto-tuning solutions. Chapter 7 also provides an evaluation of these approaches.

5.1 Introduction

We are witnessing an explosion in the number of data-processing platforms: Hadoop, Spark, HBase, Cassandra, Kafka, Storm, Flink, Presto, and others. Some key observations about these platforms are:

- **JVM-based:** Most of these platforms run on the Java Virtual Machine (JVM) and are written in JVM-based languages like Java, Scala, and Clojure. The JVM is recognized industry-wide as a developer-friendly, stable, efficient, and secure system.
- **Container-friendly:** These platforms are invariably run in multi-tenant environments where resources are allocated and isolated using *containers*, e.g., using technologies like Yarn, Mesos, and Docker.
- **Memory-intensive:** Jim Gray is credited with predicting that “Memory is the new disk.” In-memory data storage is increasingly the focus of these platforms.

The central premise of this work is that modern data analytics will increasingly be done in memory on shared-nothing clusters using JVMs running inside containers. However, to the best of our knowledge, no empirical study has been done to understand issues faced by these platforms such as:

1. The interplay of memory-management decisions made at multiple levels such as: (i) the resource-management level across various containers allocated on a cluster, (ii) within a container, (iii) at the application level for caching, aggregation, data shuffles, etc., and (iv) inside the JVM.
2. The impact of the JVM’s *garbage collection (GC)* on response time, throughput, and predictability of performance.

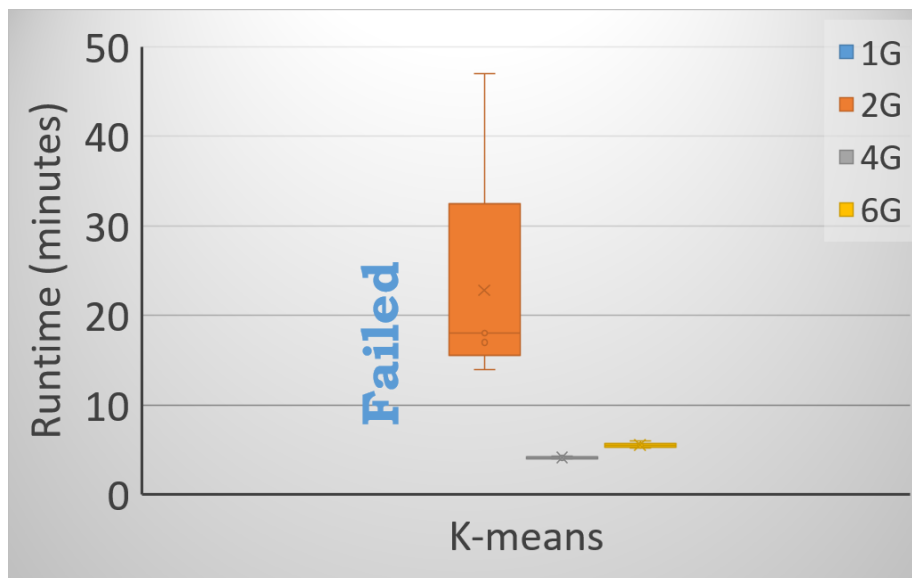


FIGURE 5.1: Analysis of K-means workflow showing interplay in different user goals by varying memory per container

3. The resource-usage overheads of serializing and deserializing data for use in the JVM heap, as well as the use of off-heap storage to possibly reduce these overheads.
4. The chances of failure due to “out-of-memory exceptions.”
5. Last, but not the least, the impact of the many tuning knobs at various levels to control memory management, including the number and sizes of memory pools, how to serialize/deserialize, how to perform GC, and others.

A simple example suffices to bring out the complexity that application developers and database administrators of these modern platforms face. Figure 5.1 plots runtimes of a sample K-means application workflow run on a ‘Spark on Yarn’ cluster of 10 nodes. The application is provisioned with one container on each node. Memory allocated to this container is varied across test configurations. The application caches a dataset of vertices in memory on which the computations are to be done. With only 1GB memory, containers run out of memory while caching this dataset. As a

result of multiple container failures, Yarn kills the application. When the memory is increased to 2GB, some containers still go out of memory. They are replaced with new containers which re-do the failed tasks and the application eventually succeeds. In this process though, not only does the performance suffer, but the predictability also takes a hit. The number of containers that fail due to insufficient memory is hard to predict due to data distribution and scheduling factors. A memory of 4GB proves sufficient for the application to give good and predictable performance. In this setting, the entire data fits in the available memory on each node. If memory is over-provisioned, like the case of 6GB in the graph, then the runtime suffers since the JVM's garbage collection delays are higher due to the larger size heap.

This example showcases the interplay among four key performance metrics: (i) response time, (ii) efficiency of resource usage, (iii) reliability, and (iv) performance predictability. We do a systematic study of how different memory management options impact performance metrics. Section 5.2 identifies all the memory pools used at different levels along with key configuration parameters (i.e., tuning knobs) that control memory management at each level. Section 5.3 presents an empirical evaluation to understand the impact of memory management options.

5.2 Memory Management Options

Data analytics systems employ a resource manager, such as Yarn [19], to allocate cluster resources to applications. Each application is provided with a set of *containers* by the resource manager. A container is simply a slice of physical resources carved out of a node allocated exclusively to the application. Figure 5.2 shows how cluster memory is allocated to multiple containers. Many popular data analytics systems (e.g., Spark, Flink, and Tez) use a JVM-based architecture for memory management. For applications running on these systems, a JVM process is executed inside each allocated container. As shown in Figure 5.3, the container memory is divided into

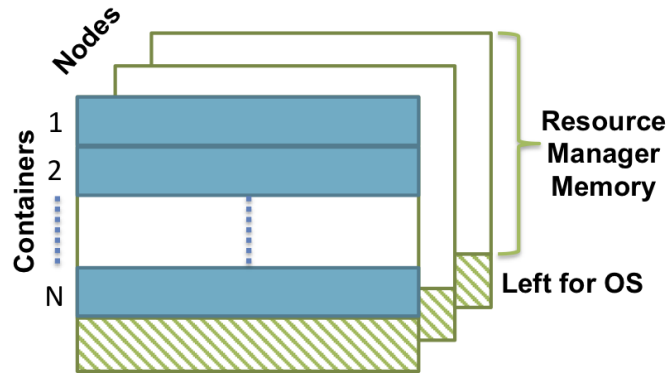


FIGURE 5.2: Node memory managed by Resource Manager

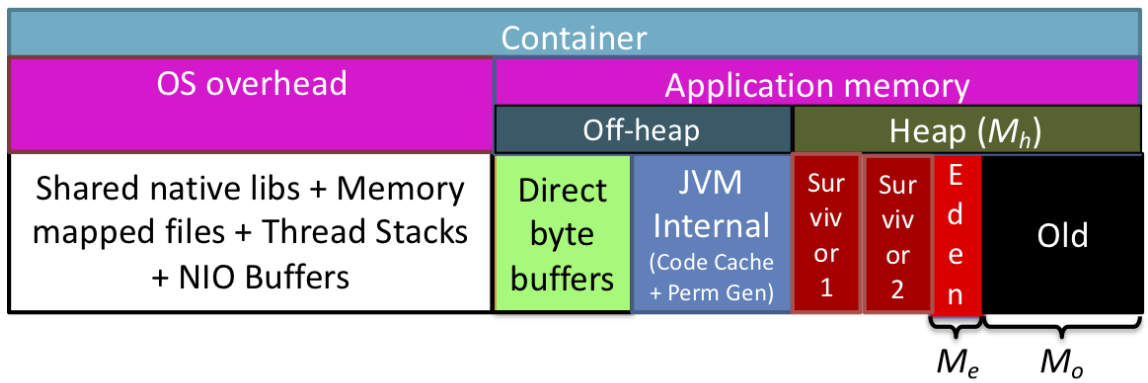


FIGURE 5.3: Container memory managed by JVM

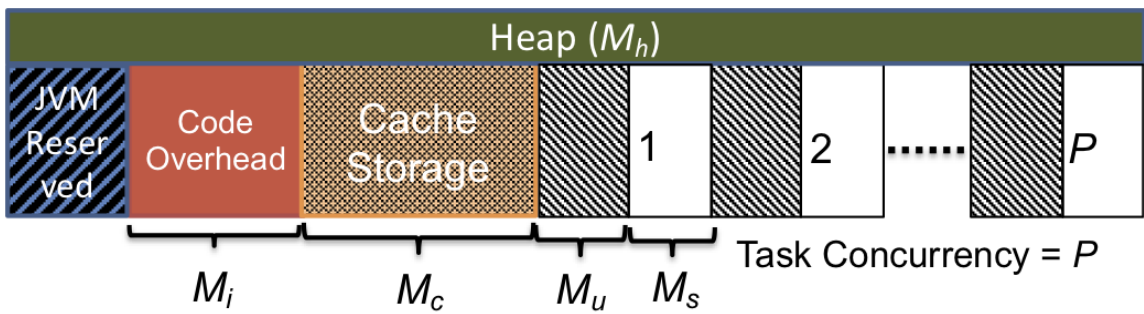


FIGURE 5.4: Heap managed by application framework

two parts: (a) Memory available to the JVM process, and (b) An overhead space used by the operating system for process management. The JVM further divides its allocation into a heap space and an off-heap space. All objects, except native byte buffers, created by the application code are allocated on the heap and are managed by the JVM's generational heap management as described next.

One of the most salient features of the JVM is the *safety* in memory management. Unlike in the native programming languages, applications written in JVM languages do not explicitly allocate and free memory. Instead, the JVM periodically runs a process of garbage collection (GC) that frees up unreferenced objects from **Heap**. **Heap** is internally organized into multiple pools, each holding objects of certain *age*. The number of pools and the size of each pool are determined by the GC policy configured at the time of process launch. We focus on the default GC policy, called ParallelGC, throughout this paper. ParallelGC uses two pools: Young generation and Old generation. As the names suggest, the young generation pool stores newly loaded objects and the old generation pool stores long-living objects.

ParallelGC splits up the young generation into one *Eden* space and two *Survivor* spaces; only one of which is occupied at any given time. Each of the pools is a contiguous block in memory. All newly created objects go to **Eden** first. When **Eden** is filled up, a collection called *Young GC* is triggered. A young GC collects unreferenced objects from **Eden** and the occupied Survivor. Objects that have *aged enough* (determined by the user-specified parameters 'InitialTenuringThreshold' and 'MaxTenuringThreshold' [100]) are moved to the **Old** pool and the objects that have not aged enough are moved to the other empty survivor. When a young GC process finds an almost full old generation, it triggers a *Full GC* process. This process collects all unreferenced objects from **Old**, moves surviving objects from Young to **Old**, and compacts the **Old**.

Certain phases of any GC processes include *stop-the-world* pauses where the ap-

Table 5.1: Memory pools across multiple levels, viz. container, application framework, and JVM, along with the parameters controlling their usage.

Pool Name	Notation	Description	Parameters
Heap	M_h	Heap size in a container	Heap Size
Cache Storage	M_c	Data stored in memory	Cache Capacity as a fraction of Heap
Task Shuffle	M_s	Shuffle memory used by a task	Shuffle Capacity as a fraction of Heap
Task Unmanaged	M_u	Objects for reading/writing task inputs	Task Concurrency
Code Overhead	M_i	Application code objects	-
Old	M_o	JVM old generation	NewRatio
Eden	M_e	Eden space in young generation	NewRatio and SurvivorRatio

plication threads are paused. It is, therefore, important to optimize the GC process from a performance standpoint. Key tuning options controlling ParallelGC are related to sizing the young and old pools. A parameter *NewRatio* sets the ratio of the capacity of Old to the capacity of Young. The capacity of Eden within Young is decided by a parameter *SurvivorRatio* which gives the ratio of the capacity of Eden to the capacity of a Survivor space.

Figure 5.4 shows how **Heap** is organized into different pools from the application’s perspective. The application cannot use the entire **Heap** available to the JVM process since some space is reserved for the JVM’s internal objects, and a survivor space is kept empty at all times for garbage collection purpose. The remaining space can be broadly categorized into three pools:

- 1 **Code Overhead**: Memory required for application code objects. Treated as a constant overhead throughout the life of the container.
- 2 **Cache Storage**: Memory used to store the data cached by application. In particular,

storing intermediate results in memory is beneficial during iterative computations.

3 **Task Memory:** The rest of the memory is used by application tasks. The number of tasks running concurrently is set as a configuration parameter (Task Concurrency) which determines the share of memory each task gets to use. A task uses its allocation for two purposes: (a) Memory for shuffle processing tasks such as sort and aggregation (M_s), (b) Memory for everything else which includes input data objects in processing pipeline and serialization/deserialization buffers (M_u).

Allocation to the pools **Cache Storage** (M_c) and **Task Shuffle** (M_s) is controlled by application frameworks both to make an efficient use of available memory and in order to avoid *out-of-memory* errors. Spark, for example, provides a configuration option called *spark.memory.fraction* to bound the two pools [101]; Configuration *taskmanager.memory.fraction* plays a similar role in Flink [102]. The other two memory pools **Code Overhead** (M_i) and **Task Unmanaged** (M_u), however, are not managed explicitly.

In summary, Table 5.1 lists each memory pool managed at each of the container level, application framework level, as well as JVM process level along with the parameters controlling them. These parameters form the list of configuration options to tune memory-based analytics applications.

5.2.1 Application Tuning Approaches

Data analytics application workflows include multiple *stages* of computations where the stages are divided by shuffle dependence. Computation within each stage is parallelized into a number of tasks each processing a small partition of data. Although all tasks from a stage can be run in parallel, they are scheduled in multiple *waves* of execution. The number of tasks in a wave is determined by the number of executors launched for the applications and the number of execution *slots* available on each executor. The number of slots on an executor is a configuration parameter which

indicates the resources (CPU, memory, I/O) expected to be consumed by a task.

Users of data analytics systems expect to achieve the best possible latency (wall clock duration) for their applications. For periodically running applications, the reliability of performance is also an important factor. An application can be tuned at multiple levels: (a) while allocating resources from the resource managers, (b) while setting options provided by the application framework related to the degree of parallelism or internal memory pools among others, and (c) while configuring JVM parameters related to garbage collection of the heap. Table 5.1 provided a summary of parameters affecting the usage of various memory pools in memory-based analytics systems. We now discuss three possible approaches towards tuning these parameters to meet user performance objectives.

I. Robust defaults: Cloud vendors and application frameworks provide default settings for certain parameters that are expected to generalize well towards a broad spectrum of applications. Amazon’s popular cloud-based offering Elastic MapReduce (EMR) provides a default policy for resource allocation on Spark clusters, called *MaximizeResourceAllocation* [103]. This policy creates a single resource container on each worker node allocating it the entire compute and memory resources on that node. The expectation is that the containers will perform the best when allocated the maximum possible resources. Application frameworks such as Spark and Flink provide default settings for application level and JVM level memory pools [101,102]. The defaults use heuristics that generalize well, e.g., `Old` pool size is set higher than the value chosen for `Cache Storage` so that the long living cache objects can fit in the tenured space in heap. However, the policy leaves a lot of scope for performance improvements which can be exploited easily by expert users [16,104] on a per-application basis.

II. Black-box modeling: Black-box approach is often used when it comes to automated configuration tuning in the presence of complex interactions among tuning

parameters. Some recent papers in database configuration tuning, *OtterTune* [105] and *iTuned* [106], as well as in cloud configuration tuning, *CherryPick* [107], are examples of the black-box approach. These solutions model the desired performance objective as a stochastic process which first trains itself using a limited number of carefully-planned experiments. The trained model is then used to suggest a close-to-optimal configuration. The black-box approaches make no assumption about system internals. Although this helps in general applicability to different problem scenarios, the model may take a long time to train to have sufficient confidence in predictions especially when complex interplays are involved among the various configuration options.

III. White-box modeling: One approach towards building an automated tuning solution is to first build a thorough understanding of the impact of configuration options on application performance and then use the understanding in developing analytical *What-If* models for performance estimations. Solutions exist in DBMSes (e.g., DB2 tuning advisor [108]), in MapReduce systems (e.g., Starfish [109]), as well as in cloud infrastructures (e.g., Ernest [110]).

Before diving into tuning approaches, we first develop a better understanding of interactions among memory management parameters using an empirical evaluation in Section 5.3. The chapter concludes with a methodology of collection of low-level profiling data given in Section 5.4; this methodology is critical towards developing the auto-tuning solutions.

5.3 Understanding impact and interactions

Data analytics applications vary widely in their computational model (e.g., SQL, shuffling, iterative processing) and physical design of input data (e.g., partition sizes). These translate to variations in resource consumption patterns of applications which are important to understand for tuning purposes. We have listed the memory pools

Table 5.2: Test suite used in empirical analysis

Application	Category	Dataset	Partition Size
WordCount	Map and Reduce	Hadoop RandomTextWriter (50GB)	128MB
Sort	Map and Reduce	Hadoop RandomTextWriter (30GB)	512MB
K-means	Machine Learning	HiBench huge (100M samples, 20 dimensions, 5 clusters)	128MB
SVM	Machine Learning	HiBench huge (100M examples, 10 features)	32MB
PageRank	Graph	LiveJournal dataset [111] (69M edges, 5M vertices)	128MB

Table 5.3: Test cluster setup for empirical analysis

Number of worker nodes	8
Memory per worker	6GB
CPU cores per worker	8
Compute Framework	Spark-2.0.1
Resource Manager	Yarn-2.7.2
JVM Framework	OpenJDK-1.8.0

critical to performance and the options for managing them in Table 5.1. Here, we explore the impact of each option using five representative benchmark applications listed in Table 5.2. The test suite covers a broad spectrum of computational models and physical designs making it ideal for the empirical study. All experiments were carried out on the cluster setup listed in Table 5.3.

5.3.1 Containers per Node

As shown in Figure 5.2, physical memory on a worker node is divided into multiple containers by the resource manager. This creates a spectrum of choices for an application from using a small number of *fat* containers to a large number of *thin* containers. The default policy used on Amazon EMR clusters, called *MaximizeResourceAllocation*, creates one fat container on each node assigning the entire node memory (minus OS overheads) to it. We vary the number of containers on a node from 1 to 4. The corresponding **Heap** allocated shrinks from 4404MB to 1101MB

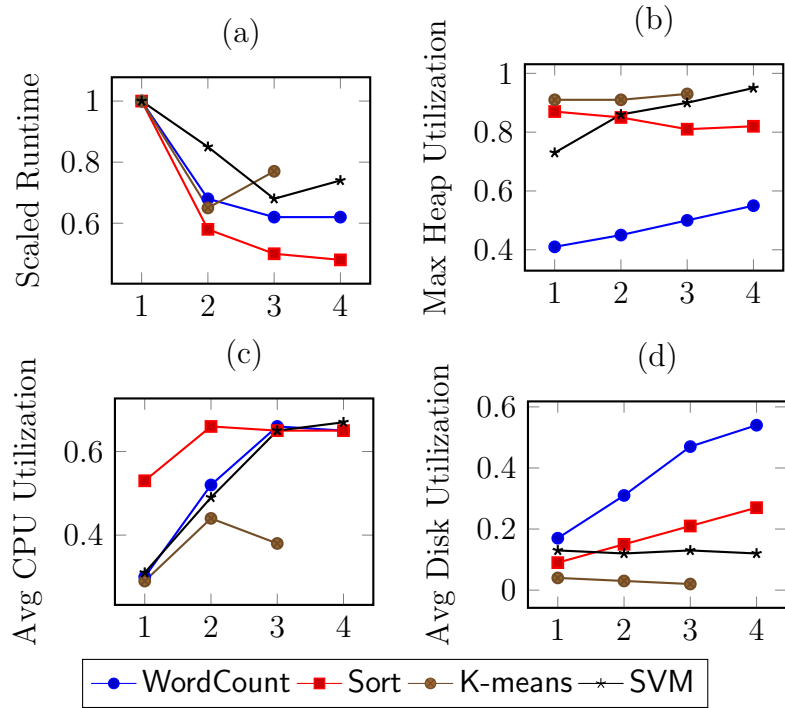


FIGURE 5.5: Impact of increasing number of containers per node on runtime (a), maximum heap utilization (b), average CPU utilization (c), and average disk utilization (d) on benchmark applications. Missing points correspond to instances of failures.

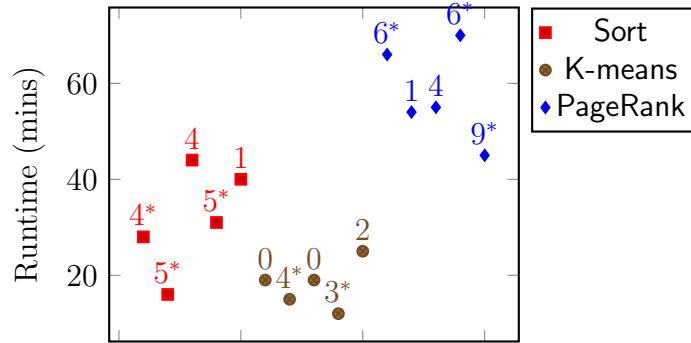


FIGURE 5.6: Exploring failures on one *unreliable* configuration each for Sort, K-means, and PageRank, namely, (1) assigning 70% memory for shuffle, (2) running 4 containers per node, and (3) keeping default settings. Each setup is executed 5 times. Labels above points indicate the total number of container failures during the run, with * mark showing aborted application runs.

proportionately. The other parameters are set to their default values.

Figure 5.5 shows the results. Only the successful application runs are included in the plots; Failures will be discussed separately.

From the runtimes (normalized to the runtimes on the default setup), it can be noticed that **WordCount** and **Sort** perform significantly better on thin containers. Both the applications do not use any cache storage and are, therefore, less memory-bound compared to the machine learning applications, namely, **K-means** and **SVM**. However, the performance does not scale linearly because of CPU and Disk bottlenecks as indicated by an increase in the corresponding resource utilization metrics. Tasks running **K-means** and **SVM** have less memory available for processing because of cache storage. As a result, thinner containers for these applications run into memory pressures leading to a degradation of performance. **K-means**, in fact, runs into *out-of-memory* failures with 4 containers per node.

Observation 1: *Containers should be adequately sized to meet the cache storage and task memory requirements without being too fat.*

Failure cases. Results presented in Figure 5.5 do not include **PageRank** application because it fails under each setup. We also observed a failure for **K-means** under a setup with 4 containers per node. We probe three setups, one each for **Sort**, **K-means** and **PageRank** where containers were observed to fail. Each of these setups was executed 5 times each; Figure 5.6 shows the runtimes and the number of container failures in each case.

Failures observed here are caused by two reasons: (a) *Out-of-memory* errors while creating objects on the heap for either input data deserialization or for buffers fetching data over the network; (b) Resource manager *killing* containers that exceed a preset limit for physical memory usage. A container failure does not necessarily translate to application failure. Spark, our evaluation system, requests new containers to replace the failed ones and retries the failed tasks. If a task fails a specified

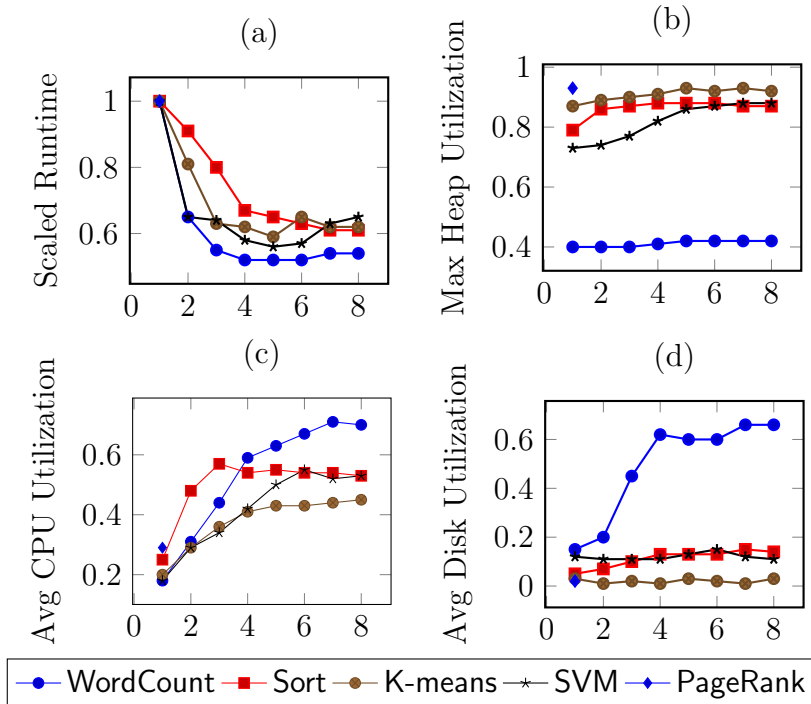


FIGURE 5.7: Impact of Task Concurrency on runtime (a), maximum heap utilization (b), average CPU utilization (c), and average disk utilization (d) for benchmark applications. PageRank runs out of memory for Task Concurrency ≥ 2 .

number of times, the entire application job is aborted. We have marked occurrences of such failures separately in the graph. It can be noticed that there is a huge variability both in terms of the number of container failures and the possibility of application failure under each setup. The runtime of each application, too, is highly unpredictable.

Observation 2: *Over-sizing of memory pools can result in unreliable performance.*

5.3.2 Task Concurrency

An important optimization to increase the throughput of the application tasks is to increase their concurrency. We study this optimization in Figure 5.7 by changing Task Concurrency from 1 to 8. The runtimes are normalized to the configuration with task concurrency set to 1. The pattern that can be noticed from runtimes is that the performance of each application improves with concurrency until a certain

value beyond which it plateaus out.

We include Maximum Heap Utilization, Average CPU Utilization, and Average Disk Utilization plots to inspect the possible bottlenecks causing the plateau effect. For all applications except `WordCount`, the effect can be explained by memory pressures indicated by the maximum heap utilization numbers. As all concurrently running tasks in a container have to compete for a fixed sized heap, increasing task concurrency leads to more garbage collection overheads, curtailing the benefits of increased parallelism. Although `WordCount` tasks have a very low memory footprint, they suffer from CPU and disk bottlenecks for higher settings of Task Concurrency.

Observation 3: *Resource bottlenecks including CPU, I/O, and memory must be accounted for while increasing Task Concurrency.*

5.3.3 Cache and Shuffle memory

Figure 5.4 shows how the memory is organized in an application framework. Applications broadly manage two pools of memory: one as a cache for intermediate data to be used across multiple stages of computation; and the other for sorting or aggregating shuffle data. We explore the impact of memory allocated to internal memory pools for `Cache Storage` and `Task Shuffle` on our benchmark applications; the results are included in Figure 5.8. Since Spark uses a unified memory pool [112] to manage both, we vary a single parameter that changes the fraction of heap allocated to the unified pool. Further, we single out the applications `K-means`, `SVM`, and `PageRank` for the analysis of Cache Capacity as they predominantly use cache over shuffle memory. Applications `WordCount` and `Sort` on the other hand, are analyzed for shuffle memory since they use the unified memory pool exclusively for shuffle objects. Task concurrency for `PageRank` is set to 1 while other applications use the default setting of 2: This change is done in order to avoid *out-of-memory* errors on higher Task Concurrency values observed in the previous experiment.

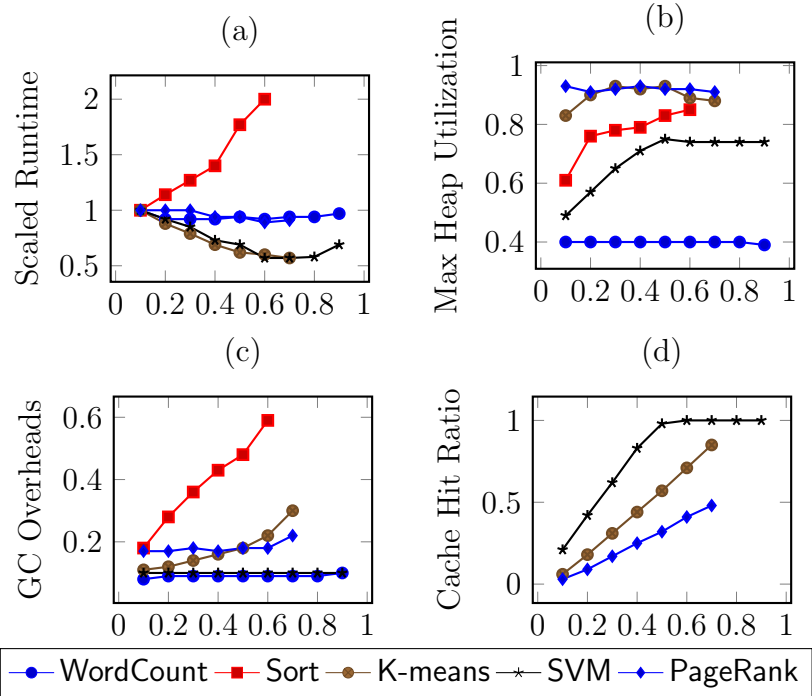


FIGURE 5.8: Impact of Cache Capacity and Shuffle Capacity on runtime (a), maximum heap utilization (b), and average per task GC Overheads (c) for benchmark applications. The X-axis represents Shuffle Capacity for WordCount and Sort which do not use any cache. In case of applications K-means, SVM, and PageRank which predominantly use cache; the X-axis represents Cache Capacity as a fraction of allocated heap. The cache hit ratio for these applications is displayed in plot (d).

It can be noticed that an increase in Cache Capacity results in performance gains for each of the K-means, SVM, and PageRank applications until a certain value beyond which either the performance plateaus or containers run out of memory. We include a plot showing ‘Cache Hit Ratio’ (Figure 5.8(d)) which gives a ratio of the number of data partitions actually found in cache over the total number of partitions requested to be cached. It can be noticed that SVM can fit 100% partitions in cache with a capacity over 0.5, the point where its performance plateaus. K-means hits memory bottlenecks before it can accommodate all the partitions in cache. The GC overheads, derived by averaging the fraction of time spent by tasks in GC processes, also indicate a sharp rise before the containers fail at a Cache Capacity over 0.8. The same effect is observed in the case of PageRank as well.

Observation 4: *Leave sufficient memory for tasks while optimizing for cache storage.*

Analysis of shuffle memory throws the most counter-intuitive result for `Sort` where assigning more shuffle memory leads to performance degradation. Tasks running reduce stage of `Sort` use memory to perform an in-memory sort of data. If the memory allocated from the shuffle pool is insufficient, then the tasks use an external merge-sort by spilling partially sorted records to disk and merging them later. Although increasing shuffle memory leads to lowering the number of spills, increased size of each spill puts more pressure on garbage collection. The GC overheads plot shows that tasks spend 60% time on average in garbage collections for a Shuffle Capacity of 0.6. We analyze an interplay between the shuffle pool size and GC settings in the next subsection which clearly explains why high Shuffle Capacity settings are undesirable. Our results validate the observations made in a recent paper [113] which, through a detailed evaluation on Hadoop, Spark, Flink, and Tez, shows that the performance impact of under-sized shuffle memory pool is very small.

5.3.4 Interactions with GC settings

In Section 5.2, we have described how JVM organizes heap objects into generational pools. From an application’s standpoint, this organization corresponds well to the cache requirements: As the cache memory is expected to have a longer shelf life compared to objects created during task executions, the cached objects are expected to reside in the `Old` pool of JVM. We use the GC parameter `NewRatio` to change the capacity of `Old` and analyze its impact on `K-means` by varying the Cache Capacity from 0.4 to 0.8 in Figure 5.9. The GC overheads are calculated as the average fraction of task time spent on garbage collection. From the runtime numbers, the extreme results are obtained on higher Cache Capacity values (viz. 0.7 and 0.8): While the setups with lower `NewRatio` result in poor performance due to large GC overheads

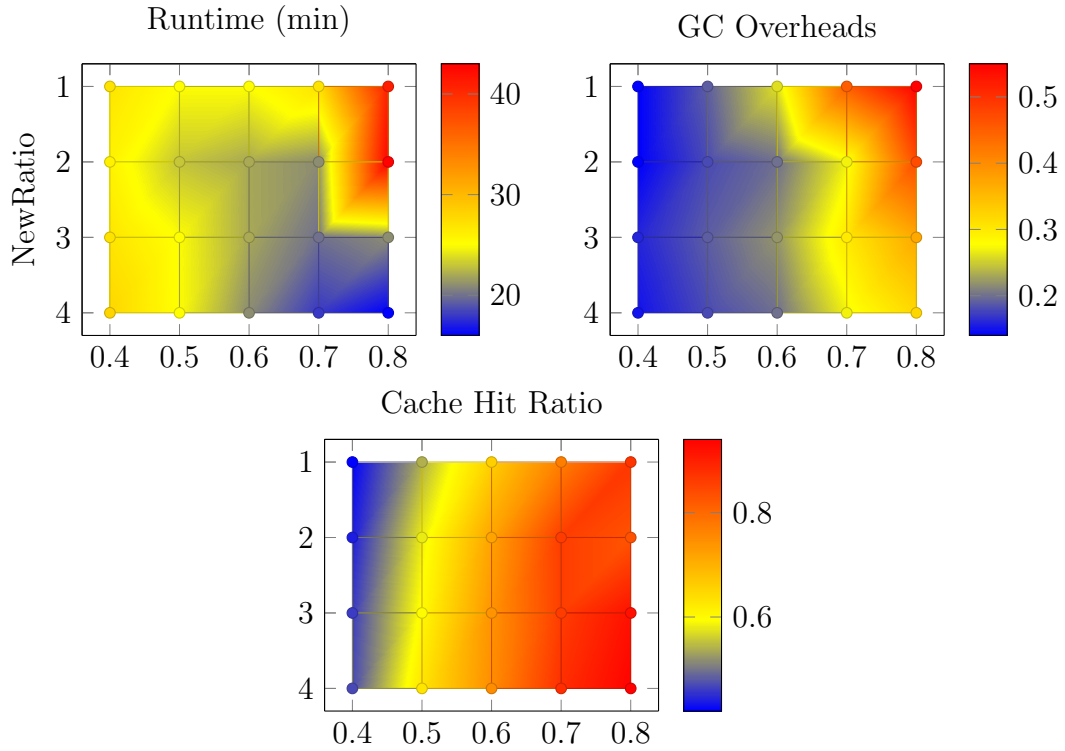


FIGURE 5.9: Interactions between NewRatio and Cache Capacity on K-means

(about 50% of task times), the setups with higher NewRatio perform exceptionally well (3x better). It is important, therefore, to set Old pool size higher than the Cache Storage pool. The key takeaway is presented below.

Observation 5: *Sizing Old smaller than Cache Storage can lead to huge GC overheads (e.g., tasks spending over 50% of their time in GC processes).*

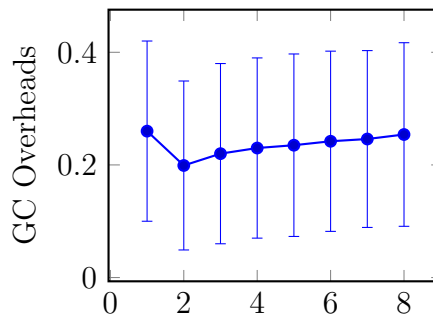


FIGURE 5.10: Impact of NewRatio on per task GC Overheads for K-means with a Cache Capacity of 0.6. Error bars indicate the standard deviation.

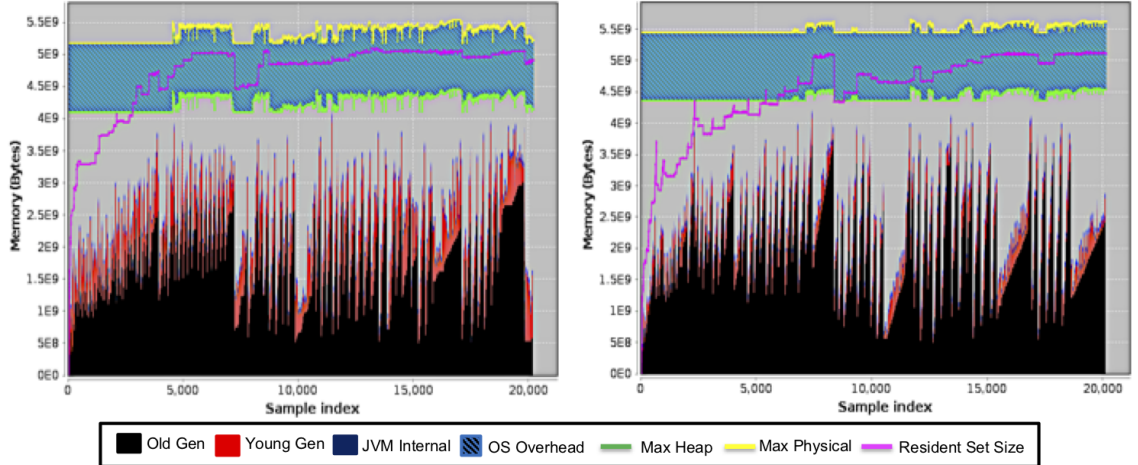


FIGURE 5.11: Comparing memory usage timeline for a container having $\text{NewRatio}=2$ (left) with a container having $\text{NewRatio}=5$. The left side configuration is more prone to failures due to physical memory usage exceeding limit set by resource manager.

The analysis above tells us to set `Old` size higher than `Cache Storage` but how high should it be? It turns out, values too high also lead to increased GC overheads due to frequent collections. Figure 5.10 analyzes `K-means` with a `Cache Capacity` of 0.6 with `NewRatio` increased from 1 to 8. Setting `NewRatio` to 2 provides the best outcome since it *just* fits the cache. Higher settings result in increasingly many invocations of *young GC* which add to the overheads.

The higher `NewRatio` settings, despite adding GC overheads, can help prevent containers exceeding physical memory usage limit set by resource managers which is one source of the container failures reported in Figure 5.6. To understand this, we plot memory usage timeline of two containers in Figure 5.11. The lower value of `NewRatio` implies a lower frequency of garbage collections which results in (on-heap) references to objects created in off-heap space (e.g., `Native ByteBuffer`s used in network data transfers) collected less frequently. It causes the physical memory usage (*magenta* line showing ‘Resident Set Size’) to grow more rapidly, and in some cases, exceeding the maximum physical memory cap (*yellow* line showing ‘Max Physical’). A higher value for `NewRatio` increases the frequency of garbage collection, and as a

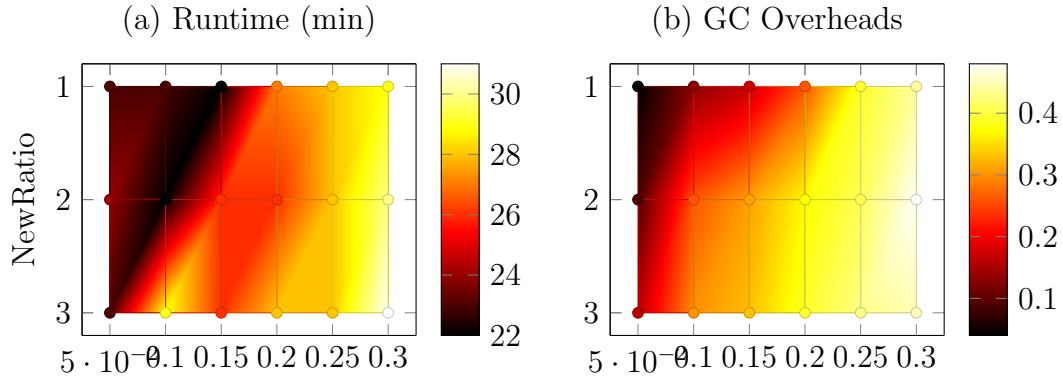


FIGURE 5.12: Impact of NewRatio and Shuffle Capacity on runtime (a) and GC Overheads (b) for Sort

result, helps arrest the growth of physical memory.

Observation 6: *Old capacity values larger than Cache Storage present a trade-off between performance and reliability.*

The shuffle memory behavior is very different compared to the cache storage. While cached objects have a long life, shuffle memory has a very short time span since tasks repeatedly spill the partially aggregated/ sorted results to disk multiple times during execution. Setting Shuffle Capacity larger than Eden pool size (an area within Young generation pool where newly created objects reside) necessitates a *full GC* every time a task spills. Figure 5.12 plots the runtimes and the GC overheads for Sort executed with Shuffle Capacity ranging from 0.05 to 0.3 fraction of Heap size. The NewRatio value is increased from 1 to 3 causing the Eden capacity to go down from 37% to 18% of Heap size. It should be noted that the Eden contains not only the shuffle objects but also other task objects including code data structures and partially processed data partitions. As it is hard to estimate occupancy of Eden at all times, a good heuristic could be to set shuffle memory to 50% of the Eden.

Observation 7: *Shuffle Capacity larger than (50% of) Eden can lead to huge GC overheads.*

Table 5.4: Changes to improve runtime and reliability of PageRank

Containers per node	Task concurrency	Cache Capacity	NewRatio	Runtime (mins)	Cache Hit Ratio	GC Overheads
1	2	0.6	2	66 (aborted)	0.3	0.28
1	1	0.6	2	59	0.32	0.14
1	2	0.4	2	49	0.19	0.12
1	2	0.6	5	53	0.33	0.27

5.3.5 Manually tuning an application

Having explored various options controlling memory pools, we test our understanding by tuning PageRank which, when run under the default configuration specified by *MaximizeResourceAllocation* policy, results in multiple failures (Figure 5.6). The application uses *LiveJournalPageRank* implementation from GraphX [114] library on Spark. The program first coalesces input data into a small number of edge partitions. The coalesced partitions are cached in memory before running iterations on them to update page rank values of graph nodes. Tasks running the coalesce operation need a large amount of memory while combining input data partitions in order to fetch data over the network as well as to store partially processed partitions while more data is being *unrolled*. The problem is further compounded by the fact that the Cache Capacity configured for the application fits only 30% of the cached partitions. This results in partitions being recomputed in each iteration which includes the coalesce computation.

Based on the understanding developed so far, we carry out three changes to the application configuration as listed in Table 5.4. The first row shows the default configuration. The second row lowers Task Concurrency to 1 resulting in a reliable execution (verified by running 5 times) with a runtime of 59 minutes. The second change, listed in the third row, lowers the Cache Capacity which in turn makes more memory available to tasks. This change, despite a lower cache hit ratio, provides a significant improvement to the runtime since it reduces the memory pressure on

tasks. The final change we make is that of increasing `NewRatio` to 5 which prevents failures by collecting the physical memory used by network buffers more aggressively.

We use the understanding developed in this section to develop an auto-tuning algorithm for data analytics applications later in next two chapters. Before that, the rest of this chapter discusses statistics collection using instrumentation of data analytics applications. The statistics collected are used in our auto-tuning approaches.

5.4 Statistics Generation

We use Thoth [16] framework to obtain a profile of the application. The profile includes the following:

- A timeline of the memory usage of JVM pools generated by the JVM GC profiler [115] for every container
- A timeline of resource usage by every container generated by IBM's Performance Analysis Tool (PAT) [116]
- A timeline of memory usage by application memory pools for cache and shuffle generated by custom instrumentation
- Application event log profile providing a timeline of tasks

Table 5.5 lists the statistics derived from an application profile. The first two entries correspond to the container configuration used for the profiled application. Values of these parameters are used by in estimating cache and shuffle memory requirements. Next, we obtain average CPU utilization and average disk utilization values from resource usage profiles. The requirement for **Code Overhead** (M_i) is obtained by looking up heap usage value at the instance of the first task submission to the container. This value corresponds to the memory required for application code objects and is expected to be occupied through the lifetime of the container.

Table 5.5: Statistics derived from an application profile

Notation	Description	Example
N	Containers per Node	1
M_h	Heap size	4404MB
CPU_{avg}	Average CPU usage	35%
$Disk_{avg}$	Average disk usage	2%
M_i	Code Overhead 90%ile value	115MB
M_c	Cache Storage 90%ile value	2300MB
M_s	Task Shuffle 90%ile value	0MB
M_u	Task Unmanaged 90%ile value	770MB
P	Task Concurrency	2
H	Cache Hit Ratio (the fraction of cached data partitions actually read from cache)	0.3
S	Data Spillage Fraction (the fraction of shuffle data spilled to disk)	0

The values obtained from multiple containers in an application profile could have a little variance, so we use a 90th percentile value for stability against outliers. The memory used by **Cache Storage** (M_c) is computed by looking up the maximum cache usage value from the profile. The cache usage value may not necessarily correspond to the actual **Cache Storage** requirement because the application could possibly have rejected some partitions from cache. We record **Cache Hit Ratio** (H) from application logs in order to evaluate the actual requirement.

While both M_i and M_c are considered long term memory requirements of a container, the memory used for task execution ($M_s + M_u$), corresponds to short-term memory requirements. We assume that each task running concurrently equally contributes to the total task memory consumed in order to estimate **Task Shuffle** value (M_s). Like in the case of M_c , the shuffle memory usage value does not necessarily correspond to the actual **Task Shuffle** requirement of the application since the shuffle data could possibly have been spilled because of capacity constraints. Data spillage fraction (S) allows us to estimate the actual memory requirement. The **Task Unmanaged** usage value is the hardest to obtain among the statistics presented in Table 5.5

since the application does not track this memory pool. We use JVM instrumentation to get a good estimate as detailed next.

As described in Section 5.2, JVM uses two garbage collection processes, namely, *young GC* and *full GC*, to collect any unreferenced objects from **Heap**. The *full GC* event cleans up garbage both from young generation and old generation pools. Monitoring heap usage right after a *full GC*, therefore, gives us a more accurate picture of task memory requirements. Figure 5.4 shows the organization of memory pools in **Heap**. Subtracting **Code Overhead** memory and instantaneous **Cache Storage** value from the instantaneous **Heap** value gives us the memory used by tasks running at that instant. As stated previously, we assume that each task contributes equally, and estimate the value of task memory accordingly. Out of the two components in task memory, the instantaneous value of **Task Shuffle** is available from the application instrumentation. The remaining component gives us the instantaneous **Task Unmanaged** value. The 90th percentile over **Task Unmanaged** values thus obtained at each *full GC* event gives us the final estimate of M_u .

***Example.** Statistics for the PageRank application studied in Section 5.3.5 are listed in the third column of Table 5.5. It can be noticed that the application has a high Cache Storage requirement indicated by a high M_c and a low H . Further, a high M_u indicates a high task memory footprint which makes the application susceptible to out-of-memory errors as analyzed before.*

Importance of full GC events: Presence of *full GC* events is an important requirement in obtaining a good estimate on M_u . In case the provided application profile contains no full GC events (significant of an application with very low memory footprint), estimating M_u accurately becomes hard. One solution is to base the calculations on maximum Old pool occupancy. This approach, though, leads to an over-estimation of task memory requirements. We empirically study the sensitivity of recommendations to the provided profile in Chapter 7. The empirical analysis shows

that the estimates made in the absence of the *full GC* events are off by up to two orders of magnitude. Based on this evidence, we discard using Old pool occupancy to estimate M_u . Instead, we recommend simple changes to the application configuration used for profiling. The changes are based on three practical heuristics for increasing GC pressure: (a) Decrease Heap Size, (b) Increase Task Concurrency, and (c) Increase NewRatio. The new profile generated using the heuristics is expected to contain *full GC* events, making it more suitable for statistics generation.

Guided Bayesian Optimization Approach

6.1 Introduction

An emerging school of thought in building autonomous (or, self-driving) data processing systems is to leverage the “black box” algorithm of Bayesian Optimization for problems of this flavor both due to its wider applicability and theoretical guarantees on the quality of results produced. The black-box approach, however, could be time and labor-intensive; or otherwise get stuck in a local minima. We study an important problem of auto-tuning the memory allocation for applications running on modern distributed data processing systems. A simple “white-box” model is developed which can quickly separate good configurations from bad ones. To combine the benefits of the two approaches to tuning, we build a framework called Guided Bayesian Optimization (GBO) that uses the white-box model as a *guide* during the Bayesian Optimization exploration process. An evaluation carried out on Apache Spark using industry-standard benchmark applications shows that GBO consistently provides performance speedups across the application workload with the magnitude of savings being close to 2x.

6.1.1 AutoTuning Techniques

Data analytics systems, such as Spark, Flink, and Tez, support a variety of computational patterns such as Map-Reduce or Iterative Machine Learning. Each of these systems provide multiple tunable configuration options to optimize each use case. The problem of finding the best configuration of such a tunable system can be specified by the following maximization (or minimization) objective f :

$$\mathbf{x}^* = \arg \max_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}) \quad (6.1)$$

where \mathcal{X} is the space of features/configuration options.

If the optimization function f is known apriori, e.g. a linear combination of features, regression-based optimization can be used to solve the problem. Such functions are referred to as *white-box* optimization models. DB2 tuning advisor [108] is a classical example of auto-tuners using white-box models. Modern white-box auto-tuners use feedback-based techniques to recalibrate and revalidate the model in order to support dynamic system changes, e.g. Starfish [109] and Ernest [110]. However, building parametric objective functions is often non-trivial in presence of complex interactions among configuration options and performance metrics [117].

Black-box optimizers are employed when the objective function is not known apriori. Black-box models learn the objective function by sampling and observing points from the configuration space. Bayesian optimization [118] is a powerful black-box technique that is applied to varied designs including Database systems [105,106], Streaming [119], Storage systems [120], and Cloud infrastructures [107,121]. Bayesian Optimization (BO) first prescribes a prior belief on the probability distribution of f (e.g. Gaussian). It then sequentially updates its belief by learning the posterior distribution from observing samples from configuration space \mathcal{X} . In each iteration, the optimizer suggests the next sample to probe using an *acquisition function* which

balances exploration (i.e. acquiring new knowledge) and exploitation (i.e. using existing knowledge in decision making). This way, BO provides a theoretically justified means of searching for optimal configuration making it an attractive choice for auto-tuning.

6.1.2 *Our Contributions*

We have learned from our experience with empirical analysis of memory-based analytics that a simple white-box tuner can *quickly* produce decent results. The black-box approach of Bayesian Optimization (BO), however, offers other benefits including wider applicability and theoretically-guaranteed convergence to the optimal settings. Motivated by this, we develop a framework called ‘Guided Bayesian Optimization’ (GBO) combining the benefits of the two. GBO supplements a Bayesian Optimizer (BO) with an approximate white-box model capable of separating good configurations from bad ones in quick time. The BO in GBO is modeled as a Gaussian Process (GP) [122]. The model is bootstrapped with a small number of pre-executed samples taken from Latin Hypercube Sampling [123]. The same set of samples is used to bootstrap a white-box model with required low-level statistics. During an iteration of sequential tuning, GBO recommends a configuration to probe next which both maximizes the acquisition function over the current posterior of the GP and is expected to perform well according to our white-box model as well. Section 6.2 details the GBO framework. Following that, Section 6.3 discusses the white-box model we have used in our system prototype.

6.1.3 *Related Work*

One approach towards auto-tuning has been to assume a shape of optimization objective, e.g. linear or quadratic. Using this approach, some researchers have used regression models [124], while others have used hill-climbing techniques [125]. Bayesian

optimization works without a need of such assumptions and is, therefore, more versatile. We use Gaussian Process (GP) Regression [122] as our candidate BO. There are alternative techniques in literature, such as ensemble random forest [126], which have been used for systems employing BO [121]. However, unlike GP, none of them provide an estimate of the variance of its predictions.

The problem of speeding up a black-box bayesian optimizer with white-box models is fairly recent. Dalibard et. al. [127] propose *Structured Bayesian Optimization* (SBO) which lets system developers add structure to the optimizer by means of bespoke probabilistic models including simple parametric models inferred from low-level performance metrics observed during a tuning run. The combination of non-parametric bayesian optimizer and evolving parametric models help with a faster system convergence. SBO needs considerable system expertise to design a combination model for auto-tuning. Our focus, instead, is restricting exploration of bayesian optimizer to a smaller space of configurations identified by simple white-box models.

Another recent work specifically targeted at finding best VM configurations, Arrow [121], augments a bayesian optimizer with not only VM characteristics but also low-level performance metrics. Providing additional features to BO, however, increases the dimensionality of the problem. In smaller dimensional configuration spaces such as ours, the benefits of adding structure by means of low level information are far outweighed by the extra exploration necessitated by the higher dimensionality. Our approach, on the other hand, can help exploit system knowledge without extra exploration overheads.

The idea of combining the black-box acquisition function with a white-box model is similar to an approach of managing a *portfolio* of acquisition functions for BO [128]. While the motivation behind the portfolio management is to find a schedule of acquisition functions that maximizes rewards during a tuning run, our motivation is to incorporate knowledge of system behavior through white-box models.

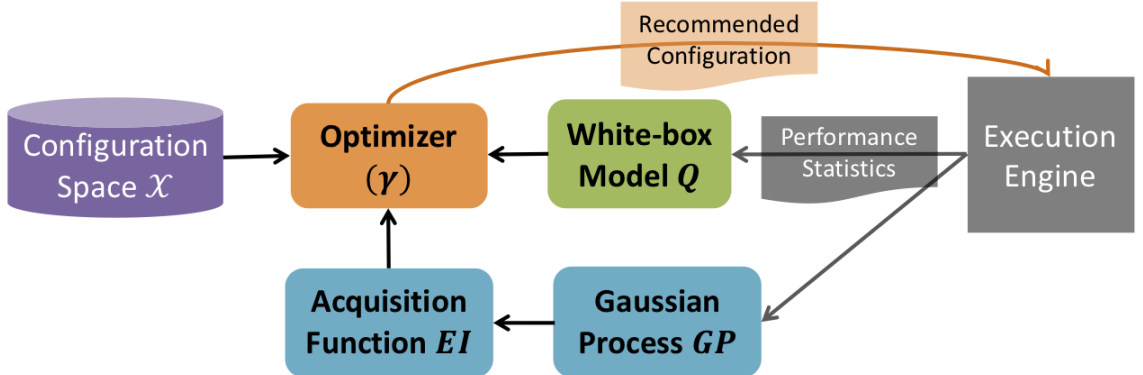


FIGURE 6.1: Workflow of Guided Bayesian Optimization (GBO)

6.2 Guided Bayesian Optimization

We have described the Guided Bayesian Optimization (GBO) framework in brief in Section 6.1. Figure 6.1 outlines the workflow of tuning process. The configuration space used during optimization comes from the parameters listed in Table 5.1. We first describe the BO model used in our framework before outlining the process of selecting next configuration to probe.

BO model requires two building blocks: (a) *Bayesian prior* prescribes a prior belief over the possible objective functions, and (b) *Bayesian posterior* provides a mechanism to sequentially update the belief by learning from new observations. Since the objective function is unknown, we need to use a non-parametric model. Gaussian Process [122] is a popular choice because of its salient features such as support for noisy observations and ability to use gradient-based methods [129]. Using Gaussian Process, the prior belief is modeled as $f(\mathbf{x}) \sim GP(\mu_0, k)$, where $\mu_0 : \mathcal{X} \rightarrow \mathbb{R}$ denotes the prior mean function and $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ denotes the covariance function. Given n sampled points $\mathbf{x}_{1:n}$ and noisy observations $y_{1:n}$ (σ^2 denoting a constant observation noise), the unknown function values $\mathbf{f} := f_{1:n}$ are assumed to be jointly Gaussian, i.e. $\mathbf{f}|\mathbf{x} \sim \mathcal{N}(\mathbf{m}, \mathbf{K})$, and the observations $\mathbf{y} := y_{1:n}$ are normally distributed given \mathbf{f} , i.e. $\mathbf{y}|\mathbf{f}, \sigma^2 \sim \mathcal{N}(\mathbf{f}, \sigma^2 \mathbf{I})$. The posterior mean and variance are then given by the

following:

$$\begin{aligned}\mu_n(\mathbf{x}) &= \mu_0(\mathbf{x}) + \mathbf{k}(\mathbf{x})^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} (\mathbf{y} - \mathbf{m}) \\ \sigma_n^2(\mathbf{x}) &= k(\mathbf{x}, \mathbf{x}) - \mathbf{k}(\mathbf{x})^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{k}(\mathbf{x})\end{aligned}\tag{6.2}$$

where $\mathbf{k}(\mathbf{x})$ is a vector of covariance between \mathbf{x} and $\mathbf{x}_{1:n}$.

An acquisition function provided by BO suggests the next sample to probe based on the posterior distribution. We use one of the most popular acquisition functions, Expected Improvement (EI), given below:

$$EI(\mathbf{x}; \mathbf{x}_{1:n}, y_{1:n}) = (\tau - \mu_n(\mathbf{x}))\Phi(Z) + \sigma_n(\mathbf{x})\phi(Z)\tag{6.3}$$

Here, τ denotes the current best observation, $Z = (\tau - \mu_n(\mathbf{x}))/\sigma_n(\mathbf{x})$, and Φ and ϕ are standard normal cumulative distribution and density functions respectively. The next sample will be either picked from a region where uncertainty is high, captured by $\sigma_n(\mathbf{x})$, or from a region close to the current best, captured by $(\tau - \mu_n(\mathbf{x}))$, thus balancing the *exploration* and the *exploitation*. The GP suggests next sample where the expected improvement is the highest.

GBO uses a white-box model $Q : \mathcal{X} \rightarrow \mathbb{R}$ which is driven by low-level performance statistics and outputs a utility score that helps rank the configurations in terms of expected performance. Q does not necessarily model the exact system behavior because of the inaccuracy in statistics as well as a possible mismatch between the assumed function and the true interactions. As part of its initialization, GBO observes a small number of samples taken using Latin Hypercube Sampling (LHS) [123] over the domain space of configuration options. LHS is an efficient technique to generate near-random samples from a multidimensional space providing a good coverage. These samples are used to bootstrap both the Gaussian process and the white-box model used as a guide. Both models can improve themselves as more samples are

observed during a tuning run.

GBO modifies exploration for the next best point: Using Q , it prunes out the configurations expected to produce poor performance. The change is given by the equation below.

$$\mathbf{x}_{n+1} = \arg \max_{\mathbf{x} \in \mathbf{X}} EI(\mathbf{x}, \mathcal{D}_n) \cdot I(\mathbf{x}) \quad (6.4)$$

where \mathcal{D}_n is a database of n samples including the configurations and the corresponding observations. The optimizer uses a small number of uniform random samples and a few invocations of quasi-Newton hill climbers (e.g. L-BFGS [130]) to explore the space of unseen configurations (\mathbf{X}).

Algorithm 4 Configuration Pruner

Input: White box function Q
Input: Configurations to be explored \mathbf{X}
Input: Query configuration \mathbf{x}
Output: $\{0, 1\}$

- 1: Set $low = \min_{\mathbf{x}' \in \mathbf{X}} Q(\mathbf{x}')$
- 2: Set $high = \max_{\mathbf{x}' \in \mathbf{X}} Q(\mathbf{x}')$
- 3: **if** $Q(\mathbf{x}) \geq U([low, high])$ **then**
- 4: Return 1
- 5: **else**
- 6: Return 0
- 7: **end if**

$I(\mathbf{x}) \sim \{0, 1\}$ is a boolean function telling whether the configuration is worthy of exploration or not. Algorithm 4 details how white box function Q is used in decision making. Q is evaluated on each of the configurations chosen for exploration by the optimizer. The configurations with high Q values are made more likely to be considered by putting a high probability mass on them. The idea behind probabilistically picking (or pruning) a configuration is to have a lesser dependence on accuracy of white-box model. The other option, of picking configurations with Q scores above certain value (say 70 percentile), makes the model completely reliant on white-box model. GBO makes a conscious choice of primarily depending on black-box acquisi-

Table 6.1: Statistics derived from an application profile to be used in GBO

Notation	Description	Example
N	Containers per Node	1
M_h	Heap size	4404MB
M_i	Code Overhead 90%ile value	130MB
M_c	Cache Storage 90%ile value	2300MB
M_s	Task Shuffle 90%ile value	0MB
M_u	Task Unmanaged 90%ile value	70MB
P	Task Concurrency	2
H	Cache Hit Ratio (the fraction of cached data partitions actually read from cache)	0.7
S	Data Spillage Fraction (the fraction of shuffle data spilled to disk)	0

tion function because it makes a more informed decision as it explores more samples; whereas the predictions of white-box function may remain inaccurate.

6.3 White-box Model for Memory Pools

We build a closed-form prototype utility model by understanding memory pool management in data analytics systems. The understanding is based on a systematic empirical study carried out in project ReIM [131]. The utility model relies on the statistics generated from profiles of applications sampled apriori as part of GBO bootstrapping. Although, the statistics could be updated as more samples are observed, we do not consider this approach here. Section 6.4 talks of merits/de-merits of this choice through evaluation. The utility model we build have two characteristics:

- (a) It assigns a high utility to the memory pool allocations meeting application requirements, and
- (b) It penalizes the memory allocations that are either expected to result in *out-of-memory* errors or lead to high garbage collection costs.

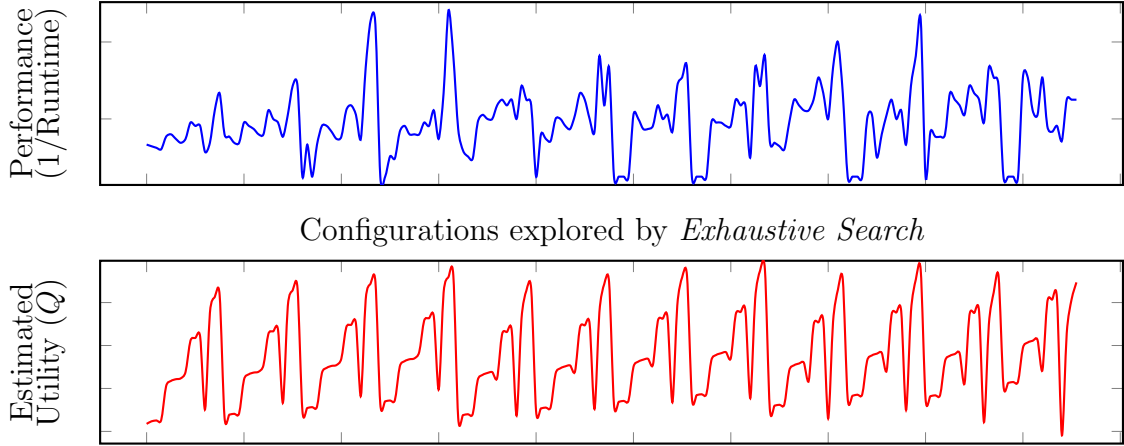


FIGURE 6.2: Accuracy of White-box model estimates for K-means. Configurations explored by gridding the configuration space and running them exhaustively (see Table 6.4) are presented on X-axis.

6.3.1 Statistics Generation

Section 5.4 has described the details on obtaining statistics relating to memory pools usage. In Table 6.1, we provide a list of statistics derived by the process to be used in white-box model.

Example. *Statistics for K-means benchmark application executed on a Spark cluster are listed in the third column of TABLE 6.1. It can be noticed that the application has a high Cache Storage requirement compared to Task Memory requirements.*

6.3.2 Utility Evaluation

Given a test configuration \mathbf{x} and the statistics derived by bootstrapping process, we evaluate the utility of \mathbf{x} using an analytical model given next. We use upperscript \mathbf{x} to denote either the parameter values or the functions evaluated for configuration \mathbf{x} .

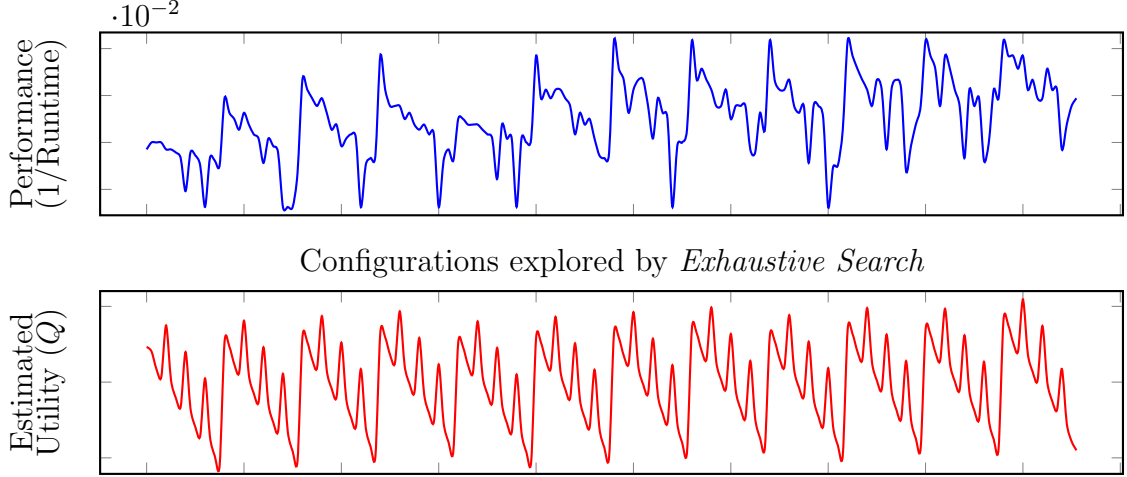


FIGURE 6.3: Accuracy of White-box model estimates for **Sort**. Configurations explored by gridding the configuration space and running them exhaustively (see Table 6.4) are presented on X-axis.

$$\begin{aligned}
 M_{cr} &= M_c/H \\
 M_{sr} &= M_s/(1 - S/P) \\
 R^x &= \frac{M_i + P^x * M_u + \min(M_c^x, M_{cr}) + \min(M_s^x, M_{sr})}{M_h^x} \\
 P1^x &= c_1 * \min(0, R^x - 1) \\
 P2^x &= c_2 * \min\left(0, \frac{M_i + \min(M_c^x, M_{cr}) - M_o^x}{M_h^x}\right) \\
 P3^x &= c_3 * \min\left(0, \frac{P^x * M_u + \min(M_s^x, M_{sr}) - M_e^x}{M_h^x}\right) \\
 Q^x &= R^x - P1^x - P2^x - P3^x
 \end{aligned} \tag{6.5}$$

We first evaluate the requirements for Cache Storage (M_{cr}) and Task Shuffle (M_{sr}) memory pools using the statistics obtained. Here, each concurrently running task is assumed to contribute equally to data spillage. Next, we compute the total utilization of the application level memory pools as a fraction of heap size (R^x). This value could exceed 1 if the memory is over-allocated. We penalize such configurations

using function $P1$. Two more penalty functions, viz. $P2$ and $P3$ are used to penalize the long term memory usage exceeding JVM's Old generation pool capacity and the short term memory usage exceeding JVM's Eden pool capacity. Both the functions correspond to the garbage collection overheads. Finally, function Q^x outputs the utility of the configuration \mathbf{x} .

Penalty factors c_1 , c_2 , and c_3 correspond to the actual magnitude of the penalty. Inferring these factors accurately is an equally hard problem to the auto-tuning problem at hand. However, since the white box model is only used as a heuristic in GBO, it is sufficient to set penalty factors that could only distinguish good configurations from bad ones without accurately modeling the performance. We set each of the factors to 2 in our evaluation. Figure 6.2 and Figure 6.3 present the utilities estimated by the white box model compared with the actual performance on exhaustively searched configurations on two applications: **K-means** and **Sort**. Visual inspection shows that the model is capable of distinguishing the best performing and the worst performing configurations apart. Furthermore, the graphs show that the shape of the objective function can be widely varied across applications substantiating the main motivation behind our work.

6.4 Evaluation

6.4.1 Setup

We carry our evaluation on a Spark cluster configured as listed in Table 6.2. We use five benchmark applications for evaluation which represent Map and Reduce computations, machine learning, distributed graph processing, and SQL processing use cases. The test suite including input data sources is provided in Table 6.3.

Configuration Space: The configuration options we tune correspond to the pa-

¹ Task Concurrency will be determined by dividing this value by the number of Containers per Node.

Table 6.2: Test cluster setup for GBO evaluation

Number of worker nodes	8
Memory per worker	6GB
CPU cores per worker	8
Compute Framework	Spark-2.0.1
Resource Manager	Yarn-2.7.2
JVM Framework	OpenJDK-1.8.0

Table 6.3: Test suite used in GBO evaluation

Application	Category	Dataset
K-means	Machine Learning	HiBench [132] Huge (100M samples, 20 dimensions, 5 clusters)
Sort	Map and Reduce	Hadoop RandomTextWriter (30GB)
WordCount	Map and Reduce	Hadoop RandomTextWriter (50GB)
SVM	Machine Learning	HiBench huge (100M examples, 10 features)
PageRank	Graph	LiveJournal dataset [111] (69M edges, 5M vertices)

Table 6.4: Sample space used in *Exhaustive Search*.

Containers per Node	[1, 2, 3, 4]
Concurrent tasks per node ¹	[2, 4, 6, 8]
Cache Capacity/ Shuffle Capacity	[.2, .4, .6, .8]
NewRatio	[1, 3, 5, 7]

rameters controlling memory pools listed in Table 5.1. The maximum heap available for allocation per node is 4404MB. We allow it to be distributed equally among 1, 2, 3, or 4 Containers. We limit the number of concurrently running tasks on a node to the number of physical CPU cores (=8). Therefore, the Task Concurrency can range from 1 to the ratio of the number of physical CPU cores to the number of containers on the node. For example, if 2 containers are launched on a node, Task Concurrency on each container ranges from 1 to 4. Cache Capacity and Shuffle Capacity values are set as a fraction (ranging from 0 to 1) of Heap size. As Spark provides a unified memory pool [112] combining both Cache Storage and Task Shuffle, we set the capacity of the unified pool to the sum of Cache Capacity and Shuffle Capacity. The

lowest possible value for NewRatio is 1. The maximum, while unbounded in theory, is limited to 9 (i.e. at least 10% of the heap is available to the young generation) in our setup. We keep the SurvivorRatio to its default value.

Exhaustive Search: In order to find an optimal configuration, we carry out an exhaustive search by gridding the configuration space. The domain of each parameter is discretized into 4 values as listed in Table 6.4. We use only one of Cache Capacity and Shuffle Capacity depending on the dominant requirement of the application under test in order to speed up the process. The minor memory pool capacity is kept constant at 10% of the heap.

Bayesian Optimization (BO): As detailed in Section 6.2, we use Gaussian Process (GP) Regression as our candidate for black-box tuning. The GP is implemented using *scikit-learn* library in Python [133]. Like in the case of *Exhaustive Search*, we use only the dominant memory pool between Cache Storage and Task Shuffle during optimization. The objective function is set to minimize latency, or alternately, maximize the inverse of the runtime. If a run is aborted due to errors, we set the objective value for the sample to twice the worst runtime obtained on the samples explored so far.

White-box Optimization (WO): The white-box model we have detailed in Section 6.3 is used for exploration in this policy. During each iteration, a small random subset of configurations (about 10%) is evaluated using the white-box model Q . Configuration with the best score is chosen for exploration. We do not use inference from previously executed configurations to improve the model.

Guided Bayesian Optimization (GBO): The GBO policy detailed in Section 6.2 is used with the white-box model Q set as a guide for exploration.

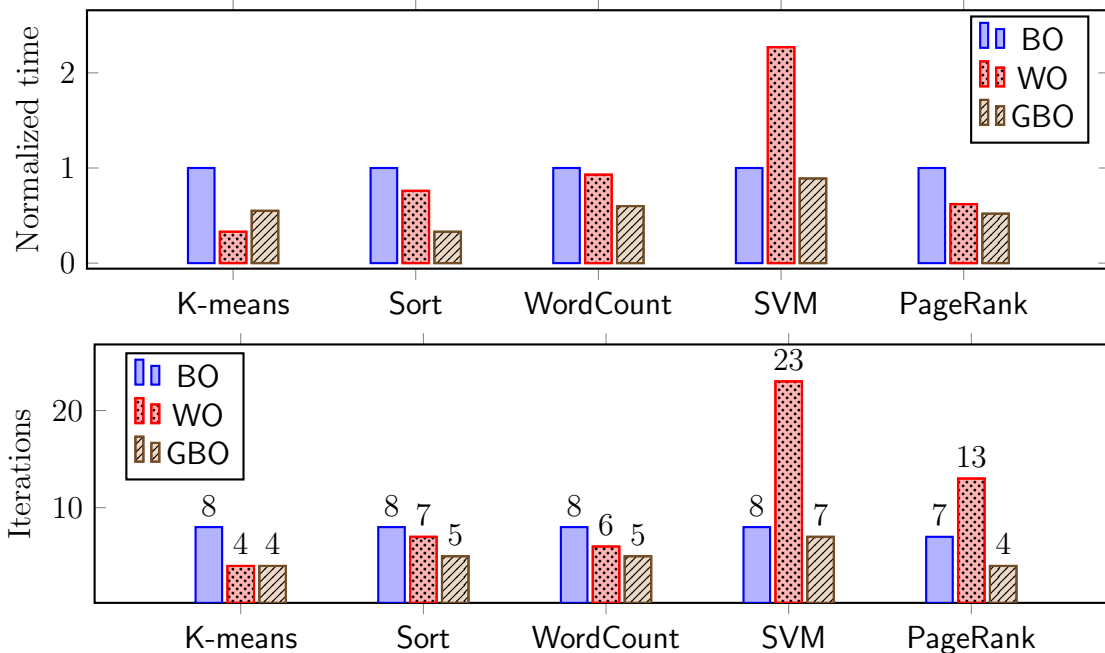


FIGURE 6.4: Analyzing training overheads for various tuners when the tuning is set to stop as soon as a configuration is found having performance within top 5 percentile of the configurations explored by *Exhaustive Search*. The first plot shows training time normalized to time taken by BO. The second plot shows the number of iterations required.

6.4.2 Convergence

We evaluate how various tuning policies fare in terms of the speed of convergence to the optimal configuration. Exhaustive Search takes one to two orders of magnitude longer to converge compared to the other three policies. We do not include its results in the graphs in order to focus on our models. The rest of the policies adaptively sample up to 50 configurations in a run; 5 such runs are carried out for each application. We analyze how much of a training overhead would a policy require if it were to carry on until we find a configuration providing a performance within top 5 percentile of the exhaustively searched configurations. Although this stopping criteria is not practical because the optimal performance is not known a priori, we use it in order to compare how long do different policies take to converge to a good configuration.

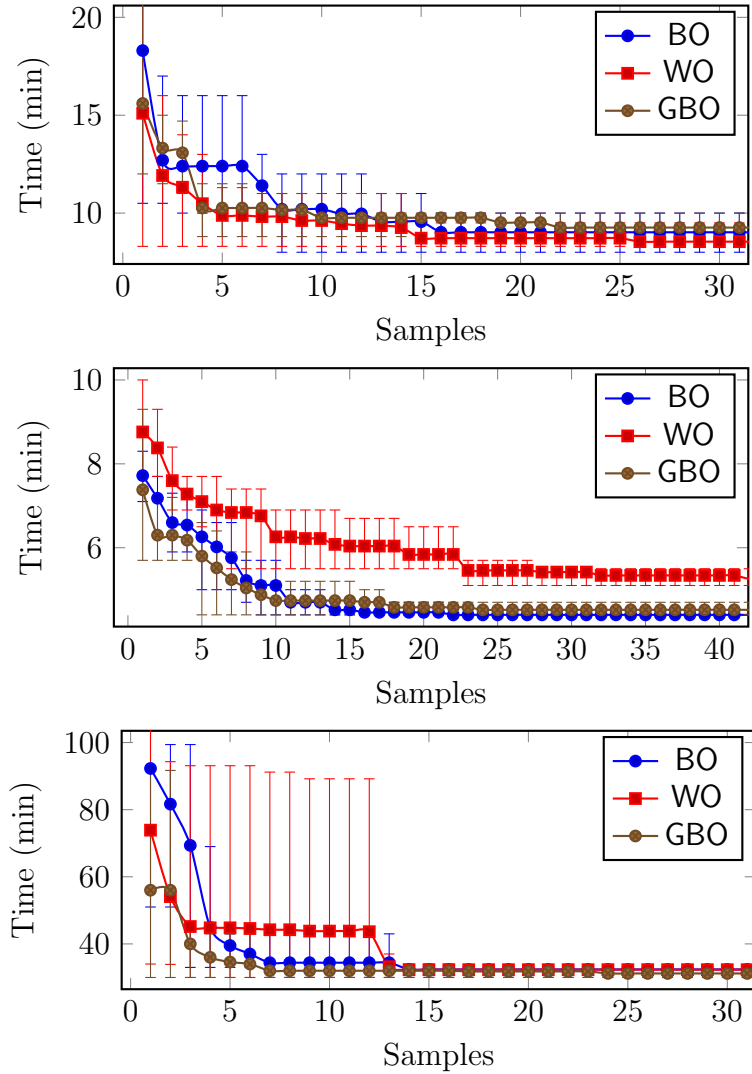


FIGURE 6.5: Convergence of tuning policies for K-means, SVM, and PageRank from left to right. Each tuner is run 5 times; the mean, min, and max values for the lowest runtime on the samples observed so far are plotted on Y-axis.

Figure 6.4 plots the training time and the number of iterations required for each tuning approach. Training times are normalized to the time taken by BO policy on the same application. It can be noticed that the WO policy shows a wide variation in performance: While it provides quick tuning for some applications, e.g. K-means; it leads to huge performance degradations on some others, e.g. SVM. On the other hand, GBO policy consistently lowers training time across applications compared to

BO. On average, GBO reduces the number of iterations to close to half the iterations needed by the black-box policy.

To dive deep into how the policies function, we provide convergence plots for three applications, viz. K-means, SVM, and PageRank, in Figure 6.5. The plot on K-means shows that the BO policy takes 8 iterations on average to find a configuration that runs within 11 minutes (top 5 percentile barrier). The exploration until then often results in configurations with expensive runtimes reflected by its high training time. The WO and GBO policies avoid such configurations and, therefore, can find a top configurations with low overheads. The white-box function Q closely mimics the performance on K-means, as seen in Figure 6.2 earlier, which helps both the policies.

The SVM application throws a curious result where WO takes more than twice the time required for BO. This is a case of white-box model Q bootstrapped with incorrect statistics. We expand on this issue in the next subsection. The important thing to note here is that, despite the substandard white-box model, GBO policy provides a performance similar to the black-box tuning policy. This is a direct consequence of our design decision to rely on acquisition function used in BO for ranking the configurations under exploration.

The PageRank application presents a case wherein WO takes longer to converge despite the white-box model using correct statistics. In this case, many configurations fail with out-of-memory errors due to very high task memory requirements of the application. Although, the white-box model Q penalizes such configurations, it cannot correctly attribute the magnitude of this penalty. Due to this inaccuracy, the WO tuner is able to find a configuration within top 10 percentile within 2-3 iterations, but struggles to find a better configuration post that. On the positive side, it avoids exploring very expensive configurations reflected by its lower training time compared to BO. The GBO policy, like in the case of SVM, brings the best of both worlds to produce a desired configuration within 4 iterations on average.

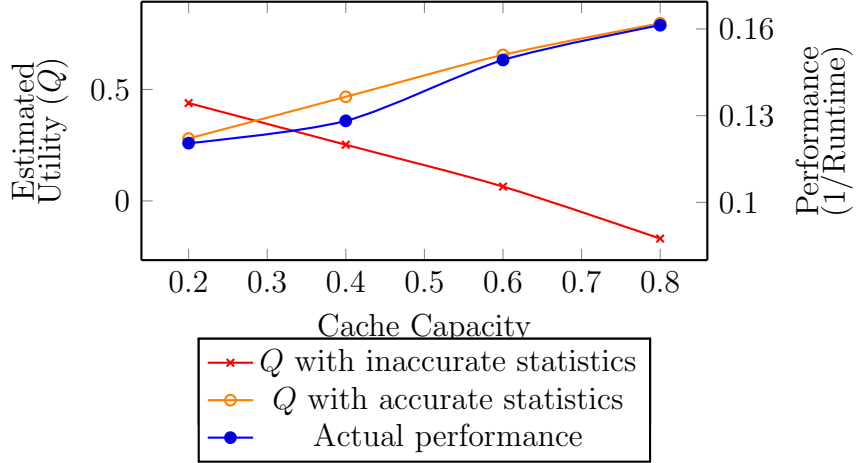


FIGURE 6.6: Graph showing that the white-box function Q modeled with accurate statistics mimics the true performance better. Configuration option ‘Cache Capacity’ alone is varied on X-axis keeping other configuration options constant.

6.4.3 Quality of White-box Model

We showed the quality of our white-box model by comparing the estimated utilities to the actual performance observed on the exhaustively searched configurations on two applications: K-means (Figure 6.2) and Sort (Figure 6.3). Q is able to distinguish good configurations from bad ones in both cases correctly. In case of SVM, however, the white-box model Q makes wrong predictions. This happens due to inaccurate estimation of task memory requirements during bootstrapping. Recalling the statistics generation methodology from Section 6.3, the ‘Task Unmanaged’ memory pool requirement is estimated by monitoring JVM’s *full GC* events. The configurations considered during bootstrapping happen to contain very few such events which leads to an over-estimation of the task memory requirement.

We plot the utility estimated with incorrect statistics along with the actual performance for comparison in Fig. 6.6. It can be seen that, due to over-estimation of task memory requirements, the model incorrectly suggests a performance degradation with increasing Cache Capacity. When we gather better statistics after observing more configurations, the model correctly predicts the actual performance pattern.

This evaluation showcases a need to update the white-box model using feedback. We plan to work on a systematic feedback mechanism to improve tuning at runtime in future.

6.5 Discussion and Future Work

We showed how black-box auto-tuner can be sped up by using simple white-box models built using low-level performance metrics to guide the exploration. We identified an important problem of auto-tuning memory pool configurations in data analytics systems to demonstrate our work. The GBO framework we developed is flexible enough to incorporate white-box models of varying degree of accuracy anytime during the tuning process. As part of the future work, we would like to work on improving the black-box optimization by means of hyperparameter tuning of the Gaussian Process model we have used in GBO. In addition, we would like to explore other system tuning applications suitable for GBO framework.

RelM White-box Approach

We study the problem of autotuning the memory allocation for applications running on modern distributed data processing systems with a contrarian view to GBO. For this problem, we show that a state-of-the-art AI-driven “black box” algorithm performs much worse than an empirically-driven “white box” algorithm, called RelM, that we have developed. The main reason for RelM’s superior performance is that memory management in modern memory-based data analytics systems is an interplay of algorithms at multiple levels: (i) at the resource-management level across various containers allocated by resource managers like Kubernetes and YARN, (ii) at the container level among the OS, pods, and processes such as the Java Virtual Machine (JVM), (iii) at the application level for caching, aggregation, data shuffles, and application data structures, and (iv) at the JVM level across various pools such as the Young and Old Generation. RelM understands these interactions and uses them in building an analytical solution to autotune memory management knobs. The tuning in RelM is fast because: (a) It relies on memory profiles collected on a single prior run, and (b) It focuses on high-impact parameters controlling memory pools. Through an evaluation based on Apache Spark, we showcase that RelM’s

recommendations are significantly better than what commonly-used Spark deployments provide, and are close to the ones obtained by brute-force exploration; while AI-driven “black box” algorithms can get stuck in local optima, or otherwise take considerable time and resources.

7.1 Introduction

Modern data analytics systems, e.g. Spark, Tez, and Flink, are increasingly using memory both for data storage and for fast computations. However, memory is a limited resource that must be managed carefully. Its management involves three players:

- *Application Developer’s perspective:* Judging by the magnitude of StackOverflow posts and various user surveys [134,135], ‘out-of-memory’ errors are a major cause of unreliable application performance. In order to safeguard against such errors, developers need an understanding of the amount of memory their applications really need and how to set various memory configuration options. The prevalent rule-of-thumb to “throw more memory at your applications” is not the best approach while considering costs or the interests of other users.
- *Resource Manager’s perspective:* A resource manager in a multi-tenant setting, e.g., YARN, needs to carefully allocate resources to meet the application performance goals of multiple tenants. Over-allocation leads to wasted resources and a lower throughput, while under-allocation could mean higher latency for tenants. Both problems are commonly observed in production clusters in the industry [98,136].
- *Application Platform’s perspective:* The onus of ensuring *safe* usage of memory is predominantly on the application platforms. Memory is used for various operations such as joins/aggregation, caching inputs/intermediate results, data

shuffling/repartitioning, and sending intermediate/output data over network. Arbitrating memory across these operations is critical in ensuring reliable and fast execution. Improving memory management is a major focus in modern analytics platforms [137, 138].

Our contributions: We study the problem of autotuning the memory allocation for applications running on modern distributed data processing systems. For this problem, we show that a state-of-the-art AI-driven “black box” algorithm performs much worse than an empirically-driven “white box” algorithm, called **ReIM**, that we propose in this work. We believe that our findings will help researchers working on autonomous (or, self-driving) data processing systems make the right choices.

We begin with an overview of the related approaches in literature. Section 7.3 presents system architecture of **ReIM**. The following two sections, viz. Section 7.4 and Section 7.4.1, describe two system components in detail. Finally, Section 7.5 present an evaluation of **ReIM** on Apache Spark using industry-standard benchmark workloads.

7.2 Related Work

There has been extensive work on autotuning the physical design of database systems [139] which includes index selection [140], data partitioning [141], and view materialization [142]. Comparatively less work has been done on autotuning internal configuration parameters like memory pools. Most commercial database systems (e.g., [108]) provide configuration tuning wizards to DBAs which first ask questions about the application workload, and then suggest settings for configuration parameters using white-box models. DB2 has a Self-tuning Memory Manager (STMM) [143] which uses analytical models to determine cost-benefits of internal memory pools. Oracle’s *ADDM* can identify performance bottlenecks due to misconfigurations and

recommend necessary changes [144].

Feedback-driven approaches to tuning [105–107, 125] address the problem by first choosing a sample of experiments to run and collect performance data from. This data is then used to adjust the configuration parameters. The process continues until either the DBA halts it or the additional performance gains are deemed minimal. These tools use machine learning algorithms to find significant configuration parameters [105] as well as to explore the search space of the selected configuration parameters. The former is not useful in our setup since the parameters controlling the memory pools all have significant impact on performance. Our evaluation of the state-of-the-art [105–107] black box approaches such as Gaussian Process Regression [122] found them to perform less favorably than ReIM both in terms of the quality (tendency to get stuck in local minima) and the tuning overheads.

Starfish [109] used a white-box approach with analytical models and simulation to build a *What-If* engine and recursive random search [145] for probing the configuration space of MapReduce systems. Elastisizer [146] used the same approach to auto-size cloud clusters. While the analytical models work well for the specific computational model (MapReduce in this case), maintaining the models for evolving computational patterns (such as iterative graph processing) and changing physical design is a challenge. Ernest [110], a tool to predict performance of large-scale data analytics workloads on the cloud, suffers from the same problem. The same can be said about the white-box approaches taken to tune database configuration parameters such as [147, 148]. ReIM can seamlessly handle evolving computational patterns and data changes in data analytics systems because its models depend solely on memory pool usage profiles.

An important contribution of ReIM is a systematic empirical analysis of memory management in data analytics systems. Other researchers have carried out similar studies in the recent past. Charles Reiss [149] carried out an extensive evaluation of

memory management in Spark for building a tool to provision the right amount of memory to satisfy maximum requirements. Iorgulescu et. al. [113] studied memory elasticity in Hadoop, Flink, Spark, and Tez frameworks and used it to improve cluster scheduling. While these papers analyze each memory pool individually, RelM also considers interactions across the memory pools at multiple levels.

7.3 RelM Tuner

The goal of RelM tuner is to recommend a configuration of memory pools which ensures a reliable and fast performance for a data analytics application in quick time. In particular, RelM meets the following objectives:

- (1) **Safety:** Resource usage should be within allocation at all times.
- (2a) **High task concurrency:** Maximize the number of concurrently running tasks after ensuring (1).
- (2b) **High cache hit ratio:** Provision sufficient memory for cache storage after ensuring (1).
- (3) **Low GC overheads:** Limit the time spent by tasks in GC processes after ensuring (1), (2a), and (2b).

The criteria suggest a priority of goals. Safety is of foremost concern to RelM as it has the highest implications to the application performance: See Figure 5.6 for an example. We rank the goals (2a) and (2b) at the same level. Depending on application characteristics, performance is primarily a function of either one of the two or both (Section 5.3). While the former is constrained by each of the CPU, memory, I/O bottlenecks, the later is constrained by the amount of memory provisioned alone. The goal of *low GC overheads* is ranked the lowest in the scheme of things: Based on the settings used to meet high priority goals, we tune the parameters affecting GC overheads.

It should be noted that we do not pursue a goal of lowering shuffle data spillage

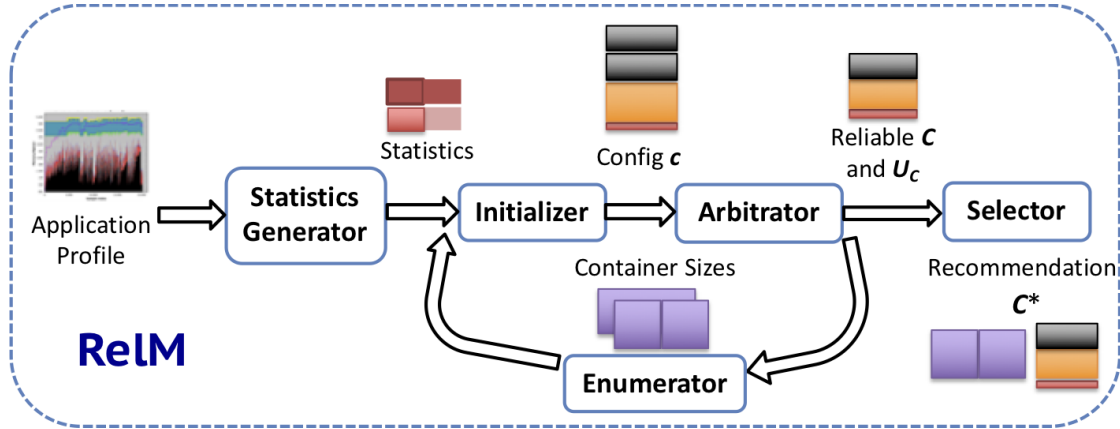


FIGURE 7.1: Application tuning process in ReIM

here. Based on an extensive empirical study carried out by Iorgulescu et. al. [113] on Hadoop, Spark, Flink, and Tez frameworks—in addition to our own evaluation presented in Section 5.3—it is evident that the memory provisioned for data shuffle has limited positive impacts on application runtime. Moreover, high values for shuffle memory could lead to GC bottlenecks as shown in Section 5.3.4. We avoid these overheads by tuning shuffle memory and GC pool settings together as part of goal (3).

ReIM relies on a profile of application run to understand resource requirements of the application. The statistics derived from this single run are used in evaluation of all combinations of container sizes, application memory pools settings, and JVM configurations using analytical modeling. We first make a comment on the container sizes we enumerate during tuning. We support multiple homogeneous containers carved out of a single node with the node memory distributed equally among them as shown in Figure 5.2. This gives us a small finite number of container size configurations.

Example. Amazon EMR’s *m4.large* node types set the maximum memory to be used by resource manager per node to 6GB with a minimum allocation size of 1GB. The possible container configurations in this case, listed as (Containers per Node,

Heap Size), are: (1, 4404MB), (2, 2202MB), (3, 1468MB), and (4, 1101MB). The unallocated memory is left for OS overheads.

We require the application profile to have been generated on a container configuration belonging to the test configurations we enumerated (not necessarily the default configuration) as well. This ensures that the statistics we use while tuning belong to a setup using the same amount of total memory; our white-box models use this assumption.

Figure 7.1 describes the steps in tuning a given application.

- 1 The application profile is processed by the *Statistics Generator* to derive a set of statistics listed in Table 5.5. (Section 5.4)
- 2 The *Enumerator* module runs each container size configuration through *Initializer* and *Arbitrator*.
- 3 Given a container size to probe and the statistics from application profile, the *Initializer* module sets initial settings for memory pools optimizing for each pool independently. (Section 7.4)
- 4 The *Arbitrator* arbitrates memory assigned to various pools by the *Initializer* in order to ensure reliability and low GC overheads. It also calculates a utility score for the resulting configuration corresponding to its memory utilization. (Section 7.4.1)
- 5 Finally, the best settings for each of the probed container size configurations are ranked by *Selector* based on their utility score and the best is returned as the final recommendation.

7.4 Initializer

We use the statistics presented in Table 5.5 to configure each memory pool for a given container configuration identified by the Containers per Node n and the Heap Size of each m_h . We use small caps notation to differentiate the test configuration from the profiled configuration used in statistics generation. A safety factor δ denoting a

fraction of memory to be kept unassigned is used to safeguard against *out-of-memory* errors. The Initializer uses analytical models to configure each of **Cache Storage**, **Task Shuffle**, and **Task Unmanaged** independently. The resulting configuration may potentially lead to memory pressures. We employ *Arbitrator* module later over this initial configuration to handle possible memory contentions as well as potential GC bottlenecks.

Cache storage **Cache Storage** requirement is determined by scaling the maximum cache storage observed in the application profile by the cache hit ratio number. The assumption here is that the total memory accumulated over executors does not change between profiled run and the configuration under test.

$$m_c = m_h * \min\left(\frac{M_c}{H * M_h}, 1 - \delta\right) \quad (7.1)$$

Shuffle memory We estimate **Task Shuffle** by scaling the maximum shuffle memory observed in the application profile by the data spillage fraction. It is assumed that each concurrently running task is an equal contributor to the spillage.

$$m_s = \min\left(\frac{M_s}{1 - S/P}, (1 - \delta) * m_h\right) \quad (7.2)$$

GC settings The **Old** pool of JVM needs to be sized at least as big as the long term requirements, viz. M_i and m_c , in order to lower the GC overheads (Section 5.3.4). The GC parameter *NewRatio* is set accordingly. The **Eden** size is calculated by subtracting two survivor spaces from young generation pool size.

$$\begin{aligned} NewRatio &= \text{ceil}\left(\frac{M_i + m_c}{m_h - M_i - m_c}\right) \\ m_o &= m_h * \frac{NewRatio}{NewRatio + 1} \\ m_e &= m_h * \frac{1}{NewRatio + 1} * \frac{SurvivorRatio - 2}{SurvivorRatio} \end{aligned} \quad (7.3)$$

Task concurrency The number of tasks that can run concurrently in a container is estimated based on the following values obtained from the application profile: (a) average CPU usage per task, (b) average disk usage per task, and (c) maximum per-task memory requirements. The models assume a linear relation in order to get a conservative estimate.

$$\begin{aligned}
 p^{CPU} &= \frac{1}{n} \frac{(1 - \delta) * 100}{CPU_{avg}/P} \\
 p^{disk} &= \frac{1}{n} \frac{(1 - \delta) * 100}{Disk_{avg}/P} \\
 p^{memory} &= \frac{(1 - \delta) * m_{heap}}{M_u} \\
 p &= \min(p^{CPU}, p^{disk}, p^{memory})
 \end{aligned}
 \tag{7.4}$$

Example. The PageRank application studied in Section 5.3.5 when evaluated on the container configuration of $n = 1$ and $m_{heap} = 4404MB$, with safety factor $\delta = 0.1$, results in the following:

$$m_c = 3798MB, m_s = 0MB, p = 5, NewRatio = 9
 \tag{7.5}$$

7.4.1 Arbitrator

Building on the observations made in empirical analysis, we present a general algorithm to tune a given configuration for reliability and low GC overheads. Algorithm 5 presents the pseudo-code.

Step 1 checks if the configuration satisfies the bare minimum requirement which is that of a container running only one task at any given time. Steps 4-9 represent the main loop where actions to change pools configuration are carried out if the combined memory consumed by Code Overhead, Cache Storage, and Task Unmanaged exceeds the Old pool. Please recall that the task memory values are obtained by profiling *full GC* events and indicate task objects tenured to Old as do the objects

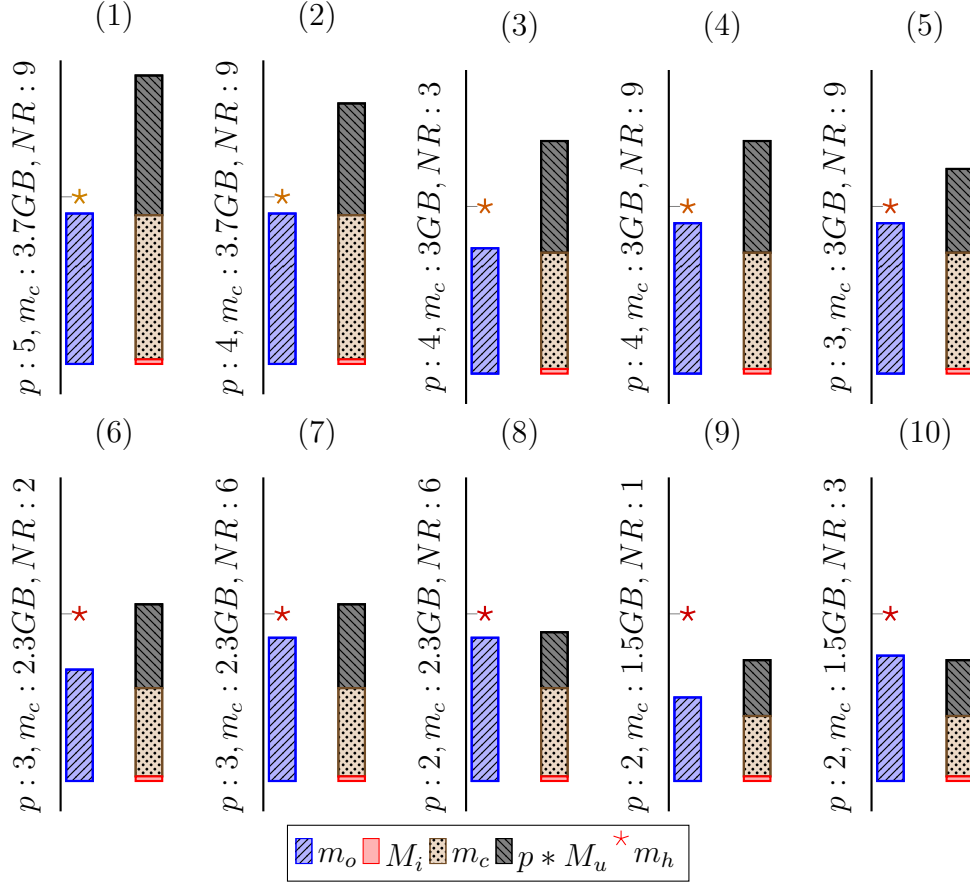


FIGURE 7.2: Working example showing steps of RelM’s *Arbitrator* algorithm on PageRank application

from Code Overhead and Cache Storage pools. If the combined memory exceeds m_{old} , we perform one of the three actions given in Steps 6, 7, and 9 in a round-robin manner:

- Decrease Task Concurrency by 1. This reduces the memory footprint by M_u .
- Decrease Cache Capacity by M_u . We also adjust GC pools so that Old pool is just larger than the value $M_i + m_c$. The idea is to probe if an optimal GC setting for the given Cache Storage value can ensure safety as well.
- Increase old generation pool size by M_u . This optimization trades-off performance to ensure safety against out-of-memory errors (Recall *Observation 6* from Section 5.3.4).

At the end of the main loop, the algorithm has locked in the settings for Task Concurrency, Cache Capacity, and NewRatio. Based on the size of Eden available, we tune the Task Shuffle pool in Step 10. This step helps us avoid the high GC overheads explained in Figure 5.12. Finally, a utility score is computed over the new configuration (Step 12); this score corresponds to the fraction of container heap allocated to internal memory pools.

Algorithm 5 Arbitrator

Input: Configuration $\mathbf{c} = (M_i, M_u, p, m_c, m_s)$
Input: Safety factor δ
Output: Reliable configuration \mathbf{C} with utility $U_{\mathbf{C}}$

- 1: **if** $(M_i + M_u) > (1 - \delta) * m_h$ **then**
- 2: Return flagging insufficient memory
- 3: **end if**
- 4: **while** $(M_i + p * M_u + m_c) > m_{old}$ **do**
- 5: one of the following three actions in a round-robin manner:
- 6: **I.** Decrease p by 1 if $p > 1$
- 7: **II.** Reduce m_c by M_u ensuring that $m_c > 0$.
- 8: Change GC pools using Equation 7.3.
- 9: **III.** Increase m_o by M_u ensuring that
- 10: $m_o < (1 - \delta) * m_h$
- 11: **end while**
- 12: Set shuffle memory $m_s = \min(m_s, 0.5 * m_e / p)$
- 13: Set output $\mathbf{C} = (M_i, M_u, p, m_c, m_s)$
- 14: Set utility score $U_{\mathbf{C}} = \frac{M_i + p * (M_u + m_s) + m_c}{m_h}$
- 15: Return $(\mathbf{C}, U_{\mathbf{C}})$.

***Example.** Continuing with the PageRank example for which the configuration produced by the initializer is given in Equation 7.5. Figure 7.2 details the changes in memory pools starting with the initial configuration shown in (1). It takes 9 iterations of the main loop to reach a reliable configuration which sets Task Concurrency = 2, Cache Capacity = 1.5GB, and NewRatio = 3. Compared to the profiled application run, in which containers fail with out-of-memory errors, this configuration lowers cache capacity by 700MB per container as well as increases NewRatio by 1 thereby improving reliability of the application. This, however, is not the only reliable configuration ReIM finds; a better performing configuration is obtained when the process is*

repeated on a container configuration of 2 Containers per Node. Section 7.5 presents the best result.

Analysis As stated in ReIM goals, *safety* is the primary objective. The Arbitrator meets this objective by ensuring that the combined allocation of internal memory pools remains within the heap size. The next two performance objectives, of having a *high task concurrency* and of having a *high cache hit ratio* are achieved by a two-phase process. The Initializer module first optimizes the **Task Unmanaged** and **Cache Storage** pools corresponding to the two requirements independently against the entire heap size available. The Arbitrator then takes small chunks out of the two memory pools in a round-robin manner until it can meet the *safety* condition. This process results in a *proportionally fair* [150] allocation for the two memory pools. The Arbitrator produces fast results because of two facts: (1) The total allocation of the two memory pools the Arbitrator is provided with can at most be twice the **Heap** size (See Equation 7.4 and Equation 7.1), and (2) Each step reduces memory allocation by M_u . This allows us to finish the arbitration process in a small finite number of steps.

7.5 Evaluation

7.5.1 Setup

We carry our evaluation on a Spark cluster configured as listed in Table 7.2. The applications we have picked for evaluation represent Map and Reduce computations, machine learning, distributed graph processing, and SQL processing use cases; each exhibiting different computational patterns compared to others. The test suite including input data sources is provided in Table 7.1. The input data is stored in HDFS co-located with the compute cluster. We have deliberately changed partition sizes for some of the applications (namely, **Sort** and **SVM**) from the default HDFS

Table 7.1: Test suite used in ReIM evaluation

Application	Category	Dataset	Partition Size
WordCount	Map and Reduce	Hadoop RandomTextWriter (50GB)	128MB
Sort	Map and Reduce	Hadoop RandomTextWriter (30GB)	512MB
K-means	Machine Learning	HiBench huge (100M samples, 20 dimensions, 5 clusters)	128MB
SVM	Machine Learning	HiBench huge (100M examples, 10 features)	32MB
PageRank	Graph	LiveJournal dataset [111] (69M edges, 5M vertices)	128MB
TPC-H	SQL	TPC-H DBGen (50 scale factor)	128MB

Table 7.2: Test cluster setup for ReIM evaluation

Number of worker nodes	8
Memory per worker	6GB
CPU cores per worker	8
Compute Framework	Spark-2.0.1
Resource Manager	Yarn-2.7.2
JVM Framework	OpenJDK-1.8.0

block size of 128MB to create another dimension of variability in the test suite and pose practical challenges for tuning.

Configuration Space

The configuration options we tune correspond to the parameters controlling memory pools listed in Table 5.1. The maximum heap available for allocation per node is 4404MB. We allow it to be distributed equally among 1, 2, 3, or 4 Containers per

Table 7.3: Default values for configuration options suggested by *MaximizeResourceAllocation* and framework defaults.

Containers per Node	1
Heap Size	4404MB
Task Concurrency	2
Cache Capacity + Shuffle Capacity	.6
NewRatio	2
SurvivorRatio	8

Table 7.4: Sample space used in *Exhaustive Search*. Generates 192 unique configurations.

Containers per Node	[1, 2, 3, 4]
Concurrent tasks per node ¹	[2, 4, 6, 8]
Cache Capacity/ Shuffle Capacity	[.2, .4, .6, .8]
NewRatio	[1, 3, 5, 7]

Table 7.5: Samples generated by Latin Hypercube Sampling used in GPR initialization.

Containers per Node	Task Concurrency	Cache Capacity/ Shuffle Capacity	NewRatio
1	4	.6	7
2	1	.4	3
3	2	.2	5
4	2	.8	1

node. We limit the number of concurrently running tasks on a node to the number of physical CPU cores (=8). Therefore, the Task Concurrency value can range from 1 to the ratio of the number of physical CPU cores to the number of containers on the node. For example, if 2 containers are launched on a node of our cluster, Task Concurrency on each container ranges from 1 to 4. Cache Capacity and Shuffle Capacity values are set as a fraction (ranging from 0 to 1) of Heap size. As Spark provides a unified memory pool [112] combining both Cache Storage and Task Shuffle, we set the capacity of the unified pool to the sum of Cache Capacity and Shuffle Capacity. The lowest possible value for NewRatio is 1. The maximum value, while unbounded in theory, is limited to 9 (i.e., at least 10% of the heap is available to the young generation pool) in our setup. We keep the SurvivorRatio to its default value.

Default Policy: The default configuration by Amazon EMR’s *MaximizeResourceAllocation* policy is listed in Table 7.3.

Exhaustive Search: We consider an exhaustive search approach for tuning by grid-
ding the configuration space. The domain of each parameter is discretized into 4 val-

ues as listed in Table 7.4. We use only one of Cache Capacity and Shuffle Capacity depending on the dominant requirement of the application under test just to avoid collecting insignificant data. The minor memory pool capacity is set to 0.1. Despite the dimensionality reduction, *Exhaustive Search* is clearly an inefficient policy: The time taken to run all 192 configurations for an application in our setup is at least 3 days. We performed the *Exhaustive Search* only in order to compare the quality of results produced by the other tuning policies.

Black-box Model: We use Gaussian Process Regression (GPR) as our candidate for black-box tuning. GPR is implemented using *scikit-learn* library in Python [133]. Like in the case of *Exhaustive Search*, we use only the dominant memory pool between Cache Storage and Task Shuffle for optimization, with the minor pool capacity set to 0.1. Since the accuracy of GPR predictions depends on the number of samples it has explored, we bootstrap the model with 4 samples generated using Latin Hypercube Sampling [123] as listed in Table 7.5. The samples provide a good coverage of the configuration space thereby providing GPR a good chance of exploring optimal regions of the space. The objective function is set as the application runtime. If a run is aborted due to errors, we set the objective value for the sample to twice the worst runtime obtained on the samples explored so far. This heuristic ensures that the failing region is ranked low during exploration.

White-box Model: RelM is our white-box model. It is implemented in Thoth [12] which collects profiles for Spark applications with minimal overheads. The details are provided in Section 5.4. The Modules *Initializer* and *Arbitrator* are implemented in Java with ≈ 200 lines of code; the source is available online [151]. We set the safety fraction δ to 0.1 throughout the evaluation.

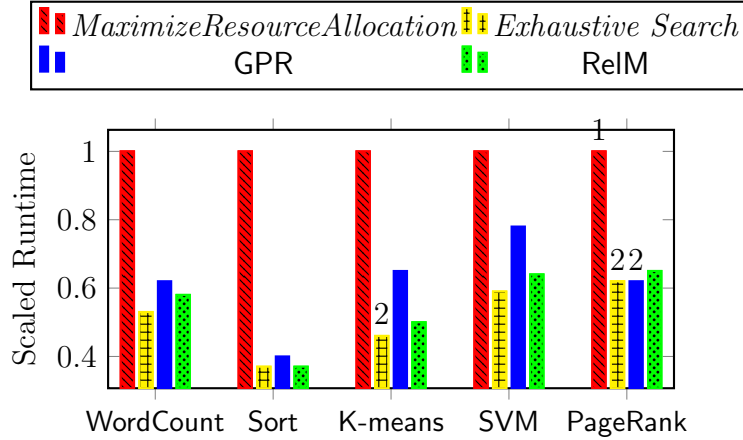


FIGURE 7.3: Performance comparisons of the configurations recommended by various tuning policies. Runtimes are scaled to the values obtained on *MaximizeResourceAllocation* policy. Numbers on top of bars show the number of failed containers.

7.5.2 Quality of Results

ReIM provides a reliable and close-to-optimal performance.

We compare the recommendations made by the default policy of *MaximizeResourceAllocation* against the three tuning policies: (a) *Exhaustive Search*, (b) GPR, and (c) ReIM. Figure 7.3 shows runtimes for the recommended configurations scaled to the runtime on the default policy. The number of failed containers is indicated by a label over the corresponding bar. Across all applications, ReIM provides a runtime not worse than 10% of the best configuration found using *Exhaustive Search*. Furthermore, ReIM ensures that no containers run out of memory. Table 7.6 lists the recommendations. It can be noticed for K-means that a high memory fraction is allocated to **Cache Storage** in the configuration found by *Exhaustive Search*, leaving very small memory for other heap objects. A similar observation can be made about **PageRank** as well. In these instances, ReIM recommends configurations with a lower **Cache Capacity** to better handle the memory pressures.

Black-box models can get stuck in local minima.

We observe that the quality of results for GPR, to a large extent, depends on the

Table 7.6: Comparing recommendations made by tuning policies.

Application	Policy	Containers per Node	Task Concurrency	Cache Capacity	Shuffle Capacity	New Ratio
WordCount	<i>Exhaustive</i>	4	2	0	.4	1
	GPR	3	2	0	.2	1
	ReIM	4	2	0	.2	1
Sort	<i>Exhaustive</i>	4	1	0	.2	1
	GPR	1	8	0	.2	1
	ReIM	4	1	0	.2	1
K-means	<i>Exhaustive</i>	2	2	.8	0	5
	GPR	3	2	.6	0	7
	ReIM	2	2	.68	0	4
SVM	<i>Exhaustive</i>	3	2	.6	.1	3
	GPR	2	4	.2	.1	2
	ReIM	3	2	.51	.07	2
PageRank	<i>Exhaustive</i>	2	1	.4	0	3
	GPR	2	1	.4	0	3
	ReIM	2	1	.24	0	5

Table 7.7: Analysis of a GPR run for SVM.

Sample #	Containers per node	Task concurrency	Cache capacity	New Ratio	Runtime (minutes)
0	1	4	0.6	7	8.5
0	2	1	0.4	3	9.1
0	3	2	0.2	5	6.9
0	4	2	0.8	1	8.5
1	4	2	0.2	8	7.4
2	2	4	0.2	4	6.2
3	1	8	0.2	3	7.3
4	4	2	0.2	1	7.3
5	2	4	0.2	4	6.9
6	2	4	0.2	2	5.8
7	2	3	0.2	1	6.4

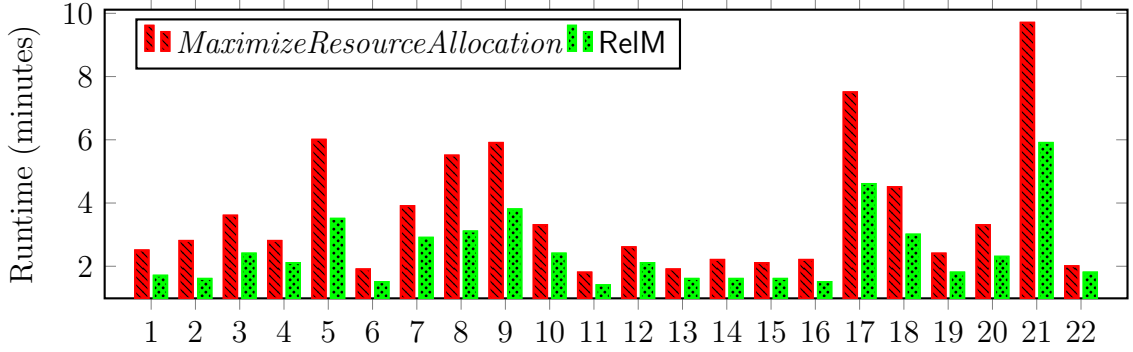


FIGURE 7.4: Comparison of TPC-H Queries run using *MaximizeResourceAllocation* policy and using ReIM.

initial samples used in bootstrapping. The regression often leads to local optima with the stopping condition kicking in before an exploration could find the region of global optima. As a case in point, we provide a log of GPR run for SVM application in Table 7.7. After processing the first four LHS samples, GPR pins down the Cache Capacity to 0.2 and continues exploring the other parameters. The application, as seen in Figure 5.8, requires a capacity of over 0.5 to fit in the entire cached data in memory. While this fact is captured in the white-box models of ReIM, GPR fails to explore this region due to the influence of the initial samples.

Another concern with the black-box models is setting the right objective function. With the objective set to minimize runtime in our evaluation, GPR recommends a configuration for PageRank which is *unreliable*. As a workaround, the GPR algorithm should be given an objective function that incorporates penalties for such failures. It is, however, hard for users to find the right objective function.

ReIM is robust to workload variations.

ReIM does not depend on application workflow and input data design directly, but rather uses interactions between configuration options and resource metrics in its models. It, therefore, can handle different workload types. We have presented results on five widely different benchmark applications. We include a result on TPC-H

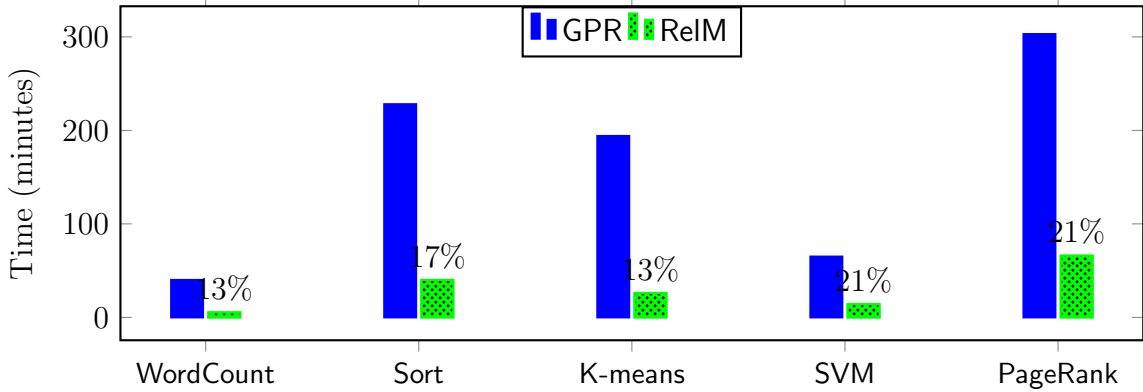


FIGURE 7.5: Training overheads of black-box and white-box tuning.

benchmark workload of 22 queries in Figure 7.4 to further press the point. The workload when executed using the default *MaximizeResourceAllocation* policy took 80 minutes. Profile of this run was then used to tune each query individually. ReIM recommendations bring the runtime down to 54 minutes, a saving of 33%.

7.5.3 Training Overheads

ReIM has minimal training overhead.

The black-box approach requires a number of samples from the configuration space in order to build sufficient confidence. ReIM, on the other hand, can provide a high-quality result by using a single profile in most cases. Figure 7.5 compares the training overheads of GPR and ReIM. While GPR does at least 10 runs, ReIM uses a single application profile. Effectively, the training overhead of ReIM is observed to be only 10-20% of that of GPR.

SVM is the only instance where ReIM needs to use a second profile for evaluation due to a lack of *full GC* events in the initial profile provided. We touched upon this issue earlier in Section 5.4; the next subsection will present a further analysis.

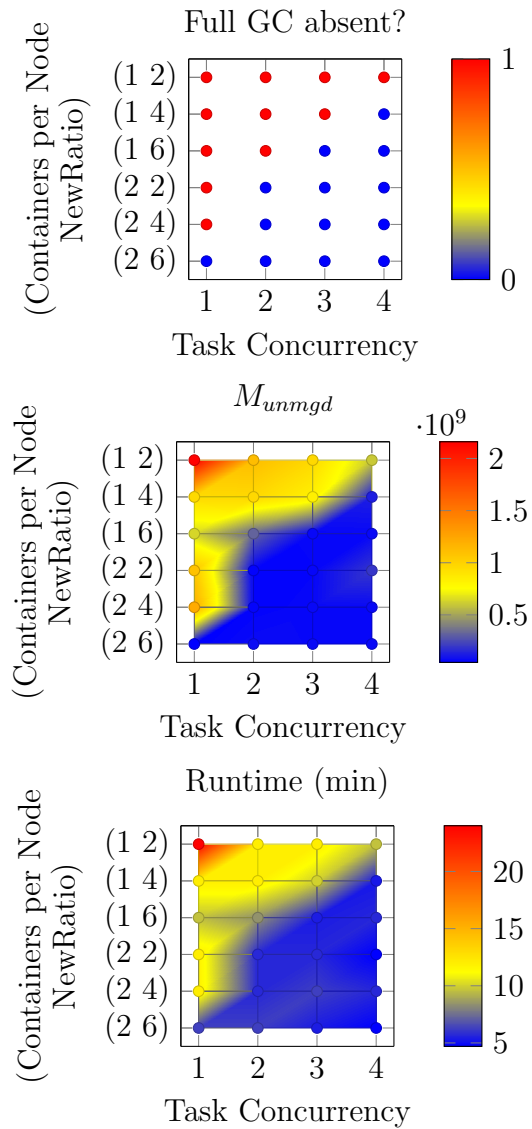


FIGURE 7.6: Understanding sensitivity of ReIM recommendations to the initial profile. When the initial profile contains no full GC events, maximum Old value is used to estimate task memory. This results in an over-estimate of memory requirements and sub-optimal recommendations. The profiles with full GC events, on the other hand, result in more accurate estimations and close-to-optimal recommendations.

7.5.4 Sensitivity to initial profile

We use the `SVM` application to study sensitivity to initial profile. `SVM` tasks use a small amount of memory because of small-sized partitions (see Table 7.1). For sufficiently large `Heap` values, this memory can be collected by young GC events with very few objects tenuring to `Old`. Further, the cache storage requirement is only about 50% of `Heap`, resulting in a low pressure on `Old` pool.

We ran 24 configurations varying `Task Concurrency`, `Containers per Node`, and `NewRatio` parameters. `Cache Capacity` is set to its default value of 0.6. Figure 7.6 shows the results. The configurations with a low number of containers (or higher heap size), low task concurrency, and low `NewRatio`, shown in the top left quadrant in the plots, are the instances of no full GC events. We set `ReIM` to use maximum old generation pool size to estimate task memory requirements in such cases. The second plot shows that we over-estimate the memory requirements by up to 2 orders of magnitude compared to the cases where full GC events are observed. The runtime of the recommendations made on each point shows the same behavior. Two lessons are learnt from this analysis:

1. `ReIM` requires full GC events in the input profile.

While the plots show the problem, they provide the solution as well. In cases of no full GC events, `ReIM` bypasses its white-box modeling approach and simply recommends a new configuration to generate a new profile using the following heuristics:

- Increase number of `Containers per Node` (lower `Heap Size`)
- Increase `Task Concurrency`
- Increase `NewRatio`

Each of the changes increases memory pressure and is expected to generate a profile with full GC events. It should be noted that even if the configuration results in *out-of-memory* errors, `ReIM` can still infer task memory requirements from the profile.

Table 7.8: Unique recommendations made by RelM invoked 16 times on each application, with each invocation made using a distinct initial profile configuration

Application	Containers per Node	Task Concurrency	Cache Capacity	Shuffle Capacity	New Ratio
WordCount	4	2	0	.2	1
Sort	4	1	0	.2	1
	4	2	0	.2	1
	3	2	0	.2	1
K-means	2	2	.7	0	4
	1	3	.7	0	4
	1	4	.7	0	4
	1	5	.75	0	5
SVM	3	2	.5	.05	2
	2	3	.5	.05	2
	2	4	.5	.07	2
PageRank	2	1	.45	0	4
	2	1	.4	0	7
	1	1	.5	0	3

2. RelM is robust for input profiles containing full GC events.

It can be observed from Figure 7.6 that the runtimes exhibited by the recommendations made from the input profiles containing full GC events are clustered around 5 minutes. In fact, there are only 3 unique recommendations in the bottom right quadrant differing only slightly in terms of Task Concurrency and Cache Capacity values. Table 7.8 shows the recommendations. It can be noticed that the number of unique recommendations is very small for every application.

We include a plot in Figure 7.7 showing the variability of estimated values for Code Overhead (M_{init}) and Task Unmanaged (M_{unmngd}) pools using different profiles containing full GC events. It can be noticed that the estimated memory requirements have very little variance. The algorithm, as a result, recommends the same (with minor changes) configuration no matter where we start.

7.5.5 Ranking by utility scores

RelM enumerates each candidate container size and ranks the recommendations using

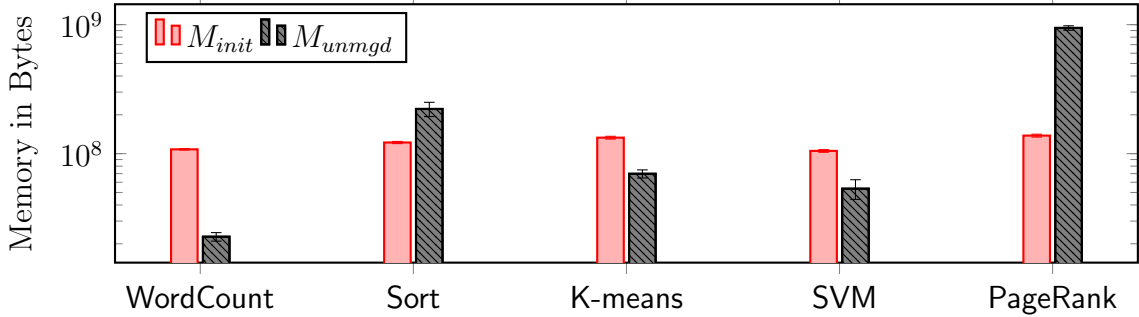


FIGURE 7.7: Analyzing sensitivity to initial profile by invoking ReIM with 16 initial profiles, each run with a different configuration. Error bars indicate standard error of the mean.

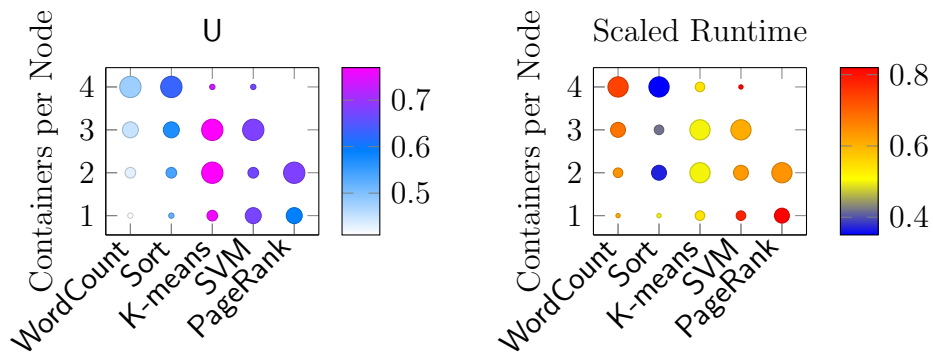


FIGURE 7.8: Evaluating accuracy of configurations recommended by ReIM. Bubble size indicates rank of the point in its column. Higher bubble sizes signify higher U values and lower runtimes.

utility score U . The utility score corresponds to the total memory utilization of the pools Code Overhead, Cache Storage, Task Shuffle, and Task Unmanaged. It should be noted that the utility score does not correspond to the expected **Heap** utilization because the formulation does not factor in garbage objects in **Heap**. The expectation, however, is that configuration with a higher utility score will perform better because of higher memory provisioned for internal pools.

We verify our intuition by plotting the U scores and the runtimes for the configurations considered while tuning our test applications. Figure 7.8 shows the results. The bubble size for a configuration is set to denote its rank relative to the other configurations considered for the application under test. For example, the configu-

ration recommended for `WordCount` with 4 Containers per Node is ranked the best, the one with 3 containers the second best, and so on. The ranks obtained based on runtimes indicate a direct correlation to the ranks based on U values. The instances of inversions are the cases where the values are very close to each other (indicated by color legends) and do not affect the quality of results significantly. In practice, the end user could be presented multiple configuration options to choose from all having U values close to the best.

7.6 Need for Database Performance Data Scientists

In this chapter, we studied the problem of autotuning the memory allocation for data analytics applications using a state-of-the-art, AI-driven, black-box approach and our new empirically-driven, white-box solution called ReIM. We showed how ReIM provides better quality results (in terms of the desired objectives of low wall-clock time and performance reliability) with minimal overheads. ReIM's superior performance highlights that tuning algorithms developed by *Database Performance Data Scientists* who combine an understanding of the underlying database platform with the ability to develop data-driven algorithms must not be overlooked while building autonomous/self-driving data processing systems.

Conclusion

This dissertation addressed an important problem of making modern data analytics platforms more autonomous. We designed and developed THOTH, a data-centric platform for multi-system analytics clusters. THOTH follows a *profile-predict-optimize* approach to develop multiple cluster management applications towards automated performance tuning. Presence of multiple “one-size” systems and a need to support multi-tenant workloads are the two main challenges we identified and addressed while building THOTH. We focused on memory management because the modern datacenters are seeing an increasing use of memory in data processing due to growing sizes and speeds of memory effectively making memory the most important factor impacting performance of the systems.

An important use case of memory during data processing is as a data cache. Modern data analytics platforms treat the data cache a first class citizen wherein a user can put cache directives on datasets which they want to reuse. In presence of a multi-tenant architecture, it opens up a possibility of shared cache optimization. Our first contribution to THOTH is a policy for *fair* cache allocation in presence of such multi-tenant workloads. We developed a novel fairness model that incorporates not

only the more standard notions of Pareto-efficiency and sharing incentive, but also envy-freeness added via the notion of *core* from cooperative game theory. The fair allocation algorithms we built using this model were implemented in a system called ROBUS. With ROBUS, we added an ability to automatically tune tenant workload processed in small online batches with appropriate cache directives. This led to fairness in terms of the speedups to performance experienced by tenants workloads.

Memory management decisions are taken at multiple levels during data processing given the presence of multiple systems managing memory, characteristic of DBMS⁺ architectures. This makes tuning memory allocation a challenging task. We identified interactions among the multi-level decisions using a systematic empirical study. This study showed the opportunities for performance optimizations by jointly tuning multi-level memory configuration options. We approached this problem from two angles: (i) A relatively black-box modeling using standard machine learning approaches assisted with system internal knowledge, and (ii) An empirically-driven white-box approach. The first approach showed that with little system internal profiling, we can easily double the speed of tuning compared to a pure black-box exploration. The second approach went a little further: We showed that, given the same monitoring information used in the first approach, a purely white-box algorithm can tune memory configurations in a small fraction of time compared to the state-of-the-art tuning techniques. While the algorithms we developed can make a tuning run lightning fast, automatically scheduling the tuning runs and identifying a right representative set of application workloads for tuning are some of the challenges that need to be addressed in the future.

Building autonomic data processing systems has taught us many lessons. Identifying interactions between the many system components is a critical step. We have taken a massive step towards this by developing the THOTH data-driven platform. The database-centric view in THOTH presents multiple opportunities for making the

data processing systems more autonomous such as automatic root cause analysis and an automated physical design tuning for data storage. In another contribution, we have undertaken a major effort towards developing a benchmarking service that can help with the automation process. Our benchmarking service, called BigFrame, creates benchmarks most suited to the needs of modern data analytics enterprise by using inputs from cluster administrators. As we demonstrated in this thesis, using BigFrame service in conjunction with the database-centric view in THOTH provides a rich vein of information which can be exploited by the system tuning models. More effort needs to be directed in future towards enhancing the benchmarking service to target the evolving systems stack.

Appendix A

BigFrame Benchmarking Service

A.1 Introduction

To meet the needs of a data-driven world, data processing industry is looking for end-to-end solutions that enable efficient and effective processing of large datasets. These solutions must now support deep analytic workflows (DAWs) which go from processing raw data all the way to visualizing useful insights extracted from the data. These workflows need to support diversity in data as well as computations. Modern datasets contain a combination of structured, semi-structured, and unstructured data that users want to analyze together. In terms of computations, there are usually multiple “one-size” systems employed for different kinds of processing needed by the workflows.

Behavioral targeting (BT) is one illustrative example of a DAW [152]. BT enables customized Web pages or advertisements to be shown to each user based on her past and current interactions with one or more websites. BT involves learning statistical machine-learning models for possibly millions of users from raw data sources such as: (i) impression logs, (ii) click logs, (iii) search logs, (iv) social media interactions,

and (v) social media influence. These models may use as input historical profiles and recent activity of users in order to generate customized content.

We propose a benchmarking-as-a-service solution, called BigFrame, that enables users to instantiate different benchmarks suitable to their needs. The user specification includes the following:

- **Data Volume:** Initial size of data and size referenced by queries
- **Query Volume:** The workload queries
- **Data Variety:** Proportion of data in formats including relational, text, graph, and time series
- **Query Variety:** Micro queries or application-based macro queries
- **Data Velocity:** Refresh patterns for data
- **Query Velocity:** Exploratory or continuous queries

Given the specification, BigFrame produces data and metadata for initial load as well as data refresh patterns. It also comes up with a number of query streams and evaluation metrics most suitable to user needs. For an ease in usability, BigFrame provides standard templates for common benchmarking use cases such as evaluation of scale-up relational data warehouse. We develop one example DAW in BigFrame which represents many common data analytics use cases.

A.2 A Sample Benchmark

We pick a business intelligence application domain that combines traditional analysis of e-commerce data with modern analysis of social media data. The domain is inspired from a real-life application workflow used in HP labs [153].

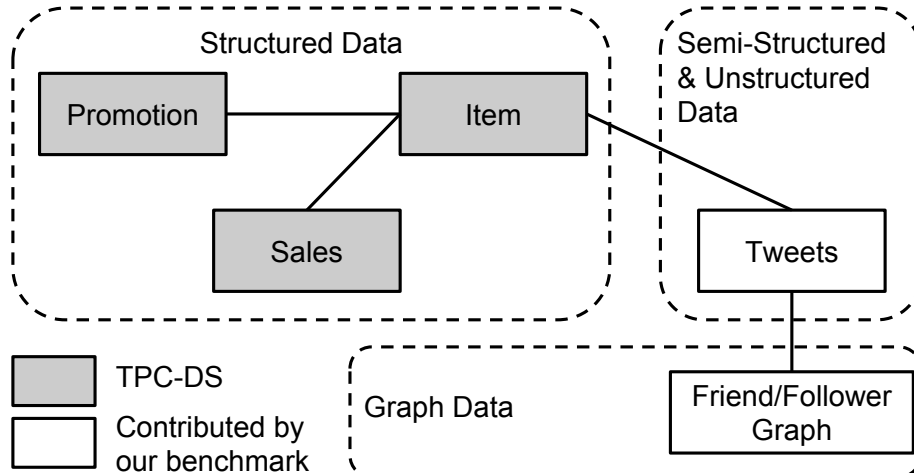


FIGURE A.1: Data model of our benchmark DAW

The benchmark DAW we create combines the processing of structured data (sales records), semi-structured and free-form text data (tweet metadata and text), as well as graph data (friend-follower graph from social media). Figure A.1 shows the data model of our benchmark DAW. We will first describe these data sources and then describe the DAW.

A.2.1 Structured Data

The structured data in our benchmark DAW consists of one logical *Sales* table, which represents data from three physical sources of sales, and two dimension tables. These tables are part of the popular TPC-DS benchmark:

- The **Item** dimension table stores information about products. The relevant attributes include the identifier and name of each product.
- The **Promotion** dimension table stores information about promotions on different products. The relevant attributes for each promotion include its identifier, the identifier of the promoted product, as well as the start and end dates of the promotion.
- **Sales** is a logical fact table storing records of sales done via the web (`web_sales`

fact table), catalog (catalog_sales fact table), and stores (store_sales fact table). The relevant attributes include identifier of the product sold, as well as the price, quantity sold, and date of sale.

We use TPC-DS’s data generator to generate data for all the above tables.

A.2.2 Semi-Structured and Unstructured Data

Tweets represent the semi-structured and unstructured data in our benchmark DAW. Tweets can be obtained by calling the Twitter streaming API [154]. Each tweet is a semi-structured JSON document that includes an unstructured text string as well as two important nested objects—*user* and *entities*—as shown in the example tweet in Listing A.1.

```
1 {
2   "created_at": "Thu Apr 06 15:24:15 +0000 2017",
3   "id": 850006245121695744,
4   "text": "Galaxy S9+ is a wonderful phone!",
5   "user": {
6     "id": 2244994945,
7     "name": "Bob",
8     "location": "internet",
9   },
10  "entities": {
11    "hashtags": ["Galaxy S9+"],
12    "user_mentions": [{"id": 238374665}],
13  }
14 }
```

Listing A.1: A sample Tweet

For our benchmark DAW, we generate synthetic Tweets for two important reasons:

- We can generate data related to dimension data in TPC-DS.
- We can control the data distribution as well as scale of data as needed, especially to match the scaling of the TPC-DS data.

As an example, we set 20% of the tweet data to have a mention of products, and 20% of the products mentioned are promoted products. The product name is put into the hashtags array inside the *entities* object in the tweet JSON. We generate

tweet text by picking, uniformly at random, one among a set of predefined words representing user sentiment about the product (e.g., “wonderful” in our example).

A.2.3 Graph Data

The graph data in our benchmark DAW represents the friend-follower relationship among users. An edge in this graph represents a person (the follower) following the tweets of another person (the friend). A real-life version of this graph can be created with a simpler crawler that uses Twitter’s API. Considering the need to correlate the graph data to other data in the system, we use a synthetic graph generator that generate a friend-follower relationship graph based on Kronecker graph model [155]. The graph generator can be seeded with a sample of the real Twitter friend-follower graph. In this case, the generated graph will maintain most of the properties of the real graph.

The default benchmark settings we use generate 2TB data in total with the proportion Tweet:Relational:Graph set to 100:10:1. This proportion is chosen based on studies that establish that over 90% of enterprise data will be semi-structured or unstructured [156].

A.2.4 The Benchmark DAW

The benchmark DAW we create over the data model just discussed is pictured in Figure 2.4. We describe the steps in detail next.

1. The first stage uses relational processing finds the products that were promoted during a time frame of interest for deep analysis.
2. The second stage works on semi-structured data to find all the tweets that mentioned a promoted product during its promotion period of the product.

3. For each tweet found in the previous step, we identify the sentiment being expressed in the tweet by carrying out a sentiment analysis proposed in [153].
4. Furthermore, we weigh the sentiment based on the degree of influence that this user holds in the system. That is, more influential users have higher weights on their sentiment. Our benchmark DAW uses the state-of-the-art *Twitter-Rank* [157] algorithm to compute each user's influence score on each product. The stage uses a PageRank-like graph processing algorithm, where a user's influence is continuously aggregated from all the followers until convergence.
5. Finally, the sales and weighted sentiment scores are combined together to generate the final output of the DAW.

Bibliography

- [1] “BigFrame: Benchmarking Service for Big Data Analytics, <https://github.com/bigframeteam/BigFrame>.”
- [2] “DataAge 2025 - The Digitization of the World — Seagate US <https://bit.ly/2tPWai6>.”
- [3] L. Cao, “Data science: A comprehensive overview,” *ACM Comput. Surv.*, vol. 50, pp. 43:1–43:42, June 2017.
- [4] M. Stonebraker and U. Cetintemel, ““one size fits all”: An idea whose time has come and gone,” in *Proceedings of the 21st International Conference on Data Engineering, ICDE '05*, (Washington, DC, USA), pp. 2–11, IEEE Computer Society, 2005.
- [5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12*, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.
- [6] “The Exploding Digital Universe <https://bit.ly/2JcdZCW>.”
- [7] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, “Zephyr: Live migration in shared nothing databases for elastic cloud platforms,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11*, (New York, NY, USA), pp. 301–312, ACM, 2011.
- [8] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, “Hadoop’s adolescence: An analysis of hadoop usage in scientific workloads,” *Proc. VLDB Endow.*, vol. 6, pp. 853–864, Aug. 2013.
- [9] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, “Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters,” *IEEE Transactions on Services Computing*, vol. 12, pp. 91–104, Jan 2019.

- [10] H. Lim, Y. Han, and S. Babu, “How to fit when no one size fits,” in *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.
- [11] H. Herodotou, *Automatic Tuning of Data-intensive Analytical Workloads*. PhD thesis, Duke University, Durham, NC, USA, 2012. AAI3504075.
- [12] M. Kunjir, P. Kalmegh, and S. Babu, “Thoth: Towards managing a multi-system cluster,” *Proc. VLDB Endow.*, vol. 7, pp. 1689–1692, Aug. 2014.
- [13] M. Kunjir, B. Fain, K. Munagala, and S. Babu, “ROBUS: fair cache allocation for data-parallel workloads,” in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD ’17*, (New York, NY, USA), pp. 219–234, ACM, 2017.
- [14] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K page replacement algorithm for database disk buffering,” in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD ’93*, (New York, NY, USA), pp. 297–306, ACM, 1993.
- [15] “Spark: Lightning fast cluster computing, <http://spark.incubator.apache.org/index.html>.”
- [16] M. Kunjir and S. Babu, “Thoth in action: Memory management in modern data analytics,” *Proc. VLDB Endow.*, vol. 10, pp. 1917–1920, Aug. 2017.
- [17] “Understanding Memory Management In Spark For Fun And Profit <https://bit.ly/2xZdfsw>.”
- [18] M. Kunjir, “Guided bayesian optimization to autotune memory-based analytics,” in *2019 IEEE 35th International Conference on Data Engineering Workshops*, April 2019.
- [19] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC ’13*, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.
- [20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in *Proceedings of the 8th USENIX Conference on Networked*

Systems Design and Implementation, NSDI'11, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2011.

- [21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: Flexible, scalable schedulers for large compute clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 351–364, ACM, 2013.
- [22] A. Rabkin and R. Katz, “Chukwa: A system for reliable large-scale log collection,” in *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, (Berkeley, CA, USA), pp. 1–15, USENIX Association, 2010.
- [23] “Announcing suro: Backbone of netflix’s data pipeline, <http://techblog.netflix.com/2013/12/announcing-suro-backbone-of-netflixs.html>.”
- [24] “Splunk, <http://www.splunk.com/>.”
- [25] “Ganglia monitoring system, <http://ganglia.sourceforge.net/>.”
- [26] Y. Chen, S. Alspaugh, and R. Katz, “Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads,” *Proc. VLDB Endow.*, vol. 5, pp. 1802–1813, Aug. 2012.
- [27] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, “SAP HANA database: Data management for modern business applications,” *SIGMOD Rec.*, vol. 40, pp. 45–51, Jan. 2012.
- [28] H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and I. Stoica, “Tachyon: Memory throughput I/O for cluster computing frameworks,” *LADIS*, vol. 18, p. 1, 2013.
- [29] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, “PACMan: Coordinated memory caching for parallel jobs,” NSDI'12, (Berkeley, CA, USA), pp. 20–20, USENIX Association, 2012.
- [30] H. Gupta and I. S. Mumick, “Selection of views to materialize in a data warehouse,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 24–43, 2005.
- [31] J. Yang, K. Karlapalem, and Q. Li, “Algorithms for materialized view design in data warehousing environment,” VLDB '97, (San Francisco, CA, USA), pp. 136–145, Morgan Kaufmann Publishers Inc., 1997.

- [32] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, “Automated selection of materialized views and indexes in SQL databases,” VLDB ’00, (San Francisco, CA, USA), pp. 496–505, Morgan Kaufmann Publishers Inc., 2000.
- [33] Y. Kotidis and N. Roussopoulos, “DynaMat: A dynamic view management system for data warehouses,” SIGMOD ’99, (New York, NY, USA), pp. 371–382, ACM, 1999.
- [34] J. Nash, “The bargaining problem,” *Econometrica*, vol. 18, no. 2, pp. 155–162, 1950.
- [35] K. Jain and V. V. Vazirani, “Eisenberg-gale markets: Algorithms and structural properties,” in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pp. 364–373, ACM, 2007.
- [36] E. Budish, “The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes,” *Journal of Political Economy*, vol. 119, no. 6, pp. 1061–1103, 2011.
- [37] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types,” NSDI’11, (Berkeley, CA, USA), pp. 323–336, USENIX Association, 2011.
- [38] D. C. Parkes, A. D. Procaccia, and N. Shah, “Beyond dominant resource fairness: Extensions, limitations, and indivisibilities,” EC ’12, (New York, NY, USA), pp. 808–825, ACM, 2012.
- [39] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan, “Rate control for communication networks: Shadow prices, proportional fairness and stability,” *The Journal of the Operational Research Society*, vol. 49, no. 3, pp. pp. 237–252, 1998.
- [40] F. Kelly, L. Massoulié, and N. Walton, “Resource pooling in congested networks: proportional fairness and product form,” *Queueing Systems*, vol. 63, no. 1-4, pp. 165–194, 2009.
- [41] S. Im, J. Kulkarni, and K. Munagala, “Competitive algorithms from competitive equilibria: Non-clairvoyant scheduling under polyhedral constraints,” STOC ’14, pp. 313–322, 2014.
- [42] A. L. Stolyar, “On the asymptotic optimality of the gradient scheduling algorithm for multiuser throughput allocation,” *Oper. Res.*, vol. 53, no. 1, pp. 12–25, 2005.

- [43] M. Andrews, “Instability of the proportional fair scheduling algorithm for hdr,” *Wireless Communications, IEEE Transactions on*, vol. 3, no. 5, pp. 1422–1426, 2004.
- [44] R. J. Aumann, “Markets with a continuum of traders,” *Econometrica: Journal of the Econometric Society*, pp. 39–50, 1964.
- [45] G. Debreu and H. Scarf, “A limit theorem on the core of an economy,” *International Economic Review*, vol. 4, no. 3, pp. pp. 235–246, 1963.
- [46] I. Kash, A. D. Procaccia, and N. Shah, “No agent left behind: Dynamic fair division of multiple resources,” in *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems, AAMAS ’13*, (Richland, SC), pp. 351–358, International Foundation for Autonomous Agents and Multiagent Systems, 2013.
- [47] A. Abdulkadiroglu and T. Sonmez, “Random serial dictatorship and the core from random endowments in house allocation problems,” *Econometrica*, vol. 66, no. 3, pp. 689–701, 1998.
- [48] A. Bogomolnaia and H. Moulin, “A new solution to the random assignment problem,” *Journal of Economic Theory*, vol. 100, no. 2, pp. 295–328, 2001.
- [49] E. Budish, Y.-K. Che, F. Kojima, and P. Milgrom, “Designing random allocation mechanisms: Theory and applications,” *American Economic Review*, vol. 103, no. 2, pp. 585–623, 2013.
- [50] H. Varian, “Two problems in the theory of fairness,” *Journal of Public Economics*, vol. 5, no. 3-4, pp. 249–260, 1976.
- [51] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, “FairRide: Near-optimal, fair cache sharing,” in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, (Santa Clara, CA), pp. 393–406, USENIX Association, Mar. 2016.
- [52] D. Gillies, *Some Theorems on n-Person Games*. PhD thesis, Princeton University, 1953.
- [53] H. E. Scarf, “The core of an n person game,” *Econometrica*, vol. 35, no. 1, pp. pp. 50–69, 1967.
- [54] D. K. Foley, “Lindahls solution and the core of an economy with public goods,” *Econometrica: Journal of the Econometric Society*, pp. 66–72, 1970.

- [55] A. Mas-Colell and J. Silvestre, “Cost share equilibria: A lindahl approach,” *Journal of Economic Theory*, vol. 47, no. 2, pp. 239–256, 1989.
- [56] M. Kaneko, “The ratio equilibrium and a voting game in a public goods economy,” *Journal of Economic Theory*, vol. 16, no. 2, pp. 123–136, 1977.
- [57] H. W. Kuhn and A. W. Tucker, “Nonlinear programming,” in *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, (Berkeley, Calif.), pp. 481–492, University of California Press, 1951.
- [58] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, “A quantitative measure of fairness and discrimination for resource allocation in shared computer systems,” tech. rep., Digital Equipment Corporation, Sept. 1984.
- [59] S. Arora, E. Hazan, and S. Kale, “The multiplicative weights update method: A meta algorithm and applications,” *Theory of Computing*, vol. 8, pp. 121–164, 2005.
- [60] Y. Freund and R. E. Schapire, “Adaptive game playing using multiplicative weights,” *Games and Economic Behavior*, vol. 29, no. 1–2, pp. 79 – 103, 1999.
- [61] Y. Cai, C. Daskalakis, and S. M. Weinberg, “Optimal multi-dimensional mechanism design: Reducing revenue to welfare maximization,” in *FOCS*, 2012.
- [62] A. Bhargat, S. Gollapudi, and K. Munagala, “Optimal auctions via the multiplicative weight method,” in *Proceedings of the fourteenth ACM conference on Electronic commerce*, pp. 73–90, ACM, 2013.
- [63] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, “Choosy: Max-min fair sharing for datacenter jobs with constraints,” in *EuroSys*, pp. 365–378, ACM, 2013.
- [64] D. Chakrabarty, A. Mehta, and V. Vazirani, “Design is as easy as optimization,” in *Automata, Languages and Programming* (M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, eds.), vol. 4051 of *Lecture Notes in Computer Science*, pp. 477–488, Springer Berlin Heidelberg, 2006.
- [65] M. Berkelaar, K. Eikland, and P. Notebaert, *lpsolve : Open source (Mixed-Integer) Linear Programming system*.
- [66] “Hadoop fair scheduler, http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.”

- [67] P. Agrawal, D. Kifer, and C. Olston, “Scheduling shared scans of large data files,” *Proc. VLDB Endow.*, vol. 1, pp. 958–969, Aug. 2008.
- [68] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey, “Miso: Souping up big data query processing with a multistore system,” *SIGMOD ’14*, (New York, NY, USA), pp. 1591–1602, ACM, 2014.
- [69] D. Jacobs, S. Aulbach, and T. U. München, “Ruminations on multi-tenant databases,” in *BTW Proceedings, volume 103 of LNI*, pp. 514–521, GI, 2007.
- [70] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, “Multi-tenant databases for software as a service: Schema-mapping techniques,” *SIGMOD ’08*, (New York, NY, USA), pp. 1195–1206, ACM, 2008.
- [71] V. Narasayya, S. Das, M. Syamala, S. Chaudhuri, F. Li, and H. Park, “A demonstration of sqlvm: Performance isolation in multi-tenant relational database-as-a-service,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, *SIGMOD ’13*, (New York, NY, USA), pp. 1077–1080, ACM, 2013.
- [72] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, “Workload-aware database monitoring and consolidation,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, *SIGMOD ’11*, (New York, NY, USA), pp. 313–324, ACM, 2011.
- [73] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra, “Adaptive self-tuning memory in DB2,” in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, *VLDB ’06*, pp. 1081–1092, VLDB Endowment, 2006.
- [74] “The self managing database: Automatic SGA memory management. Oracle white paper, November 2003. <http://goo.gl/7in5Vb>.”
- [75] “REST job server for Apache Spark. <https://github.com/spark-jobserver/spark-jobserver>.”
- [76] “Livy job server. <https://github.com/cloudera/livy>.”
- [77] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri, “Sharing buffer pool memory in multi-tenant relational database-as-a-service,” *Proc. VLDB Endow.*, vol. 8, pp. 726–737, Feb. 2015.
- [78] “Amazon EC2 - virtual server hosting. <https://aws.amazon.com/ec2/>.”

- [79] C. A. Waldspurger, “Memory resource management in VMware ESX server,” *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 181–194, Dec. 2002.
- [80] “Apache hadoop, <http://hadoop.apache.org/>.”
- [81] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: Fair scheduling for distributed computing clusters,” SOSP ’09, (New York, NY, USA), pp. 261–276, ACM, 2009.
- [82] “Discardable distributed memory: Supporting memory storage in HDFS, <http://hortonworks.com/blog/ddm/>.”
- [83] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, “The case for RAMClouds: Scalable high-performance storage entirely in DRAM,” *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 92–105, Jan. 2010.
- [84] “Presto: Interacting with petabytes of data at Facebook. <http://on.fb.me/LnFVAM>.”
- [85] “Building the Periscope Data Cache with Amazon Redshift. <https://goo.gl/c68EzG>.”
- [86] J. Goldstein and P.-A. Larson, “Optimizing queries using materialized views: A practical, scalable solution,” SIGMOD ’01, (New York, NY, USA), pp. 331–342, ACM, 2001.
- [87] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, “Materialized view selection and maintenance using multi-query optimization,” SIGMOD ’01, (New York, NY, USA), pp. 307–318, ACM, 2001.
- [88] N. Bruno and S. Chaudhuri, “An online approach to physical design tuning,” in *ICDE*, 2007.
- [89] I. Elghandour and A. Aboulnaga, “Restore: Reusing results of mapreduce jobs,” *Proc. VLDB Endow.*, vol. 5, pp. 586–597, Feb. 2012.
- [90] T. K. Sellis, “Multiple-query optimization,” *ACM Trans. Database Syst.*, vol. 13, pp. 23–52, Mar. 1988.

- [91] M. Zukowski, S. Héman, N. Nes, and P. Boncz, “Cooperative scans: Dynamic bandwidth sharing in a DBMS,” *VLDB '07*, pp. 723–734, VLDB Endowment, 2007.
- [92] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, “MRShare: Sharing across multiple queries in mapreduce,” *Proc. VLDB Endow.*, vol. 3, pp. 494–505, Sept. 2010.
- [93] X. Wang, C. Olston, A. D. Sarma, and R. Burns, “CoScan: Cooperative scan sharing in the cloud,” in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, (New York, NY, USA), pp. 11:1–11:12, ACM, 2011.
- [94] “TPC-DS benchmark, <http://www.tpc.org/tpcds/>.”
- [95] “TPC-H benchmark, <http://www.tpc.org/tpch/>.”
- [96] J. Gray, K. Baclawski, P. Sundaresan, and S. Englert, “Quickly generating billion-record synthetic databases,” Association for Computing Machinery, Inc., January 1994.
- [97] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou, “Workload characterization on a production hadoop cluster: A case study on Taobao,” in *IISWC'12*, pp. 3–13, 2012.
- [98] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, “Reservation-based scheduling: If you’re late don’t blame us!,” in *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, (New York, NY, USA), pp. 2:1–2:14, ACM, 2014.
- [99] “ROBUS Cache Planner, <https://github.com/dukedbgroup/ROBUS>.”
- [100] “Java garbage collection basics. <https://bit.ly/2N8Jy0p>.”
- [101] “Spark configuration. <https://bit.ly/2rXR4NK>.”
- [102] “Flink configuration. <https://bit.ly/2P03eRM>.”
- [103] “Amazon EMR Documentation. <https://amzn.to/2zrpNtt>.”
- [104] “How-to: Tune your apache spark jobs (part 2) <https://bit.ly/2BuyZyS>.”

- [105] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang, “Automatic database management system tuning through large-scale machine learning,” in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 1009–1024, 2017.
- [106] S. Duan, V. Thummala, and S. Babu, “Tuning database configuration parameters with ituned,” *PVLDB*, vol. 2, no. 1, pp. 1246–1257, 2009.
- [107] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, “Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, (Boston, MA), pp. 469–482, USENIX Association, 2017.
- [108] E. Kwan, S. Lightstone, K. B. Schiefer, A. J. Storm, and L. Wu, “Automatic database configuration for db2 universal database: Compressing years of performance expertise into seconds of execution,” in *BTW*, 2003.
- [109] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, “Starfish: A self-tuning system for big data analytics,” in *In CIDR*, pp. 261–272, 2011.
- [110] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, “Ernest: Efficient performance prediction for large-scale advanced analytics,” in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, (Santa Clara, CA), pp. 363–378, USENIX Association, 2016.
- [111] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection.” <http://snap.stanford.edu/data>, June 2014.
- [112] “Deep dive: Apache spark memory management <https://bit.ly/2C2x1YH>.”
- [113] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel, “Don’t cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 97–109, USENIX Association, 2017.
- [114] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “Graphx: Graph processing in a distributed dataflow framework,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI’14*, (Berkeley, CA, USA), pp. 599–613, USENIX Association, 2014.

- [115] “Java Management Extensions (JMX). <https://bit.ly/2KIvbNn>.”
- [116] “Intel’s performance analysis tool. <https://bit.ly/2FHpYGI>.”
- [117] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback, “Self-tuning database technology and information services: From wishful thinking to viable engineering,” in *Proceedings of the 28th International Conference on Very Large Data Bases, VLDB ’02*, pp. 20–31, VLDB Endowment, 2002.
- [118] J. Mockus, *Bayesian approach to global optimization: theory and applications*. Mathematics and its applications (Kluwer Academic Publishers).: Soviet series, Kluwer Academic, 1989.
- [119] P. Jamshidi and G. Casale, “An uncertainty-aware approach to optimal configuration of stream processing systems,” in *24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016, London, United Kingdom, September 19-21, 2016*, pp. 39–48, IEEE Computer Society, 2016.
- [120] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok, “Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems,” in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC ’18*, (Berkeley, CA, USA), pp. 893–907, USENIX Association, 2018.
- [121] C. Hsu, V. Nair, V. W. Freeh, and T. Menzies, “Arrow: Low-level augmented bayesian optimization for finding the best cloud VM,” in *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pp. 660–670, IEEE Computer Society, 2018.
- [122] C. E. Rasmussen, “Gaussian processes for machine learning,” MIT Press, 2006.
- [123] C. Ireland, “Fundamental concepts in the design of experiments,” *Technometrics*, vol. 7, no. 4, pp. 652–653, 1965.
- [124] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung, “A new approach to dynamic self-tuning of database buffers,” *Trans. Storage*, vol. 4, pp. 3:1–3:25, May 2008.
- [125] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, “A smart hill-climbing algorithm for application server configuration,” in *Proceedings of the 13th International Conference on World Wide Web, WWW ’04*, (New York, NY, USA), pp. 287–296, ACM, 2004.

- [126] L. Breiman, “Random forests,” *Mach. Learn.*, vol. 45, pp. 5–32, Oct. 2001.
- [127] V. Dalibard, M. Schaarschmidt, and E. Yoneki, “BOAT: Building auto-tuners with structured bayesian optimization,” in *Proceedings of the 26th International Conference on World Wide Web*, WWW ’17, (Republic and Canton of Geneva, Switzerland), pp. 479–488, International World Wide Web Conferences Steering Committee, 2017.
- [128] M. Hoffman, E. Brochu, and N. de Freitas, “Portfolio allocation for bayesian optimization,” in *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, UAI’11, (Arlington, Virginia, United States), pp. 327–336, AUAI Press, 2011.
- [129] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, “Taking the human out of the loop: A review of bayesian optimization,” *Proceedings of the IEEE*, vol. 104, pp. 148–175, 2016.
- [130] R. H. Byrd, P. Lu, and J. Nocedal, “A limited-memory algorithm for bound constrained optimization,” *SIAM Journal on Scientific Computing*, 1994.
- [131] “RelM Report. <https://users.cs.duke.edu/~mayuresh/relm-report.pdf>.”
- [132] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, “The hibench benchmark suite: Characterization of the mapreduce-based data analysis,” in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pp. 41–51, March 2010.
- [133] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [134] “What Ails Spark in production?. <https://goo.gl/mBZH7Z>.”
- [135] L. Xu, W. Dou, F. Zhu, C. Gao, J. Liu, H. Zhong, and J. Wei, “Experience report: A characteristic study on out of memory errors in distributed data-parallel applications,” in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 518–529, Nov 2015.
- [136] Z. Tan and S. Babu, “Tempo: Robust and self-tuning resource management in multi-tenant parallel databases,” *Proc. VLDB Endow.*, vol. 9, pp. 720–731, June 2016.

- [137] “Project Tungsten: Bringing Apache Spark closer to bare metal. <http://bitly.com/1KPPfBC>.”
- [138] “Juggling with bits and bytes. <https://bit.ly/2C1zYbN>.”
- [139] S. Chaudhuri and V. Narasayya, “Self-tuning database systems: A decade of progress,” in *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB ’07, pp. 3–14, VLDB Endowment, 2007.
- [140] S. Chaudhuri and V. R. Narasayya, “An efficient cost-driven index selection tool for microsoft sql server,” in *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB ’97, (San Francisco, CA, USA), pp. 146–155, Morgan Kaufmann Publishers Inc., 1997.
- [141] J. Rao, C. Zhang, N. Megiddo, and G. Lohman, “Automating physical database design in a parallel database,” in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’02, (New York, NY, USA), pp. 558–569, ACM, 2002.
- [142] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, “Automated selection of materialized views and indexes in sql databases,” in *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB ’00, (San Francisco, CA, USA), pp. 496–505, Morgan Kaufmann Publishers Inc., 2000.
- [143] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra, “Adaptive self-tuning memory in db2,” in *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB ’06, pp. 1081–1092, VLDB Endowment, 2006.
- [144] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood, “Automatic performance diagnosis and tuning in oracle,” in *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pp. 84–94, www.cidrdb.org, 2005.
- [145] T. Ye and S. Kalyanaraman, “A recursive random search algorithm for large-scale network parameter configuration,” *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 196–205, June 2003.
- [146] H. Herodotou, F. Dong, and S. Babu, “No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics,” SOCC ’11, (New York, NY, USA), pp. 18:1–18:14, ACM, 2011.

- [147] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung, “A new approach to dynamic self-tuning of database buffers,” *Trans. Storage*, vol. 4, pp. 3:1–3:25, May 2008.
- [148] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, “Using probabilistic reasoning to automate software tuning,” in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’04/Performance ’04, (New York, NY, USA), pp. 404–405, ACM, 2004.
- [149] C. Reiss, *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.
- [150] A. K. Austin, “Sharing a cake,” *The Mathematical Gazette*, vol. 66, no. 437, pp. 212–215, 1982.
- [151] “RelM Source code. https://github.com/dukedbgroup/global_log.”
- [152] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, S. Grell, O. Nano, and I. Santos, “Microsoft cep server and online behavioral targeting,” *Proc. VLDB Endow.*, vol. 2, pp. 1558–1561, Aug. 2009.
- [153] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal, “Optimizing analytic data flows for multiple execution engines,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’12, (New York, NY, USA), pp. 829–840, ACM, 2012.
- [154] “Twitter Streaming API <https://bit.ly/2AF30Bv>.”
- [155] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, “Kronecker Graphs: An Approach to Modeling Networks,” *Journal of Machine Learning Research*, vol. 11, pp. 985–1042, 2010.
- [156] F. Gens, “IDC Predictions 2012: Competing for 2020,” tech. rep., IDC, 2012.
- [157] J. Weng, E.-P. Lim, J. Jiang, and Q. He, “Twitterrank: Finding topic-sensitive influential twitterers,” in *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM ’10, (New York, NY, USA), pp. 261–270, ACM, 2010.

- [158] M. Kunjir, “Allocating shared resources in multi-tenant analytics clusters,” in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’15, (New York, NY, USA), ACM, 2015.
- [159] M. Kunjir, B. Fain, K. Munagala, and S. Babu, “ROBUS: fair cache allocation for multi-tenant data-parallel workloads,” *CoRR*, vol. abs/1504.06736, 2015.

Biography

Mayuresh Prakash Kunjir graduated with a Bachelors in Technology in 2008 from College of Engineering, Pune, India majoring in Computer Science. Post that, he received a Masters in Engineering from Indian Institute of Science, Bangalore while working at Database Systems research group in the department of Computer Science and Automation. His work included power optimization in database systems.

He did his doctoral studies in the department of Computer Science at Duke university from the year 2012 to 2019. His doctoral research focused on automation of resource management in Big data analytical clusters. His work is published in database and systems conferences and workshops [12,13,16,18,158,159]. In addition, he has contributed to a public software release [1].

He has had an extensive industry experience both as a software engineer as well as a researcher through multiple internships with companies like Microsoft, IBM, HP Labs, and RocketFuel.