Understanding and Managing the Performance Variation and Data Growth in Cloud Computing

Thesis for obtaining the title of Doctor of Engineering Science of the Faculty of Natural Science and Technology I of Saarland University

by
Jörg Schad, B.Sc.
Saarbrücken
May 2015

Day of Colloquium	15.04.2016
Dean of the Faculty	Univ.-Prof. Dr. Frank-Olaf Schreyer
Char of Committee	Prof. Dr. Gerhard Weikum
Reporters	
First Reviewer	Prof. Dr Jens Dittrich
Second Reviewer	Prof. Dr. Sebastian Michel
Academic Assistant	Dr. Alvaro Torralba

Acknowledgements

I would never have been able to finish this dissertation without the guidance of my committee members, and support help from friends..

I would like to express my deepest gratitude to my advisor, Prof. Dr. Jens Dittrich, for his guidance, support, and providing me with valuable discussions. Furthermore, I would like to thank Dr. Jorge-Arnulfo Quiané-Ruiz, who provided much guidance, and patiently helped develop new ideas.

I would also like to thank all fellow team member from the Information Systems Group for being great companions and providing support for the past several years.

Abstract
The topics of Cloud Computing and Big Data Analytics dominate today's IT landscape. This dissertation considers the combination of both and the resulting challenges. In particular, it addresses executing data intensive jobs efficiently on public cloud infrastructure, with respect to response time, cost, and reproducibility. We present an extensive study of performance variance in public cloud infrastructures covering various dimensions including micro-benchmarks and application-benchmarks, different points in time, and different cloud vendors. It shows that performance variance is a problem for cloud users even at the application level. It then provides some guidelines and tools on how to counter the effects of performance variance. Next, this dissertation addresses the challenge of efficiently processing dynamic datasets. Dynamic datasets, i.e., datasets which change over time, are a challenge for standard MapReduce Big Data Analytics as they require the entire dataset to be reprocessed after every change. We present a framework to deal efficiently with dynamic datasets inside MapReduce using different techniques depending on the characteristics of the dataset. The results show that we can significantly reduce reprocessing time for most use-cases. This dissertation concludes with a discussion on how new technologies such as container virtualization will affect the challenges presented here.

Zusammenfassung
Cloud Computing und die Verarbeitung großer Datenmengen sind allgegenwärtige Themen in der heutigen IT Landschaft. Diese Dissertation befasst sich mit der Kombination dieser beiden Technologien. Insbesondere werden die Problematiken mit effizienter und reproduzierbarer Verarbeitung von großen Datenmengen innerhalb von Public Cloud Angeboten betrachtet. Das Problem von variabler Rechenleistung bei Public-Cloud-Angeboten wird in einer ausführlichen Studie untersucht. Es wird gezeigt, dass sich die Varianz der Rechenleistung auf verschiedenen Leveln bis zur Applikation auswirkt. Wir diskutieren Ansätze um diese Varianz zu reduzieren und präsentieren verschiedene Algorithmus um homogene Rechenleistung in einem Rechnerverbund zu erreichen. Die Verarbeitung von großen, dynamischen Datenmengen mit heutigen MapReduce basierten Systeme ist relativ aufwändig, weil Änderungen der Daten eine Neuberechnung des gesamten Datensatzes erfordert. Solche Neuberechnungen können insbesondere in Cloud Umgebungen schnell zu hohen Kosten führen. Diese Dissertation präsentiert verschiedene Algorithmen zum effizienten Verarbeiten solcher dynamischen Datensets und ein System welches automatisch den fü das Datenset passenden optimalen Algorithmus auswählt. Wir schließen mit einer Diskussion welchen Einfluss neue Technologien wie Docker auf die hier präsentierten Probleme haben.

Contents

1 Introduction

1.1 Motivation

Cloud Computing is an omnipresent topic in today's IT landscape. Cloud computing platforms change the economics of computing by allowing users to pay only for the capacity that their applications actually need (the pay-as-you-go model) while letting vendors exploit the economic benefits of scale. Another important advantage for users is the enormous scalability of such cloud infrastructure. These characteristics allow users to spawn several thousand new servers within a few minutes, but also to tear down servers quickly once the the demand shrinks. The demand for such scalable infrastructures is growing especially for the problem of Big Data Analytics. In the past, companies have collected massive volumes of data including user related data, log data, sales data, and sensor data. Deriving the value from such data collections usually requires the use of large amounts of distributed computing resources such as storage, CPU and memory. Using such distributed infrastructures can be challenging as a developers has to deal with a number of problems, including data distribution, scheduling compute tasks or resources failing. MapReduce frameworks such as Hadoop offer a solution to these problems by abstracting these problems from the user and offering a simple to use interface. Amazon Elastic MapReduce is one prominent example of the successful combination of Cloud Computing and MapReduce by offering a simple to use service to users. For researchers the combination is equally attractive for running large scale experiments or analyses. However, there are still a number of challenges when performing MapReduce data analyses in a cloud environment:

1. Most cloud services rely on virtualized resources and therefore the same virtual machine can be executed on different physical hardware with different performance characteristics. Also cloud services usually serve multiple tenants on a single physical host. As a result **performance variance** is a common problem for users as it makes performance measurements hard to interpret and estimating the required amount of resources for a given task becomes very challenging.

2. MapReduce is especially sensitive to performance variance between different nodes in the cluster. Hence, the performance variance of different virtual machines cannot only lead to a variance in the actual MapReduce job runtime, but also **slow down** the MapReduce job in general. Also the upload times while transferring data into Hadoop's Distributed File System HDFS can suffer from such performance variance.

3. Typically, data is dynamic: it grows (appending new records) or it is altered (altering existing records). Using MapReduce, users typically have to run their jobs again over the entire dataset (often including several terabytes of data) every time the dataset is changed. Especially in a cloud setting this manner of **processing changing datasets** can lead to long response times and high costs.

1.2 Problem Definition

This thesis addresses the question "*How can users execute data intensive jobs, especially MapReduce jobs on public cloud infrastructure (e.g., Amazon EC2) efficiently with respect to response time, cost, and reproducibility?*". To answer this question we focus on the following problems:

1. How large is the performance variance on a micro-level (i.e., CPU, memory, I/O, and network performance) between different virtual machines of the same specification? Is this variance different between different cloud vendors? How much does performance of a single instance vary over time?

2. How much does this micro level variance result in application level performance variance?

3. Does the cluster heterogeneity caused by this performance variance effect the actual MapReduce job response time?

4. How can we process MapReduce jobs over a growing and changing (i.e., incremental) dataset efficiently in a cloud setting?

1.3 Publications

During the term of this project we published the following related papers:

[162] Jörg Schad, Jorge-Arnulfo Quiane-Ruiz, Jens Dittrich.
Elephant, Do not Forget Everything! Efficient Processing of Growing Datasets
IEEE Cloud 2013, Santa Clara, USA.

[109] Jens Dittrich, Jorge-Arnulfo Quiane-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, Jörg Schad.
Only Aggressive Elephants are Fast Elephants
VLDB 2012/PVLDB, Istanbul, Turkey.

[155] Jorge-Arnulfo Quiane-Ruiz, Christoph Pinkel, Jörg Schad, and Jens Dittrich.
RAFT at Work: Speeding-Up MapReduce Applications under Task and Node Failures (Demo Paper)
SIGMOD 2011, Athens, Greece.

[156] Jorge-Arnulfo Quiane-Ruiz, Christoph Pinkel, Jörg Schad, and Jens Dittrich.
RAFTing MapReduce: Fast Recovery on the Raft
ICDE 2011, Hannover.

[161] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiane-Ruiz.
Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance
VLDB 2010, Singapore.

[108] Jens Dittrich, Jorge-Arnulfo Quiane-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad.
Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)
VLDB 2010, Singapore.

[160] Jörg Schad.
Flying Yellow Elephant: Predictable and Efficient MapReduce in the Cloud
PhD Workshop VLDB 2010, Singapore.

1.4 Outline

In this thesis we examine the interaction between two trends in modern Information Technology: Cloud Computing and Big Data Analysis in particular the MapReduce framework.

First in Chapter 2 we provide the reader with the necessary background knowledge for the remainder of this thesis.

Next, we present the general problem of performance variance in Infrastructure as a Service (IaaS) Cloud Computing in Chapter 3. In this Section we measure the amount of variance by executing a number of micro and system level benchmarks on virtual machines provided by different cloud providers. In Chapter 3.6.1 we examine how this performance variance effects the actual response time of MapReduce jobs. We measure the effect of cluster heterogeneity caused by performance variance on the actual MapReduce job response time and HDFS upload times. In Chapter 4 we propose and evaluate different techniques for reducing the performance variance and obtain more homogeneous cluster.

In Chapter 5 we look at the problem of processing growing and changing datasets in MapReduce. Each dataset and job has different characteristics and so requires different techniques for efficient processing appends and changed in such datasets. We propose a system which can automatically choose the best of three different techniques for incremental processing.

1.5 Contributions.

Following the outline of this thesis, we can group our contributions into three different areas:

- Study of performance variance in cloud settings and its effects on workloads.
- Dealing efficiently with performance variance.
- Effective processing of dynamic datasets.

For each of these three areas we give in the following a detailed overview of our contributions.

1. **Study of performance variance in cloud settings and its effects on workloads.** In Section 3, we focus on the issue of performance variance in today's cloud systems, and exhaustively evaluate the performance of Amazon EC2 and other major cloud vendors.

   The major contributions are as follows.

   a) An exhaustive study of cloud performance variance. We perform performance measurements at different granularities:
      - single instances, which allow us to estimate the performance variance of a single virtual node
      - multiple instances, which allow us to estimate the performance variance of multiple nodes
      - different locations, which allow us to estimate the performance variance of different data centers
      - different vendors, which allow us to classify performance variance across different platforms (and therefore also across different virtualisation techniques)
      - different points in time (2010 and 2013), which allow us identify potential improvement over time
      - different parts of the cloud stack (IaaS and PaaS)

   b) We identify that performance can be divided in different bands and, among other factors, the heterogenous underlying physical hardware is a major source of performance variance. When considering potential reduction in performance variance between our measurements in 2010 and 2013 we can see improvements in some aspects, but actually an increase in variance in other aspects. The performance variance problem is present across all major vendors. We provide some hints to users to reduce the performance variability of their experiments.

   Section 3.6.1 considers the effects of the prior measured performance variance on MapReduce jobs. Here we can clearly identify a significant correlation between cluster heterogeneity (How much does performance differ between different cluster nodes?) and MapReduce job runtime.

2. **Dealing with performance variance.** In Chapter 4 we propose different techniques of how to obtain more homogenous clusters. We start by allocating a larger cluster and then successively terminate instances in order to reduce overall cluster heterogeneity. We show that especially for long running batch jobs the initial overhead of allocating more nodes is justified by reduced runtimes on a more homogene cluster.

   For selecting which instance to terminate we propose three different algorithms:

   - *Variance Driven Optimization* uses a set of micro-benchmarks to compute a heterogeneity score and then terminates the instance contributing most to this score.
   - *System Driven Optimization* avoids the expensive execution of the micro-benchmarks and uses information about systems instead. The strategy is to terminate instances having a different underlying system (e.g., processor or storage).
   - *Performance Driven Optimization* again uses micro-benchmarks to determine relative performance of cluster nodes. But instead of computing a heterogeneity score as the Variance Driven Optimization would do, it simply selects to terminate the node with the lowest relative performance.

   We show that with these techniques we can reduce the cluster variance and improve MapReduce job runtimes.

3. **Efficient incremental processing of changing datasets.** In Chapter 5 we address the challenge of processing growing and changing datasets in MapReduce. The problem with current MapReduce based systems is that they require the entire dataset to be reprocessed after any potentially minor change. We present Itchy, a framework to deal efficiently with dynamic datasets inside MapReduce. The main goal of Itchy is to execute incremental jobs by processing only relevant parts from the input of the initial job together with the input of incremental-jobs. Here we make the following contributions:

   a) We first identify different classes of incremental MR jobs. Each of these classes allows for different optimizations while processing growing datasets. We then propose three different techniques to efficiently process incremental-jobs, namely Itchy QMC (which stores provenance information), Itchy MO (which stores the Map Output,

hence intermediate data), and Itchy Merge (which combines the output of MapReduce jobs).

b) We show that choosing between Itchy QMC and Itchy MO is basically a tradeoff between storage overhead and runtime overhead. Thus, we present a decision model that allows Itchy to balance automatically the usage of QMCs and intermediate data to improve query performance. In this way Itchy can decide the best option for each incoming MapReduce job considering both job runtime and storage overhead.

c) We present a framework that implements the Itchy concepts in a manner invisible to users. This framework uses Hadoop and HBase to store and query both QMCs and intermediate data. The Itchy implementation includes many non-trivial performance optimizations to make processing incremental jobs efficient. In particular, Itchy runs map tasks in two map waves: one map wave to process the incremental dataset (containing the appended records) and one map wave to process the initial dataset (the input dataset to the first job). This allows Itchy to perform incremental-jobs in a single MapReduce job instead of two MapReduce jobs. As a result, Itchy avoids reading, writing, and shuffling growing datasets twice.

d) We present an extensive experimental evaluation of Itchy against the Hadoop MapReduce framework and Incoop. Our results demonstrate that Itchy significantly outperforms both Hadoop MapReduce and Incoop when dealing with incremental-jobs. We also show that Itchy incurs only negligible overhead when processing the first-job. Finally, we provide a detailed comparison between using QMCs or intermediate data in different settings.

2 Background

In this chapter we provide the reader with the necessary background knowledge for the remainder of this thesis. Therefore, we first give an overview of Cloud Computing in Section 2.1 followed by a discussion of the different virtualization techniques used in cloud computing in Section 2.2. Next, we provide an overview of the MapReduce framework in Section 2.3.

2.1 Cloud Computing

2.1.1 Overview

The services offered today by cloud providers are quite heterogeneous and range from entire infrastructures to specialized applications. In fact, the cloud market is growing rapidly in the number of vendors, but also in the number of concepts being marketed as cloud products. Judging from announcements for new IT products and services, it seems cloud computing can be found in almost any product today including Telekom Cloud, Amazon Cloud Player, Google Cloud Connect, Apple Cloud, Cloud Ready, SAP Cloud Solutions, or Datev Cloud Computing. As each vendor has a different understanding of the term Cloud Computing there exits no generally agreed-upon definition, but the following definition by NIST [60] covers most aspects:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

In the following, we look at certain aspects of this definition in more detail:

On-demand. Users can dynamically allocate and release the cloud resources as their application requires. This flexibility together with the pay-per-use

principle of most cloud vendors allows especially small companies to scale their IT infrastructure according to their current demand.

Network access. Cloud resources are usually provisioned via network and standard protocols. As a result users are often unaware of the exact location (i.e., data center, cluster or node) where the resources are actually allocated.

Shared resources. Cloud providers usually employ a multi-tenant model, where several users share the same physical machine. One common practice for such sharing of resources is the use of virtualized resources.

2.1.2 Classification of Cloud Systems

Nowadays, cloud services are frequently categorized into pyramid with three layers depending on the type of the provided capability [60]:

Infrastructure as a Service (IaaS). At this level, providers offer an entire virtual on-demand infrastructure such as virtual computing nodes, storage, or network. Thus, through virtualization, providers are able to split, assign and dynamically resize these resources to build ad-hoc systems as demanded by users. This is why it is referred to as infrastructure as a service. Users can then configure their requested infrastructure at a low level (OS, programs, and security among others). Amazon Web Services [2], Rackspace Cloud Sites [68], IBM Blue Cloud [46], and AppNexus Cloud [14] are just some examples of IaaS.

Platform as a Service (PaaS). Instead of supplying virtualized resources, cloud providers can offer software platforms to developers where they can develop and deploy their applications inside the cloud. Similar to IaaS, the underlying infrastructure is hidden from developers and can be scaled it in a transparent manner. PaaS offerings include, among others, the Google App Engine [39], Microsoft Azure [58], Facebook Platform [37], Salesforce.com [70], and Sun Cloud [75].

Software as a Service (SaaS). Cloud providers can also supply applications of potential interest to users and relief them from the burden of local software installation and configuration. These applications can be anything from Web based email to applications like Twitter or an online word processor. Examples at this level are Google Apps [41], Salesforce.com [70], and SAP On-Demand [71].

In addition to the above listed categories, there exists a growing number of other XaaS offers. For example the International Telecommunication Union considers also Network as a Service (NaaS) as a separate category [127]. NaaS includes network related offers such as VPN connection or dynamic bandwidth allocation. Also Database as a Service (DaaS) is a common concept [123]. DaaS offers include standard relational databases such MySQL or Oracle at AWS RDS, but also NoSQL databases such as AWS's Dynamo DB.

Besides this classification according to the offered service level, there is also another dimension considering the different deployment model of cloud services [60, 127].

Private cloud infrastructures are operated and used by a single organization. Often this deployment model is used in cases, where sensible data has to processed.

Community cloud infrastructures are shared among several organizations for a common usage scenario. One example in this category is the British G-Cloud [38], which offers a number of cloud services to the public sector.

Public cloud infrastructures can be used the general public. Usually public cloud are offered by vendors such as AWS, Google, or Rackspace on a pay-per-use basis.

Hybrid cloud infrastructures are a mix of several cloud deployment model. Often public cloud services are used to support the private cloud infrastructure in cases of performance bursts.

Cloud APIs The downside of this growing number of cloud providers is that the number of cloud APIs grows almost in the same rate. As such different APIs are partially a consequence off the different types of services offered. Still, different APIs often result in a *vendor-lockin*, which makes changing a cloud provider very difficult as it might require re-factoring large parts of the applications. There are several projects, such as Apache deltacloud [6], libcloud [50], or simplecloud [72], trying to offer a unified interface on top of the individual APIs. Besides these IaaS API abstractions, there also exist some projects providing service abstractions for specific applications. For example Apache Whirr is able to allocate, configure, and manage a Hadoop Map Reduce Cluster on several cloud vendors. Even though the AWS API is most commonly used, there exists no common standard API which could enable a number of other

beneficial side-effects [89]. For example, a standard cloud API would enable users to freely move their workloads across clouds and private clusters.

2.1.3 Cloud Provider

The number of cloud providers offering different services in a public cloud infrastructure has exploded over the last years. Many large IT vendors such as Microsoft, Google, Apple, SAP, HP, or Oracle offer products in the Cloud. Besides these large players, Cloud Computing has also enabled many small startups such as Salesforce, ProfitBricks, Coursera and many more to offer innovative services in the cloud. Next, we will describe a few prominent cloud offers:

Amazon Web Services (AWS) [2] is probably the best known cloud provider. When AWS EC2 was released in 2006, it was the first commercial large scale public cloud offering. Nowadays, Amazon offers a wide range of cloud services besides EC2: S3, SimpleDB, RDS, ECS, and Elastic MapReduce. Amazon EC2 is also very popular among researchers and enterprises requiring instant and scalable computing power. Actually, the Amazon Elastic Computing Cloud (EC2) was not initially designed as a cloud platform. Instead, the main idea at Amazon was to increase the utilization of their servers, which only had a peak around Christmas.

Microsoft Azure [56] offers IaaS (computing and storage) as well as PaaS (web application hosting) services. Azure's IaaS offer focuses on Windows based virtual machines but also includes Linux based virtual machines. In addition, Microsoft also offers a number of other SaaS cloud offers including Microsoft Office 365 [57] offering the traditional office programs and storage space in the cloud.

Google App Engine [40] is Google's PaaS offer, it allows user to create Java, Python, Go, or PHP based web applications which can be scaled automatically. Since 2012 Google also offers IaaS services with the Google Compute Engine [42]. In 2014 Google introduced one additional IaaS offer Google Container Engine [43] allowing easy deployment of docker containers.

SalesForce.com [70] was one of first successful offers of SaaS around 2000. The inital SaaS offer Salesforce.com is a cloud based software for Customer Relationship Management (CRM) is today used even by many larger companies. Their product platform has since been extended by a PaaS offer Heroku [70] for web application hosting.

Coursera [34] and *Udacity* are among the first companies offering Education as a Service in the Cloud. Students can follow lectures, submit exercises and participate in forums from all over the world and the time best for them.

Table 2.1 presents different other cloud providers with their classification and some examples of their offered services.

Cloud Provider	**Category**	**Major Offer(s)**
3tera	IaaS	VPD
Amazon AWS	IaaS, PaaS	EC2, S3, EMR ...
AppNexus	IaaS	AppNexus Cloud
IBM	IaaS	Blue Cloud
Rackspace	IaaS	Cloud Server
VMWare	IaaS	vCloud Express
Facebook	PaaS	Development Platform
Google	PaaS	App Engine
Google	IaaS	Compute Engine
Microsoft	PaaS	Windows Azure
Sun	PaaS	Sun Cloud
Google Apps	SaaS	Gmail, Google Docs
Salesforce.com	SaaS	Custom Cloud
SAP On-Demand	SaaS	SAP CRM
ProfitBricks	IaaS	virtual data center
Coursera	SaaS	Education as a Service

Table 2.1: Cloud Providers

Amazon EC2 Infrastructure The Amazon Elastic Computing Cloud (EC2) was not initially designed as a cloud platform. Instead, the main idea at Amazon was to increase the utilization of their servers, which only had a peak around Christmas. When EC2 was released in 2006 it was the first commercial large scale public cloud offering. Nowadays, Amazon offers a wide range of cloud services besides EC2: S3, SimpleDB, RDS, and Elastic MapReduce. Amazon EC2 is very popular among researchers and enterprises requiring instant and scalable computing power. This is the main reason why we focus our analysis study on this platform. Amazon EC2 provides resizable compute capacity in a computational cloud. This platform changes the economics of computing by allowing users to pay only for the capacity that their applications actually need (pay-as-you-go model). The servers of Amazon EC2 are Linux-based virtual machines running on top of the Xen virtualization engine. Amazon calls these virtual machines *instances*. In other words, it presents a true virtual computing environment, allowing users to use web service inter-

faces to acquire instances for use, load them with their custom applications, and manage their network access permissions. Amazon EC2 is AWS's IaaS offer and provides resizable compute capacity in a cloud environment. The servers of Amazon EC2 are Linux-based virtual machines running on top of the Xen virtualization engine. Amazon calls these virtual machines *instances*. In other words, it presents a true virtual computing environment, allowing users to use web service interfaces to acquire instances for use, load them with their custom applications, and manage their network access permissions. Instances are classified into different classes: *standard instances* (which are well suited for most applications), *high-memory instances* (which are especially for high throughput applications), *high-cpu instances* (which are well suited for compute-intensive applications), cluster-compute instances (which are well suited high performance compute cluster requiring high network performance), *high-memory-cluster instances* (which are well suited for compute clusters requiring both high network and memory performance), *cluster GPU instances* (which in addition to the cluster-compute instances provide access to GPUS for highly parallel programs), *high I/O instances* (which offer increased I/o performance), and *high storage instances* (which offer a high storage density per instance). In each of these classes AWS offers a number of *instance types*, which specifies the concrete virtual hardware and performance characteristics. As of March 2013 AWS offers total of 18 different instance types. In the following we give the specification of a selection of these instance types.

(1.) *small instance* (Default), corresponding to 1.7 GB of main memory, 1 EC2 Compute Unit (i.e., 1 virtual core with 1 EC2 Compute Unit), 160 GB of local instance storage, and 32-bit platform.

(2.) *large instance*, corresponding to 7.5 GB of main memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of local instance storage, 64-bit platform, and 10 Gigabit Ethernet.

(3.) *extra large instance*, corresponding to 15 GB of main memory, 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each), 1690 GB of local instance storage, 64-bit platform, and 10 Gigabit Ethernet.

(4.) *cluster compute eight extra large instance*, corresponding to 60.5 GB of main memory, 88 EC2 Compute Units, 3370 GB of local instance storage, 64-bit platform, and 10 Gigabit Ethernet.

(5.) *high memory cluster eight extra large instance*, correspond to 244 GB of main memory, 88 EC2 Compute Units, 240 GB of local instance storage, 64-bit platform, and 10 Gigabit Ethernet.

(6.) *high I/O quadruple extra large instance*, corresponding to 60.5 GB of main memory, 35 EC2 Compute Units, $2 * 1024$ GB of SSD-based local instance storage, 64-bit platform, and 10 Gigabit Ethernet.

In our performance evaluation we focus on small instances, because they are the default instance size and frequently demanded by users. Standard instances are classified by their computing power which is claimed to correspond to physical hardware:

One EC2 Compute Unit is claimed to provide the "equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor" [2]. Nonetheless, as there exist many models of such processors in the market, it is not clear what is the CPU performance any instance can get.

While some resources like CPU, memory, and storage are dedicated to a particular instance, other resources like the network and the disk subsystem are shared amongst instances. Thus, if each instance on a physical node tries to use as much of one of these shared resources as possible, each receives an equal share of that resource. However, when a shared resource is underutilized, it is able to consume a higher share of that resource while it is available. The performance of shared resources also depends on the instance type which has an indicator (moderate or high) influencing the allocation of shared resources. Moreover, there exist currently different physical locations (three in the US, three in Asia, one in South America, and one in Ireland) with plans to expand to other locations. Each of these locations contains different availability zones being independent of each other in case of failure.

In addition to the standard on-demand provisioning model, AWS also offers *reserved instances* and *spot instances*. When using reserved instances the user pays a one-time fee for a certain reserved capacity and in turn receives a significant discount on the hourly price when requesting an instance. In a different model, spot instances allow users to bid on non-utilized capacity at their own price. For this model, a so called *spot price* is computed every hour and all instances for bids exceeding this spots price are allocated.

Service Level Agreements Cloud computing is rapidly gaining momentum as an alternative to traditional IT Infrastructure. As more users delegate their applications to cloud providers, *Service Level Agreements* (SLAs) between users and cloud providers emerge as a key aspect. For example, large companies and researchers with critical computing needs usually require well defined SLAs in addition to the ease of use and scalability properties of clouds.

However, cloud providers usually consider in their SLAs only the availability of services, e.g., Amazon EC2 provides a basic SLA, which states a 99.95% monthly uptime percentage [2]. Like Amazon, this QoS based on uptime of services is an inherent feature of many other clouds [1, 68]. However, the QoS attributes that are generally part of an SLA (such as response time and throughput) are not offered by cloud providers. This is mainly because such attributes change constantly due to the elasticity of clouds. Thus, as cloud providers do not give such performance guarantees for their hardware, the burden to ensure SLAs is up to the application developer.

Software for Private Cloud Infrastructure For provisioning private (or hybrid cloud installation) there exist a number of software packages allowing users to setup their own IaaS cloud. Eucalyptus [146] for example allows users to create a private cloud which is API compatible to AWS. Eucalyptus is also an official AWS partner allowing for easy setup of hybrid cloud installation across the private datacenter and AWS: Openstack [62] is another open source tool for creating a private cloud setup. Openstack was originally developed by NASA and rackspace and is today supported by many large companies [62] including IBM, HP, Intel, Cisco, and VmWare.

2.1.4 Legal Aspects and Privacy

One issue with Cloud Computing is that the user gives up control over his data to a certain degree as it by definition stored or processed in the Cloud. Especially, with public cloud vendors this is an issue for many companies handling sensitive data. Another issue is that most cloud vendors are based in the US where the data privacy legislation is sometimes in conflict with the european data privacy requirements [1]. For example, the german Bundesdatenschutzgesetz [24] (federal data privacy law) states that no person related data is allowed to be stored on servers outside Europe. This restriction does not apply if the cloud provider has signed the *Safe-Harbor-Agreement* [69] which offers certain

[1] "While the United States and the EU share the goal of enhancing privacy protection for their citizens, the United States takes a different approach to privacy from that taken by the EU. The United States uses a sectoral approach that relies on a mix of legislation, regulation, and self-regulation. The EU, however, relies on comprehensive legislation that requires, among other things, the creation of independent government data protection agencies, registration of databases with those agencies, and in some instances prior approval before personal data processing may begin." [69]

data privacy guarantees. Still, as US based cloud vendors fall under the legislation of the *Patriot Act* [66, 36],US law enforcement agencies can request all data from the providers servers even if these servers are located in other countries. Users can use encrypt their data as a partial countermeasure [98]. But as of today encryption is used in practice mostly for storage or transmitting data. As soon as it comes to processing data the use of encryption is severely limited. There have been recent advancements in the field of of *Homomorphic Encryption* [118, 140, 87] which promises the ability to perform any operation on encrypted data. But as of today, such operations are limited to very basic ones such as additions or multiplications and there are still many open challenges [140].

CryptDB [153] follows a different approach: CryptDB allows for SQL queries over encrypted data. While the data is stored encrypted on the the server it has to be decrypted for processing the data. As for decrypting the data the user still has to provide the decryption key to the server, the system still requires the user to trust the system and provider to delete the key and all plain data after processing the query. Note that encryption by the vendor does not guarantee privacy, especially in case the cloud provider cannot fully be trusted (e.g., if he falls under the jurisdiction of Patriot Act).

But even though the data itself might be encrypted by the user, privacy might still be at risk. First of all, the user's cloud usage pattern reveals much information in itself. For example an attacker might employ activity mining (e.g., identifying excessive cpu usage for simulations) to gain insight of a certain project status of a competitor. Secondly, also the data transfer to the vendor is a potential thread to privacy even when using SSL encryption. As of autumn 2013 the NSA is believed to have obtained a number of SSL master keys enabling them to monitor even SSL encrypted data transfers. Also in some settings the SSL encryption keys can be retrieved even long time after the initial communication has finished.

Hence, sensible data should be already encrypted on a secure and trusted user controlled environment and no decryption key should be provided to any third party including the cloud provider. Data encrypted on a safe device with a safe encryption standard (i.e., not influenced by the NSA) is to the best of our knowledge safe. As decryption technology (and speed) is advancing, the user should use an established encryption scheme which is likely to be safe in the future (e.g., Elliptic Curves [141]) and choose a large enough key length.

2.1.5 Cloud Security

Using virtualization to share physical resources also yields several new security risks compared to traditional in house datacenters [98]. Especially virtual machines which are co-located on the same physical hardware experience less security isolation than physically separated machines. Even though the hypervisor tries to provide reliable isolation there exits a number of side channel attacks for co-located virtual machines. For example usually not all cache levels are isolated between virtual machines. Hence the cache usage can usage can be used to determine co-location of virtual machines [158, 175] and [93, 85] describe how a shared cache can be exploited to extract AES keys and Open SSL DSA keys. Another potential security threat comes from Denial of Service (DoS) attacks which can slow down cloud services to a state where they are useless. This can be done for co-located virtual machines by overusing the different cache levels or I/O devices [158], but also cloud wide attacks exits which attack a larger part of the providers cloud services: In 2009 emails send from addresses of a large part of EC2's IP range where marked automatically as spam [98]. This was because hackers had used EC2 virtual machines to send spam and hence the IP range became blacklisted. As co-location allows for more elaborate attacks the question of how to determine and force co-location of virtual machines has been a topic of interest for a while. An extensive map of the EC2 network has been performed by [158]. They also describe several techniques including network placement, network routing information, ping times and cache usage for determining co-location of instances inside EC2.Finally the authors also examine techniques for forcing the co-location of a given virtual machines with a victim virtual machine.

What are potential countermeasures against those techniques in particular the ones exploiting co-location? The best solution would be to avoid co-location of virtual machines from different customers at all. This can be done by EC2's dedicated instances [16] which provides dedicated physical hardware for each customer. This service can only be used in conjunction with EC2's Virtual Private Cloud [21] which provides an isolated virtual network for the customer. And of course EC2 charges additional fees for both services. For US based government client EC2 even offers the AWS GovCloud [17] which runs on dedicated virtual machines inside the US and is maintained by selected US personal only. The problem of exploiting a shared hierarchy is countered by [157]. As potential solutions the authors propose a cache hierarchy aware scheduling and a cache partitioning scheme.

2.1.6 Benchmarking of Cloud Systems

Benchmarking cloud systems provides many challenges beyond the well established techniques for benchmarking physical systems. First of all there is the question which metrics are interesting when benchmarking a Cloud Setting? Besides traditional performance metrics such as CPU performance or I/O throughput there are are also other metrics which are relevant in a Cloud Setting. For example [95] suggests to include fault tolerance (i.e., does the system react to failures in an invisible way for users), peak performance (i.e., how does the system cope with with short performance bursts), scalability (i.e., to which workloads does the system scale), and cost (i.e., how expensive is a single piece of work) as new metrics. From our experience we would suggest to also include performance robustness (i.e., how stable is performance over time or different instances) and the provisioning effort (i.e., how easy and costly is it to scale up or down).

Another challenge in cloud benchmarking is that each vendor offers its own set of products even for relatively standard offers such as IaaS. This poses a challenges when comparing cloud products from different vendors [95].

From a technical point benchmarking also offers new challenges. During our benchmarks we frequently encountered problems with failing infrastructure or undesired side effects. Such effects for example include outliers in network benchmarks due to dropped packages by the hypervisor. Therefore *active benchmarking* [121] is even more important for cloud settings than it is for benchmarking physical. Active benchmarking refers to the practice of monitoring whether the benchmark is measuring the desired metric and running as expected. For example one should for example monitor whether an benchmark which is supposed to measure network performance is not actually bound by CPU or memory performance.

Another dimension for cloud benchmarking is the quantification of performance isolation, i.e., whether different virtual machines on the same host affect each other. The Isolation Benchmark [142] has a simple approach to this question: One virtual machines runs a web server while other provide stress test for CPU, memory, disk IO, or network IO. The degree of performance isolation is then measured be the increase of dropped requests to the web server.

Still, with the growing importance of cloud computing in general also cloud specific benchmarking gains importance. Therefore a number of cloud specific benchmarks have been proposed over the last years. These benchmark do not

focus on micro-benchmarking but rather on typical cloud applications. One of the first cloud benchmarks was the Yahoo Cloud Serving Benchmark [103, 81]. This benchmark focuses on OLTP like cloud systems such as cloud databases including various NoSQL database systems.

Cloud Suite [28] is a more recent cloud benchmark and also covers a wider range of typical cloud use cases. Here a list of the use cases covered by Cloud Suite and the respective benchmark:

- **Data Analytics** Mahout executing MapReduce Jobs
- **Data Caching** Memcached
- **Data Serving** Cassandra
- **Graph Analytics** TunkRank machine learning algorithm
- **Media Streaming** Darwin streaming server
- **Software Testing** Cloud 9
- **Web Search** Nutch/Lucene

Furthermore, there exists a number of companies which offer cloud benchmarking. Notably CloudHarmony [26] and CloudSpectator [27] allow the user to execute any from a large selection of benchmark on any of the many cloud vendors. This can help other companies to quickly compare a selection of different cloud systems using a benchmark relevant to their actual workload.

2.1.7 Hardware Trends for the Cloud

Cloud Computing and especially the use case for massive data analytics does not only require new algorithms such as MapReduce but also different hardware.

One of the major cost driver for large cloud vendors is often energy cost which includes the productive energy to power to the hardware but the energy required to cool the datacenter. Facebook started early to investigate efficient hardware and datacenter solutions in order to minimize the acquisition and maintenance cost. As a result Facebook uses customs made chassis and motherboards in their data centers [143, 115]. These results are even available to the public by the *opencompute* project [61] where Facebook has made it datacenter and hardware designs available as open source.

So far the vendors where mostly focused on datacenter or system layout, but new research also indicates that cloud infrastructure might require a different micro-architecture design at processor level [113, 139].

Today's processors like the Intel's E7 are complex processor include many features such as Out of Order execution, branch prediction, and large multistage execution pipelines. Such features are mainly developed for the use cases of typical desktop user (i.e., many different operations with a instruction level parallelism). Unfortunately, these feature also come at the cost of die space (which could also be used for more cores) and increased energy consumption. Another problem for such processors is that they usually contain a complex cache hierarchy usually consisting of a small split L1 data and instruction cache (about 32-64 KB), a medium sized shared L2 cache (about 256 KB), and a large L3 cache (about 12 MB). Again such design is optimized for the typical use cases of end users.

Such processors are also used by cloud vendors. Throughout our experiments in 2013 we found for example the following processor types for the different cloud vendors:

- **AWS EC2** Intel E5-2650 2.00GHz, Intel E5430 2.66GHz, Intel E5507 2.27GHz, and Intel E5645 2.40GHz
- **Google Compute Engine** Intel Xeon CPU 2.60GHz
- **MS Azure** AMD Opteron 4171 2.094 GHz

The issue with such processors is that typical cloud computing use cases (especially big data analytics) does not require the complex features offered by these processors.

[113] compares the processor efficiency for different cloud use cases such as data serving (using cassandra), data analytics (using Mahout/MapReduce), and Web Search (using Nutch/Lucene). Their results indicate three main inefficiencies in modern processors for such workloads:

- **High instruction-cache miss rate.** The small L1 instruction cache is too small for instruction sets used by the cloud workloads. In addition the complex cache hierarchy makes these cache misses even more expensive.
- **Low benefit of Out-of-Order executions and other features.** The cloud workloads typically incur only little instructions-level parallelism (in contrast to data parallelism). Therefore the benefit of features such

as multistage pipelines, branch predication, or Out-of-Order execution is limited and the die space (and power) consumed by these features could be spend more efficiently on for example more cores.

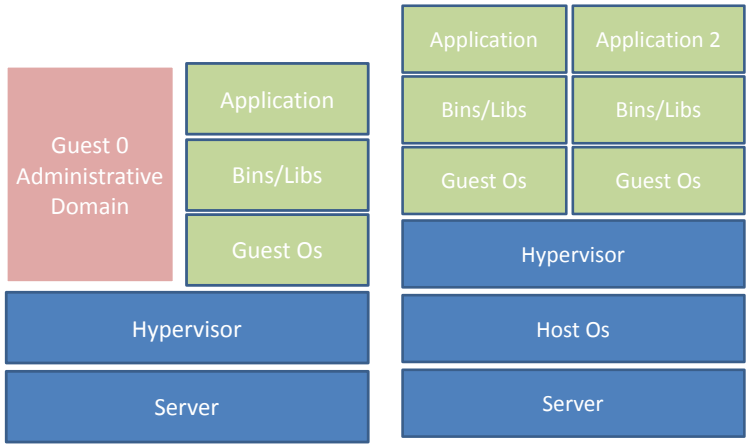
- **Data sets exceeding L3 caches.** The L3 caches are by orders of magnitude too small for the cloud sized datasets. Here again the die space used by the L3 cache could be used more efficiently for other components.

So it seems to be time for cloud vendors to explore different processor architectures as indicated by [113, 139]. One candidate here could be the simple and energy efficient ARM processors [119] which are even already included in the opencompute project's designs [61].

2.2 Virtualization

Most cloud vendors rely heavily on virtualization, allowing the partitioning and separation of resources for different users on a single physical machine. There exists a large number of different virtualization techniques which we will classify in two different categories according to the level of isolation they provide. In Section 2.2.1, we consider different *Hardware Virtualization* techniques which enable the user run his own Operating System Kernel on top of an Hypervisor. As Hardware Virtualization provides each virtual machine with its own Operating System, one achieves good isolation between different virtual machines. But this comes at the price of hight overhead for emulating an entire Operating System. Therefore *Operating System Virtualization* provides virtualization on top of existing operating system on a process level. We discuss Operating System Virtualization in Section 2.2.3.

2.2.1 Hardware Virtualization



(a) Native Virtualization. (b) Nested Virtualization .

Figure 2.1: Different Hypervisor Types adapted from [121] .

Hardware virtualisation is usually rely on Hypervisors or Virtual Machine Monitors to translate the logical requests from the virtual machines to physical requests for the underlying hardware. Hypervisors [99] can classified based on

its distance to the underlying hardware: *Native Hypervisors* as shown in Figure 2.1(a) work directly on top the hardware. They usually contain a special privileged virtual machine responsible for administration purposes. Xen [90] is one example of Native Hypervisors.
Nested hypervisor as shown in Figure 2.1(b) work on top of another Operating System, which can also be used for administrative purposes. One example for nested hypervisors is the Kernel Based Virtual Machine (KVM) [49].

Furthermore hypervisors can be classified into three categories based on the underlying virtualization technique [159]:

Full Virtualization is a virtualization which fully emulates the physical hardware and therefore provides a binary translation for the low level system calls. Because of the complete emulation, full virtualization supports any unmodified operating system but incurs a certain overhead due to the the binary translation of low level commands. Examples of hypervisors based on full virtualization include VmWare's ESX Server [78], Microsoft's HyperV [54], and Kernel Based Virtual Machine (KVM) [49].

Para-Virtualization on the other hand requires the use of special, modified Operating Systems which are aware of the virtualization. This is due to the fact that the virtual hardware provided by the hypervisor is only similar to the physical hardware emulated. Some system calls which in case of full virtualization are translated, have to handled by hypervisor in case of paravirtualization. This requires the (guest) kernel to use the respective hypervisor calls instead of the original system calls. For this reason, as of today Microsoft windows does not run on para-virtualized hypervisor. But, para-virtualized machines usually show better performance than fully virtualized machines as it does not require binary translation of system calls. Since 2005 there is also hardware support for *Hardware-Assisted Virtualization* which supports the idea of para-virtualization on the hardware level. Intel VT-x [47] and AMD-v [5] are the most prominent techniques enabling hardware-assisted virtualization in the CPU. These CPU offers special commands for the low level system calls which can be used by the virtualized Operating System directly. Examples of hypervisors based on para-virtualization are Xen [90] (and the most prominent commercial Xen based product Citrix Xen-Server [25]) and Oracle VM Server [65].

Hosted Virtualization refers to solution which provide virtualization on top of other operating systems. As all system calls have to be translated and then passed on to the host operating system the performance is usually worse

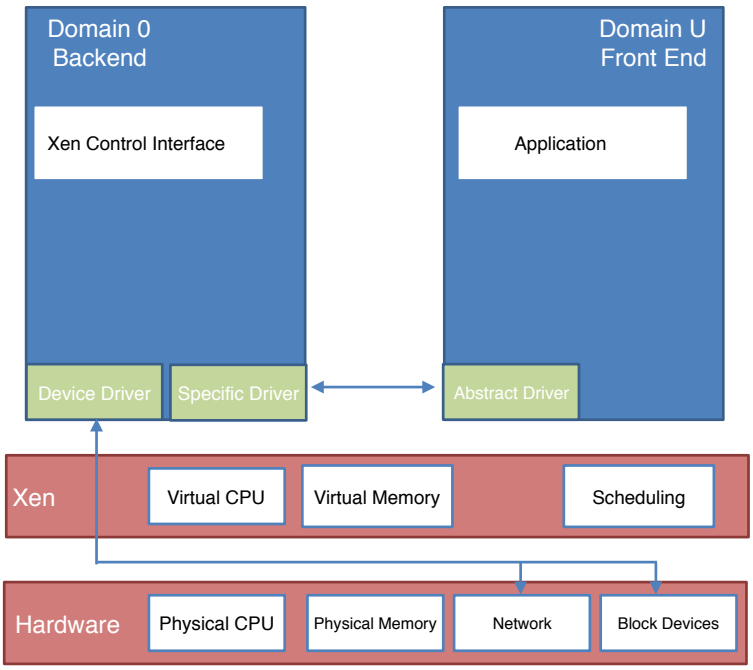


Figure 2.2: Xen Dom0 and DomU (adapted from [99]).

than with fully or paravirtualized machines. Examples in this category include VmWare Workstation [79] and Oracle Virtual Box [64].

2.2.2 Hypervisor Implementation

The implementation of the hypervisor and especially the virtualization of devices has a large effect on the virtual machine performance and performance separation between different virtual machines. Therefore we will take a look on how the virtualization of devices is implemented in the example of the Xen hypervisor [90] which is the hypervisor used by AWS [3].

Recall that Xen offers paravirtualized meaning that the virtual hardware provided by the devices is only similar to the physical hardware. This allows for more efficient handling for low level instructions.

The architecture of Xen basically differentiates between two different kinds of virtual domains as shown in Figure 2.2. As Xen does not include any standard device drivers by itself, the Dom0 virtual machine is responsible for providing

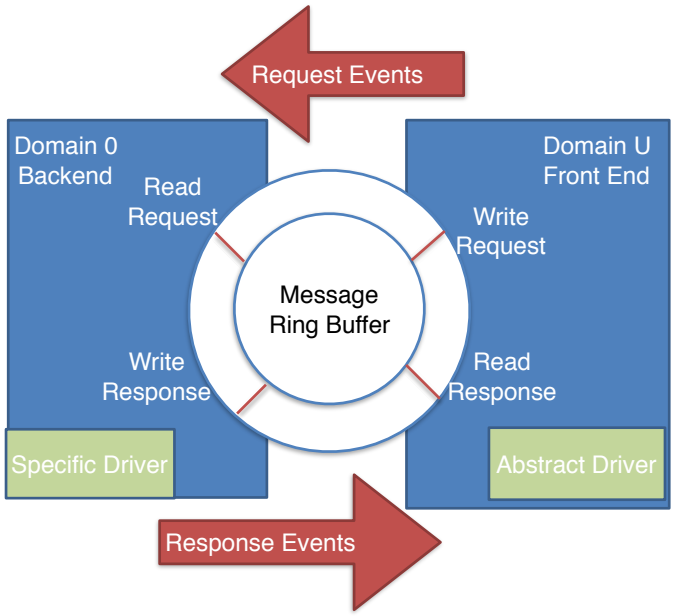


Figure 2.3: Split Device Driver (adapted from [99]).

the actual device drivers for the underlying physical hardware. This relieves the Xen developers from having to develop special device drivers for all different hardware. The Dom0 machine has usually an operating system with broad driver support such as a Linux. As the Dom0 driver control the actual hardware it requires special security. Therefore it should rather be considered a part of Xen than a virtual machine exposed to users.

That is the job for the virtual machines in the unprivileged DomUs. Here the user can choose which operating system (with paravirtualization support) and software he wants to run on those machine. As the DomUs have usually no direct access to the physical hardware they require special drivers. For simplicity these drivers are quite simple. For example the hard disk driver is not designed for specific SCSI or IDE hard disk but instead for an abstract block device. The actual work taking care of the device specifics is done by standard driver in the Dom0. Xen describes this concept as `split driver` [154, 99]. As shown in Figure 2.3 a Xen split driver consists of 4 different parts:

- The actual driver on the Dom0 responsible for communicating with the actual hardware.
- The left half of the split driver on the Dom0 responsible for scheduling

and controlling the access by the different DomUs.

- The ring buffer acting as communication queue between Dom0 and DomUs.
- The right half of the split driver acting as simplified device driver on the guest DomUs.

This separation allows several different virtual machines to use the physical devices in parallel. Especially the split driver half located on the Dom0 has therefore great responsibility in controlling access to the underlying hardware. For example in case of an hard disk this would mean that each virtual machine can access its own part of the physical hard disk. In addition this part of the split driver is also responsible for the multiplexing access between the different virtual machine and the complete driver should not suffer from too much performance penalty due the indirection. Therefore the implementation of the split drivers is of great importance for the overall performance of Xen.

A Sample workflow with the example of a network driver is shown in Figure 2.4. The network stack on the guest DomU first goes through the normal network stack. Once it is time for the virtual network adapter (i.e., the split driver) to transfer the package it turns control over to the Xen hypervisor which in turn transfers control to the Dom0. Here the other half of the split driver processes the package and then uses the real network device driver provided by the Kernel to transmit the package.

In addition is is also possible to expose certain devices such as PCI adapters or USB drives for exclusive use to an individual DomU. Since Xen 3 there is also the possibility for so called `isolated driver domains` (IDDs) [99]. IDDs are basically special domains which only run a certain (physical) device driver. As this device driver is not required to run in the Dom0 anymore this approach increases the stability of the Dom0 and therefore the entree system.

CPU: Basic CPU virtualization is quite easy compared to memory or I/O virtualization. The hypervisor basically has to allocate a share of the physical CPU cycles to each virtual CPU. Once a virtual CPU's share is exhausted, the hypervisor has to interrupt the virtual CPU, save the current CPU state and grants the next virtual CPU access to the physical CPU. But if the virtual CPU would be identical to the physical CPU, a virtual machine could use all low level system call which could also influence the state of the other virtual machine's (e.g., by halting the entire system or power down physical CPUs).

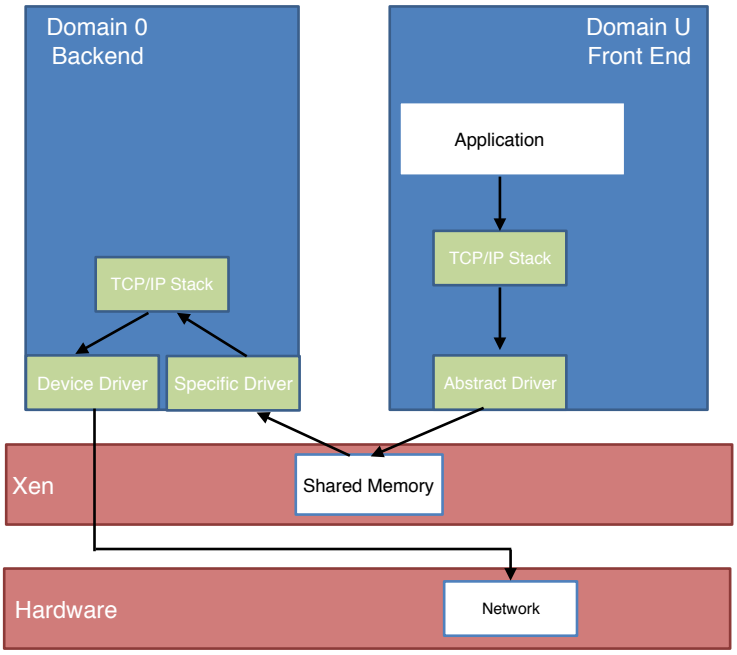


Figure 2.4: Xen Network Driver (adapted from [99]).

Therefore the virtual CPU offered by XEN is slightly different from the physical CPU which ensures that all such `behavior sensitive instructions` [99] are handled by the hypervisor. Still, most operations are directly mapped to physical CPU and do not require binary translation as in the case of full virtualization.

Memory: The challenge for the hypervisor when providing memory is mostly to partition the available main memory between different virtual machines and ensure each virtual machine can only access its own virtual memory.

This challenge is quite similar to what a `Memory Management Unit` (MMU) already does in a physical system. In modern systems the concept of `virtual memory` exists already for a long time, but is not directly related a machine virtualization. Instead the term virtual memory refers to an abstraction in modern system which allows each process to have its own virtual address space in main memory. The MMU is then responsible for translating these virtual addresses to the actual physical addresses in main memory. This indirection allows virtual memory to be paged to disk or moved in the physical main

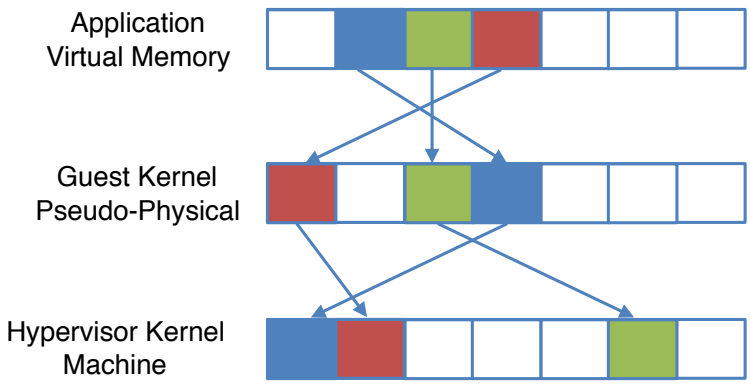


Figure 2.5: Xen Memory Indirection (adapted from [99]).

memory. Typically a MMU also includes a cache, the `Translation Lookaside Buffer` (TLB) in order to speed up the expensive translation of virtual to physical addresses.

The hypervisor adds yet another layer in between such that each virtual machine has its own address space which is usually referred to as `protected memory` [99]. The pattern for a protected memory access in a virtual machine is shown in Figure 2.5. This additional indirection can cause performance problems as it requires one additional translation by the hypervisor for each memory access. Therefore to improve the performance Xen can allow each virtual machine direct read access to the MMU. Hence only operations changing the page table inside the MMU (e.g., request for new memory pages) have to be processed by the hypervisor [99, 80].

Still direct read access to the MMU from one virtual machine can still have impact on the memory performance of the other machines. For example read access from one virtual machine can cause the virtual memory from another virtual machine to be evicted from the physical memory and be paged to disk. The next memory access to such virtual memory by the other machine would results in an expensive page fault and require the virtual memory to be loaded back from disk into physical memory. Another example is the cache utilization of the MMU's TLB. Depending on the TLB size each virtual machine will fill the (physical) TLB with addresses from its own virtual address space and thereby evict TLB entries from other virtual machines. On the next memory to such memory location this triggers an expensive calculation of the mapping

from virtual to physical memory.

Other Devices Other devices provide even more challenges for the hypervisor. The main problem is that the hypervisor is unaware of the internal state of the devices. This is problematic when switching between different virtual machines as for each switch the hypervisor has to save the current state (for the virtual machine just being halted) and then restore a previous (for the virtual machine being loaded). This is for example especially challenging for advanced 3D graphics adapters which have a lot of internal state including large frame buffers, which are infeasible to being unloaded and reloaded at each each time the hypervisor switches between different virtual machines [80].

Another problem arises by the use of Direct Memory Access (DMA). In modern systems DMA is used to allow devices to write directly to main memory without going through the CPU. One example could be a network adapters which using DMA can writes received packages to main memory without having to interrupt the CPU. In a virtualized setting DMA becomes more challenging as the main memory is not exclusively owned by a single physical machine, but instead many virtual machines. One example would be the virtual network adapter for one virtual machine wants to write a certain memory region which the hypervisor has allocated to another virtual machine. In theory hypervisor could examine all commands for DMA call but this would be prohibitively expensive and also quite complex.
The Xen solution for the DMA problem are `I/O Memory Units` (IOMMU) which are basically the DMA equivalent of Memory Mangement Units. IOMMUs can translate and isolate the DMA accesses for different virtual machines [91, 80]. They can be either implemented in hardware as for example AMD's Gart[91] or in software as for the Unix swiotlb library which is also used by Xen. Note that IOMMUs introduce yet another level of indirection between physical and virtual hardware which can influence performance negatively. Also the scheduling strategy of the IOMMU can affect the performance of concurrently running machines as it is a single resources required by these different virtual machines.

2.2.3 Operating System / Container based Virtualization

In contrast to Hardware Virtualization, where each virtual machines runs with its own operating system, *Operating System Virtualization* or *Container based*

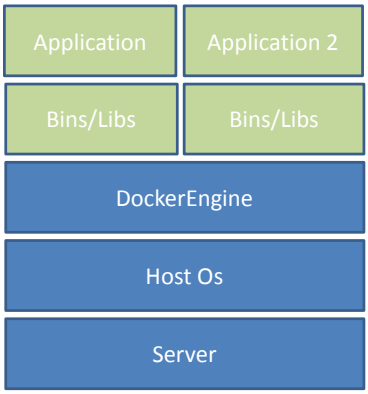


Figure 2.6: OS Virtualization (adapted from [121]).

Virtualization [163] execute isolated containers on top of one common Operating System as shown in Figure 2.1.

Modern operating systems offer a number of different container techniques including Solaris Zones [73], OpenVZ [63], Linux-VServer [52] and probably most prominent LXC (LinuX Containers) [51] . Container enable users to run multiple isolated systems on top of single operating system kernel. Here each system can have its own configuration and resources. Resource isolation (CPU, memory, block I/O, network, etc.) is achieved by kernel isolation techniques. For example the Linux kernel offers *cgroups* [51] for resource isolation. These cgroups enable the user to provision isolated namespaces for a given group of processes and provide isolation on process, network, memory, user and file system level. The new standard linux init daemon *sytemd* [164] also relies heavily on cgroups for grouping and isolation between different processes.

Docker [35, 169] is built on top of Linux Containers and allows lifecycle management of containers and enabled a similar treatment as virtual machines. Docker also enables a more fine-grained security management such as separating the container root users from the system root user.

With the increase in management tools such as Dockers and massively reduced performance overhead compared to hardware virtualization many companies are moving towards container based virtualization. For example Google's entire infrastructure is based on container based virtualzation and each week Google deploys more than 2 billion container instances [31]. With *coreOS* [33]

there exits even a Linux Distribution focusing on providing a foundation for container based virtualization. Still, one disadvantage of sharing the same kernel for container based virtualization is that the kernel versions for the different containers have to match and there is no support for the Windows operating system. Container have also arrived in the public cloud with Amazon EC2 Container Service [15] and Google Container Engine [43]. Such offers allow the user to easily deploy docker containers into a cloud environment.

2.2.4 Comparison of Hardware and Operating System Virtualization

Both virtualization approaches have advantages and disadvantages as shown in Table 2.2. Basically Hardware Virtualization offers better isolation between different virtual machines and (as of today) better support while container based virtualization offers much lower overhead and a growing community [110, 166]. Therefore, most public cloud vendors as for example Amazon EC2 offer Hardware based virtual machines as of 2014. But there is a growing number of PaaS cloud offers, where users usually span a large number of virtual machines using container based virtualization including the Google Cloud Platform [32].

	Hardware	**Container**
Operating System	one OS kernel per VM	all VMs share common OS
Isolation	(almost) complete isolation	based on cgroups
Performance	large overhead due to emulation	very little overhead
Start Up Time	minutes	seconds
Storage Overhead	each VM stores own OS	only settings and applications
Memory Access Overhead	None	Some
I/O Latency	None	Some

Table 2.2: Virtualization Techniques Comparison [110, 121]

2.3 MapReduce

2.3.1 Overview

MapReduce was proposed by Google in 2004 [105] as a framework to facilitate the implementation of massively parallel applications processing large data sets. MapReduce is inspired by functional programming where developers simply need to describe their analytical tasks using two functions *map* and *reduce*. Everything else including parallelization, replication, and failover will then be handled by the MapReduce framework. Usually, any computing node may run several map and reduce processes at the same time. The MapReduce implementation of Google is not freely available, but an open source implementation exists, coined Hadoop [44].

In a MapReduce cluster, one central process acts as the *Job Tracker* and coordinates MapReduce jobs. All other processes act as *workers*, meaning that they execute tasks as assigned by the master. A single worker can run either map or reduce tasks. For clarity, we denote a worker designed for running map tasks as *mapper* and a worker designed for running reduce tasks as *reducer*. Thus, by convention, a computing node can perform both mappers and reducers at the same time.

Let us look in detail how the following query is executed `JobAVG SELECT category, AVG(price) FROM SALES GROUP BY category` over the Sales data shown in Table 2.7.

Before starting her MapReduce job, the user uploads the SALES table (i.e., Figure 2.7) into the *Hadoop Distributed File System* (HDFS). Once the dataset is uploaded to HDFS, the user can execute JobAVG using Hadoop MapReduce. In turn, Hadoop MapReduce executes JobAVG in three main phases: the *map phase*, the *shuffle phase*, and the *reduce phase*.

Map Phase. Hadoop MapReduce first partitions the input dataset into smaller horizontal partitions, called *input splits*. Typically, an input split correspond to an HDFS block. Hadoop MapReduce creates a map task for each input split. Then, the Hadoop MapReduce scheduler is responsible of allocating the map tasks to available computing nodes. Once a map task is allocated, the map task uses a *RecordReader* to parse its input split into key-value pairs. For JobAVG, the RecordReader produces key-value pairs in the form `(SALES.id;(SALES.category,SALES.price))` for each line of input data. The map task then executes a map-call for each of key-value pair independently.

	id	category	price
r1	100	b	4
r2	189	b	6
r3	132	c	2
r4	73	f	9
r5	150	f	9

Figure 2.7: SALES Table

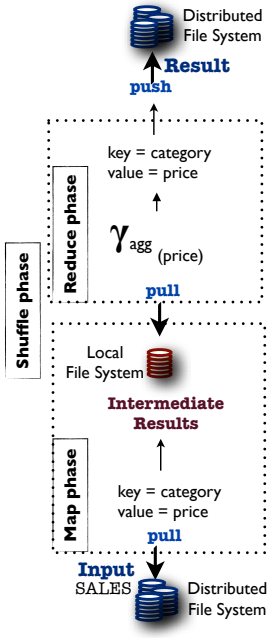


Figure 2.8: MapReduce plan

The output of a map-call might be zero or more intermediate key-value pairs. For JobAVG, the map output is in the form `(SALES.category;SALES.price)`. Hadoop MapReduce stores the map output in local disk.

Shuffle Phase. Hadoop MapReduce partitions the intermediate key-value pairs by intermediate key and assigns each partition to a different reduce task. The number of reduce tasks is user-defined. Then, Hadoop MapReduce shuffles the partitions through the network to its respective reduce task. This means that the intermediate results produced by all map tasks that have the same intermediate key will end up in the same reduce task. Additionally, since the input of reduce tasks can contain intermediate key-value pairs with different intermediate keys, Hadoop MapReduce performs a sort-based group-by over the input of each reduce task. Hence, JobAVG groups and partitions the intermediate data by `SALES.category`.

Reduce Phase. The reduce task executes a reduce-call for each group of intermediate key-value pairs. The output of a reduce-call might be zero or more final key-value pairs. Finally, Hadoop MapReduce stores the output of reduce tasks in HDFS. Notice that, if a user specifies zero reduce tasks, then

Hadoop MapReduce executes only the map phase and stores the map output directly in HDFS rather than on local disk.

MapReduce Failover Properties

As MapReduce usually processes large amounts of data using hundreds or even thousands of nodes, tolerating task and worker failures is crucial. A major advantage of using MapReduce [105] and Hadoop [44] is the fact, that both frameworks deal with such failures and achieve fault-tolerance in the following ways:

Task failures are usually caused by either bugs or bad records. Bugs in the map or reduce function or even in an external tool (e.g., the JVM) may cause a task to fail if the exception thrown by the bug is not handled correctly. *Bad records* in input data can also cause a task failure if the implementation of user defined functions is unable to handle this particular form of bad records. While bugs is unlikely, bad records and contention are unpredictable and highly likely to occur because of two main reasons: (i) MapReduce jobs typically process data produced by other sources and thus there is no guarantee that data is consistent; (ii) disk corruption [165] is also a cause because both map and reduce functions do not often account for all possible forms of corrupted media, like hard drives; (iii) MapReduce jobs usually compete for shared resources, which increases the probability for a task to hang.

To correctly recover from task failures, whenever one of these errors occurs, workers mark the task as failed and inform the master about the task failure. Then, the master puts the task back in its scheduling queue. If the failed task is a map task, it notifies reducers of the re-execution decision as soon as the task is rescheduled. This allows reducers to fetch their required data from the new mapper.

Worker failures are mainly due to a computing node failure, which in many cases results from hardware failures. If the MapReduce framework notices a worker failure it marks it as blacklisted. This means for a task tracker that already scheduled jobs are rescheduled to other nodes and for a data node that the data blocks residing on this node are re-replicated to other data nodes.

Master failure are caused when the master (i.e., job tracker or name node) fail. In MapReduce, the master is a single point of failure, but this can easily be solved by having a backup node maintaining the status of the master node. Thus, the backup node can take over the master role in case of failure.

MapReduce State of the Art

Over the past years MapReduce has attained considerable interest from both the database and systems research community [138, 109, 124, 114, 155, 132, 131, 130, 128, 147, 117, 84, 167, 101, 172, 144, 102]. As a result, some DBMS vendors have started to integrate MapReduce front-ends into their systems including Aster, Greenplum, and Vertica. However, these systems do not change the underlying execution system: they simply provide a MapReduce front-end to a DBMS. Hail [109] and Hadoop++ [108] speed up the processing of MapReduce jobs by the use of indexes. For this purpose Hail even generates a different clustered index per HDFS replica. As the indexing done while uploading it incurs no to little overhead. HadoopDB [84] combines techniques from DBMSs, Hive [167], and Hadoop. In summary, HadoopDB can be viewed as a data distribution framework to combine local DBMSs to form a *shared-nothing DBMS*. The results in [84] however show that HadoopDB improves task processing times of Hadoop by a large factor to match the ones of a shared-nothing DBMS. Nevertheless, above systems are still databases. Therefore, in contrast to MapReduce, they require advanced knowledge from the user-side on database management, data models, schemas, SQL, load distribution and partitioning, failure handling, and query processing in general, but also on the specific product in particular.

[131, 132] have considered different data layouts for efficiently storing relational data. As MapReduce is usually I/O bound these approaches help to reduce the amount of input data which has to read from disk.

Manimal [96] proposed to analyze MapReduce jobs to determine filter conditions. Then, Manimal rewrites MapReduce jobs to match an previously created index. On the other side, much work has been done on scheduling MapReduce jobs with different goals in mind. Hadoop for example include a Fair and Capacity scheduler with the aim of sharing computing cluster among jobs [45]. However, the homogeneity assumption made by Hadoop might lead to a degradation on performance in heterogenous clusters such as the Cloud. Zaharia et al. [174] proposed a scheduler to schedule MapReduce jobs in heterogeneous MapReduce clusters. Nonetheless, this work does not allow users to neither dynamically change the setup of their experiments nor take into account monetary costs of jobs.

Even though continuing popularity of Big Data Analytics the traditional MapReduce is rapidly overtaken by more flexible processing models such as

Apache Spark [10], Apache Flink [7], or Apache Tez [12]. This trend is also reflected by Hadoop 2.0 [44] supporting a number of different processing engines besides the traditional MapReduce engine.

3 Performance Variance in the Cloud

3.1 Introduction

As seen in Section 2.1.1 Cloud Computing is a model that allows users to easily access and configure a large pool of remote computing resources (i.e., a *Cloud*). This model has gained a lot of popularity mainly due to its ease of use and its ability to scale up on demand. As a result, several providers such as Amazon, IBM, Microsoft, and Google already offer public *Infrastructure as a Services (IaaS)* where users can request a certain number of virtual machines. For many users, especially for researchers and medium-sized enterprises, the IaaS cloud computing model is quite attractive, because it is up to the cloud providers to maintain the hardware infrastructure. However, despite the attention paid by cloud providers, some of requested nodes may attain orders of magnitude worse performance than other nodes [89]. This indeed may considerably influence performance of real applications. For example, we show the runtimes of a MapReduce job for a 50-node EC2 cluster and a 50-node local cluster in Figure 3.24. We can easily see that performance on EC2 varies considerably. may occur. In particular, contention for non-virtualized resources (e.g., network bandwidth) is clearly one of the main reasons for performance unpredictability in the cloud.

Figure 3.1: Runtime for a MapReduce job.

This Performance unpredictability is in fact a major issue for many users and it is considered as one of the major obstacles for cloud computing [89]. For example, researchers expect comparable performance for their applications at any time, independent of the current workload of the cloud; this is quite important for researchers, because of repeatability of results. Another example are enterprises that depend on *Service Level Agreements* (SLAs) (e.g., a Web page has to be rendered within a given amount of time). Those enterprises expect cloud providers to make *Quality of Service* (QoS) guarantees. Therefore it is crucial that cloud providers offer SLAs based on performance features — such as response time and throughput.

There is currently a clear need for users, who have to deal with this performance unpredictability, to better understand the performance variance for several cloud vendors. This chapter aims to first quantify and understand this performance variance based on several micro benchmarks in Section 3.6.1. Following, we examine whether the performance variance effects the application layer in Section 3. For these measurements we consider several different dimensions including time (i.e., we show results from the years 2010 and 2013) and different cloud vendors (i.e., we compare the Amazon EC2, Google, and Amazon public IaaS offers).

3.2 Related Work

Cloud computing has been the focus of several research works and is still gaining more attention from the research community. As a consequence, many cloud evaluations have been done with different goals in mind. Armbrust et al. [89] mention performance unpredictability as one of the major obstacles for cloud computing. They found that one of the reasons of such unpredictability is that certain technologies, such as PCI Express, are difficult to virtualize and hence to share. Lenk et al. [135] propose a generic cloud computing stack with the aim of classifying cloud technologies and services into different layers, which in turn provides guidance about how to combine and interchange technologies. Binnig et al. [95] claim that traditional benchmarks (like TPC) are not sufficient for analyzing the novel cloud services as they require static settings and do not consider metrics central to cloud computing such as robustness to node failures. Li et al. [136] discuss some QoS guarantees and optimizations for the cloud. Ristenpart et al. focus on security aspects and conclude that fundamental risks arise from sharing physical infrastructure between mutually distrusful users [158]. Cryans et al. [104] compare the cloud

computing technology with database systems and propose a list of comparison elements. Kossmann et al. [134] evaluate the cost and performance of different distributed database architectures and cloud providers. They mention the problem of performance variance in their study, but they do not evaluate it further. Other authors [116, 149] evaluate the different cloud services of Amazon in terms of cost and performance, but they do not provide any evaluation of the possible impact that performance variance may have on users applications. Dejun et al. [107] also study the performance variance on EC2, but they only focus on an application level (MySQL, Tomcat performance). Therefore, they do not provide detailed insight to the source of the performance variance issue.

Finally, new projects that monitor the performance of clouds have recently emerged. For example, CloudClimate [29] and CloudKick [30] already perform performance monitoring of different clouds. EC2 also offers CloudWatch [55] which provides monitoring for *Amazon Web Services* cloud resources. However, none of the above works focuses on evaluating the possible performance variability in clouds or even give hints on how to reduce this variability.

[145, 133] try to port the idea of hardware counters to virtual machines.

[92] provides an overview of current monitoring solutions for cloud infrastructure. Most tools focus on detecting failures or monitoring SLAs. The authors do not discuss the impact for applications.

[88] examines the performance problems of running MapReduce on heterogenous clusters. They describe two main problem: (1) Fast nodes stealing (remote) map task from slow nodes resulting in network traffic. This is especially problematic as stealing occurs at the end of the reduce phase and hence network traffic is conflicting with shuffle network traffic. (2) Reduce key distribution is a problem as usually each node receives the same amount of reduce keys (assuming a single Reduce wave). The authors also propose a prototype Tarazu dealing of this problems by chaining the scheduling policies and reduce key distribution. Does not consider the HDFS performance and has quite very different node types (8 Xeon cores with 48GB Ram vs. 2 Core Atoms with 4GB Ram). Does not consider varying performance over time and assumes disk performance to be stable (not necessarily true in Cloud Setting).

LATEh [174] explores Hadoop's problem of speculative execution in heterogeneous environments. Hadoop assumes equal task progress across nodes. In

order to select tasks for speculative execution [1] Hadoop compares each tasks progress to the average progress. This behavior leads to problems in clusters with varying performance across nodes as too many speculative tasks are spawned degrading the overall performance below a scheduling policy without speculative reexecution. A second problem is that Hadoop randomly selects the speculative task among the slow running tasks. LATE deals with these problems by having an upper limit on the number of speculative running tasks and selecting the task to reexecute by the estimated remaining runtime.

[171] considers the effects of data-placement in heterogeneous clusters on MapReduce job runtime. The distribute the data blocks according to the compute-capicity of each node which is determined by previously executed MapReduce jobs. The paper does neither consider runtime variance over a clusters lifetime nor upload speed.

[1] Hadoop speculativly executions slow running tasks as the original task might be running on a failed or slow straggler node.

3.3 Experimental Setup

To evaluate the performance of a cloud provider, one can run typical cloud applications such as MapReduce [105] jobs, which are frequently excecuted on clouds, or databases applications. Even though these applications are a relevant measure to evaluate how well the cloud provider operates in general, we also wanted a deeper insight of application performance. This is why we focus on a lower level benchmark and hence measure the performance of individual components of the cloud infrastructure. Besides a deeper understanding of performance, measuring at this level also allows users to predict performance of a new application to a certain degree. To relate these results to real data intensive applications, we analyze the impact of the size of virtual clusters on variance and the impact on MapReduce jobs. In the following, we first discuss the different infrastructure components and aspects we focus on and then present the benchmarks and measures we use in our study.

3.3.1 Components and Aspects

We need to define both the set of relevant performance indicators and the relevant dimensions. In other words, we have to answer the following two important questions:

What are relevant indicators for performance? We focus on the following indicators that may considerably influence the performance of actual applications (we discuss benchmark details in Section 3.3.2).

1. *Instance startup* is important for cloud applications in order to quickly scale up during peak loads,
2. *CPU* is a crucial component for many applications,
3. *Memory speed* is crucial for any application, but it is even more important for data-intensive applications such as DBMSs or MapReduce,
4. *Disk I/O (sequential and random)* is a key component because many cloud applications require instances to store intermediate results on local disks if the input data may not be processed in main memory or for fault-tolerance purposes, such as MapReduce,
5. *Network bandwidth between instances* is quite important to consider because cloud applications usually process large amounts of data and exchange them through the network,

	CPU [Ubench]	Memory [Ubench]	Sequential Read [KB/second]	Random Read [seconds]	Network [MB/second]
Mean	1,248,629	390,267	70,036	215	924
Min	1,246,265	388,833	69,646	210	919
Max	1,250,602	391,244	70,786	219	925
Range	4,337	2,411	1,140	9	6
COV	0.001	0.003	0.006	0.019	0.002

Table 3.1: Physical Cluster: Benchmark results obtained as baseline

6. *S3 access from outside of Amazon* is important because most users first upload their datasets to S3 before running their applications in the cloud.

Across which dimensions do we measure these indicators? For each of the previous performance indicators there are three important aspects that may influence the performance. First: *Do small and large instances have different variations in performance?* Second: *Does the EU location suffer from more variance performance than the US location? Do different availability zones impact performance?* Third: *Does performance depend on the time of day, weekday, or week?*

Here, we study these three aspects and provide an answer to all these questions.

3.3.2 Benchmarks Details

We now present in more detail the different benchmarks we use for measuring the performance of each component.

Instance Startup. To evaluate this component, we measure the elapsed time from the moment a request for an instance is sent to the moment that the requested instance is available. To do so, we check the state of any starting instance every two seconds and stop monitoring when its status changes to "running".

CPU. To measure CPU performance of instances, we use the *Unix Benchmark Utility* (Ubench) [76], which is widely used and stands as the definitive Unix synthetic benchmark for measuring CPU (and memory) performance. Ubench provides a single CPU performance score by executing 3 minutes of various concurrent integer and floating point calculations. In order to properly utilize multicore systems, Ubench spawns two concurrent processes for each CPU available on the system. As AWS started using the new EBS backed AMIs, the CPU component of Ubench reported wrong results. Hence for the later

benchmark results we used unixbench a benchmark suite offering similar functionality. We indicate for each reported result which of the two benchmark was used.

Memory Speed. We also use the Ubench benchmark [76] to measure memory performance. Ubench executes random memory allocations as well as memory to memory copying operations for 3 minutes concurrently using several processes. The result is a single memory performance score. As the memory component of Ubench is also working for the new AWS AMIs, we used Ubench for all Memory speed measurements.

Disk I/O. To measure disk performance, we use Bonnie++ benchmark which is a disk and filesystem benchmark. Bonnie++ is a c++ implementation of Bonnie [23]. In contrast to Ubench, Bonnie++ reports several numbers as results. These results correspond to different aspects of disk performance, including measurements for *sequential reads*, *sequential writes*, and *random seeks*, in two main contexts: *byte by byte I/O* and *block I/O*. For further details please refer to [23]. In our study, we report results for sequential reads/writes and random reads block I/O, since they are the most influencing aspects in database applications.

Network Bandwidth. We use the Iperf benchmark [48] to measure network performance. Iperf is a modern alternative for measuring maximum TCP and UDP bandwidth performance developed by NLANR/DAST. It measures the maximum TCP bandwidth, allowing users to tune various parameters and UDP characteristics. Iperf reports results for bandwidth, delay jitter, and datagram loss. Unlike other network benchmarks (e.g., Netperf), Iperf consumes less system resources, which results in more precise results.

S3 Access. To evaluate S3, we measure the required time for uploading a 100 MB file from one unused node of our physical cluster at Saarland University (which has no network contention locally) to a newly created bucket on S3 (either in US or EU location). The bucket creation time and deletion time are included in the measurement. It is worth noting that such a measurement also reflects the network congestion between our local cluster and the respective Amazon datacenter.

3.3.3 Benchmark Execution

We ran our benchmarks two times every hour during 31 days (from December 14th 2009 to January 12th 2010) on small and large instances. The reason

for making such a long measurements is because we expected the performance results to vary considerably over time. This long period of testing also allows us to do a more meaningful analysis of the system performance of Amazon EC2. We have even one month more of data, but we could not see any additional patterns than those presented here[2]. We shut down all instances after 55 minutes, which allowed us to enforce Amazon EC2 to create new instances just before running again all benchmarks. The main idea behind this is to better distribute our tests over different computing nodes and hence to get a real overall measure for each of our benchmarks. To avoid that benchmark results were impacted by each other, we sequentially ran all benchmarks so as to ensure that only one benchmark was running at any time. Notice that, as sometimes a single run can take longer than 30 minutes, we ran all benchmarks only once in such cases. To run the Iperf benchmark, we synchronized two instances just before running it, because Iperf requires two idle instances. Furthermore, since two instances are not necessarily in the same availability zone, network bandwidth is very likely to be different. Thus, we ran different experiments for the case when two instances are in the same availability zone and when they are not.

3.3.4 Experimental Setup

In order to measure IaaS cloud variance we first ran experiments on Amazon EC2 using one small standard instance and one large standard instance in both locations US and EU. For both types of instances we used a Linux Fedora 8 OS. For each instance type we created one *Amazon Machine Image* per location including the necessary benchmark code. We used standard instances local storage and *mnt* partitions for both types when running Bonnie++.

To compare EC2 results with a meaningful baseline, we also ran all benchmarks — except instance startup and S3 — in our local cluster having physical nodes. It has the following configuration: one 2.66 GHz Quad Core Xeon CPU running 64-bit platform with Linux openSuse 11.1 OS, 16 GB main memory, 6x750 GB SATA hard disks, and three Gigabit network cards in bonding mode. As we had full control of this cluster, there was no additional workload on the cluster during our experiments. Thus, this represents the best case scenario, which we consider as baseline.

[2]The entire dataset is publicly available on the project website [55].

As cloud technology and platforms are changing rapidly we repeated the measurement three years after the original experiments from 2009/2010 in order to see whether there had been a change in performance variance. Note that as the technology for the AWS AMIs changed in that timeframe we had to use another CPU benchmark to rerun of the experiments.

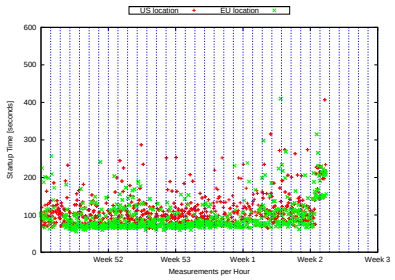
In order to establish the best case for cloud variance, i.e., without any contention on the physical machine we also executed the experiments on AWS Dedicated Instances [16]. The physical hardware for such instances is dedicated to a single customer only and hence there is no contention by other users. Note that it is still possible for a single user to allocate several virtual dedicated instances on the same physical hardware. In addition to the best case scenarios, we also allocated up to four instances on the same physical hardware and executed the benchmarks in parallel in order to simulate contention by different users.

Next, we examined whether there are any differences in the performance variance between different IaaS providers. In addition to EC2 we allocated instances at the Microsoft Azure Cloud [56] and Google Compute Cloud [42] . In case of Microsoft Azure we used the small instance type which has a 1.6 GHz CPU and 1.75 GB of RAM in the East US region. On Google's Compute Engine we allocated the n1-standard-1-d instance type which has 1 virtual core (equivalent of a single hyperthread on a 2.6GHz Xeon), 3.75GB of RAM in the US region. Note that the different providers are using different virtualization technologies and the results are also influenced by the different Hypersivors: AWS uses Xen [25], Microsoft Azure uses their own Windows Azure hypervisor (which is based on HyperV [54]) and Google Compute Engine uses KVM [49].

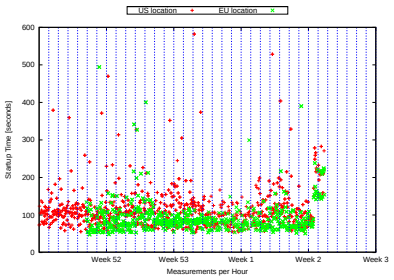
We used the default settings for our micro benchmarks Ubench, unixbench, Bonnie++, and Iperf. As Ubench performance also depends on compiler performance, we used gcc-c++ 4.1.2 on all Amazon EC2 instances and all physical nodes of our cluster. Finally, as we allocate instances in different timezones we decided to use *CET* as the coordinated time for presenting results.

3.3.5 Measure of Variation

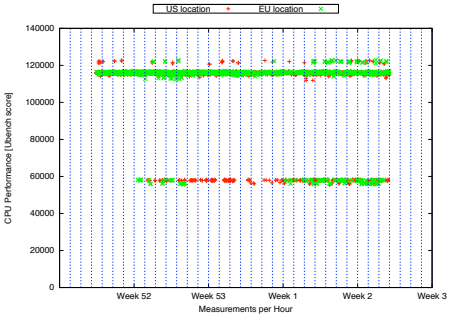
Let us now introduce the measure we use to evaluate the variance in performance. There exist a number of measures to represent this: range, interquartile range, and standard deviation among others. The standard deviation is a



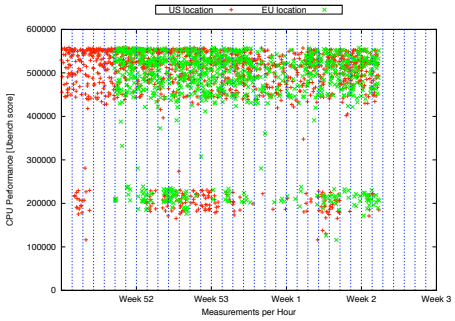
(a) Startup time small instances
$\overline{x}$ =107, COV: 1.14



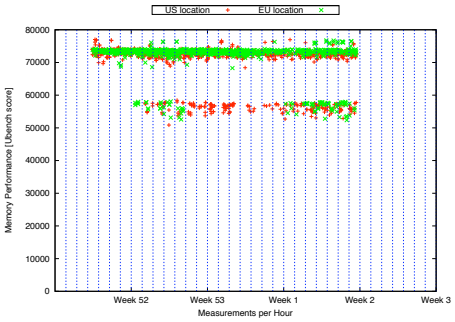
(b) Startup time large instances
$\overline{x}$ =115, COV: 1.73



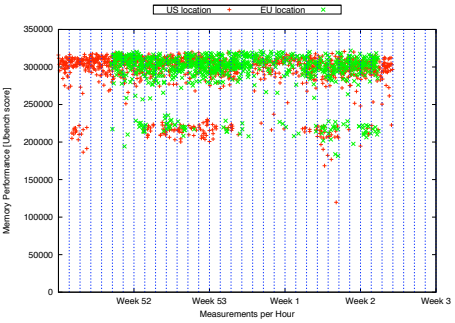
(c) CPU perf. on small instances
$\overline{x}$: 116,167, COV: 0.21



(d) CPU perf. on large instances
$\overline{x}$: 465,554, COV: 0.24



(e) Memory perf. on small instances
$\overline{x}$: 70,558, COV: 0.08



(f) Memory perf. on large instances
$\overline{x}$: 291,305, COV: 0.10

Figure 3.2: EC2: Benchmark results for CPU and memory.

widely used measure of variance, but it is hard to compare for different measurements. In other words, a given standard deviation value can only indicate how high or low the variance is in relation to a single mean value. Furthermore, our study involves the comparison of different scales. For these two reasons, we consider the *Coefficient of Variation* (COV), which is defined as the ratio of the standard deviation to the mean. Since we compute the COV over a sample of results, we consider the *sample standard deviation*. Therefore, the COV is formally defined as follows,

$$COV = \frac{1}{\overline{x}} \cdot \sqrt{\frac{1}{N-1} \cdot \sum_{i=1}^{N} (x_i - \overline{x})^2}$$

Here N is the number of measurements; $x_1, .., x_N$ are the measured results; and $\overline{x}$ is the mean of those measurements. Note that we divide by $N-1$ and not by N, as only $N-1$ of the N differences $(x_i - \overline{x})$ are independent [129].

In contrast to the standard deviation, the COV allows us to compare the degree of variation from one data series to another, even if the means are different from each other.

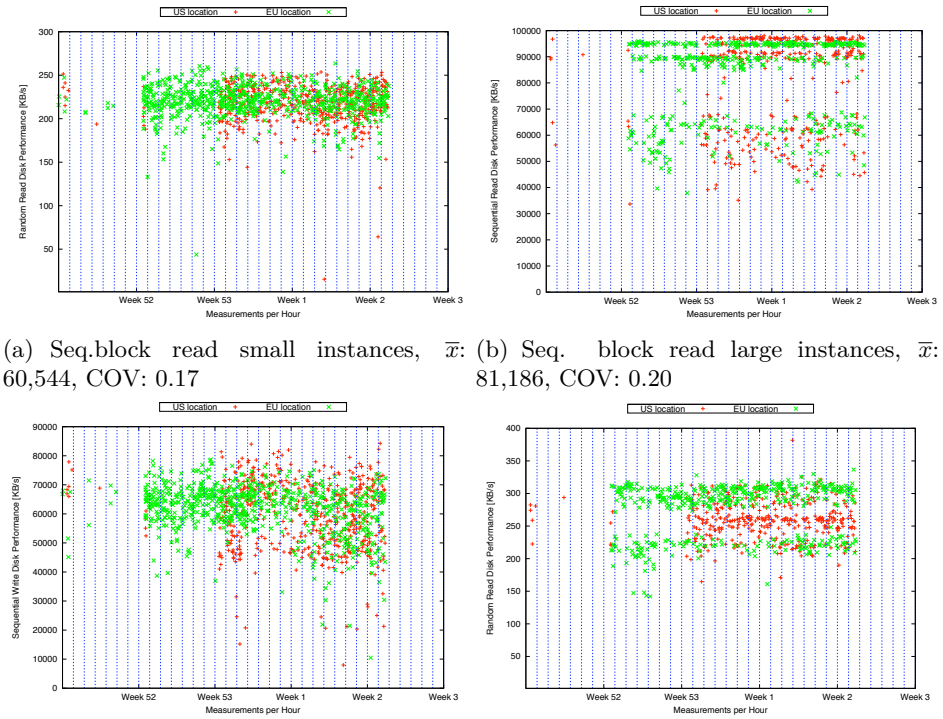
3.4 Results for Microbenchmarks

We ran our experiments with one objective in mind: to measure the variance in performance of EC2 and analyze the impact it may have on real applications. With this aim, we benchmarked the components as described in Section 3.3. We show all baseline results in Table 3.1. Recall that baseline results stem from benchmarking the physical cluster we described in Section 3.3.4.

3.4.1 Startup Time

Applications have to scale quickly in order to adjust to varying workload and thus save money. Thus, in this section, we evaluate this feature on Amazon EC2. We illustrate the results for small instances in Figure 3.2(a) and for large instances in Figure 3.2(b). The results show that the startup time is 107 seconds for small instances and 115 seconds for large instances, respectively. This is acceptable for several applications. The minimum startup time is about 50 seconds. Probably, Amazon uses the same allocation procedure for different

instances and what changes is only the amount of resources that EC2 has to allocate. The COV results strengthen this: the COV is 114% for small and 173% for large instances. A reason for this could be that it is simply harder for EC2 to allocate a virtual large instance having four times the computing power of a small instance. Notice that for these experiments we do not report baseline results because our cluster does not allocate virtual nodes and hence there is no startup time to measure.



(a) Seq.block read small instances, $\overline{x}$: 60,544, COV: 0.17

(b) Seq. block read large instances, $\overline{x}$: 81,186, COV: 0.20

(c) Random read small instances, $\overline{x}$: 219, COV: 0.09

(d) Random read large instances, $\overline{x}$: 267, COV: 0.13

Figure 3.3: EC2: Disk IO results for sequential and random

3.4.2 CPU

The results of the Ubench benchmark for CPU are shown in Figures 3.2(c) and 3.2(d). These results show that the CPU performance for both instances varies considerably. We identify two bands: the first band is from 115,000 to 120,000 for small instances and from 450,000 to 550,000 for large instances; the second band is from 58,000 to 60,000 for small instances

and from 180,000 to 220,000 for large instances. Almost all measurements fall within one of these bands. The COV in large instances is also higher than for small instances: it is 24%, while for small instances it is 21%. Note that the mean for large instances $\overline{x} = 465,554$ over $\overline{x} = 116,167$ for small instances is 4.0076 which corresponds to the claimed CPU difference of factor 4 almost exactly. The COV of both instances is at least by a factor 200 worse than in the baseline results (see Table 3.1).

In summary, our results show that the CPU performance of both instances is far less stable as one would expect.

3.4.3 Memory Speed

The results of the Ubench results for memory speed are shown in Figures 3.2(e) and 3.2(f). Both figures show two bands of performance. Thus unlike for CPU performance, we can see two performance bands for *both* instance types. Small instances suffer from slightly less variation than large instances, i.e., a COV of 8% versus 10%. In contrast, the COV on our physical cluster is 0.3% only. In addition, for small instances the range between the minimum and maximum value is 26,174 Ubench memory units, while for our physical cluster it is only 2,411 Ubench memory units (Table 3.1). This is even worse for large instances: they have a range value of 202,062 Ubench memory units. Thus, also for memory speed the observed performance on EC2 is by at least an order of magnitude less stable than on a physical cluster.

3.4.4 Sequential and Random I/O

We measure disk IO from three points of view: sequential reads, sequential writes, and random reads. However, since the results for sequential writes and reads are almost the same, we only present sequential read results in Figure 3.3. We can see that the COVs of all these results are much higher than the baseline. For instance, Figure 3.3(a) shows the results for sequential reads on small instances. We observe that the measurements are spread over a wide range, i.e., a band from approximately 55,000 to 75,000. The COV is 17%, which is much higher than baseline results (see Table 3.1). Figure 3.3 shows an interesting pattern: the measurements for random I/O on large instances differ considerably from the ones obtained in the EU. One explanation for this might be different types of disk used in different data centers. Overall we observe COVs from 9% to 20%. In contrast, on our physical cluster we observe

COVs from 0.6% to 1.9% only. So again, the difference in COV is about one order of magnitude. We expect these high COVs to have a non-negligible effect when measuring applications performing large amounts of I/O-operations, e.g., MapReduce.

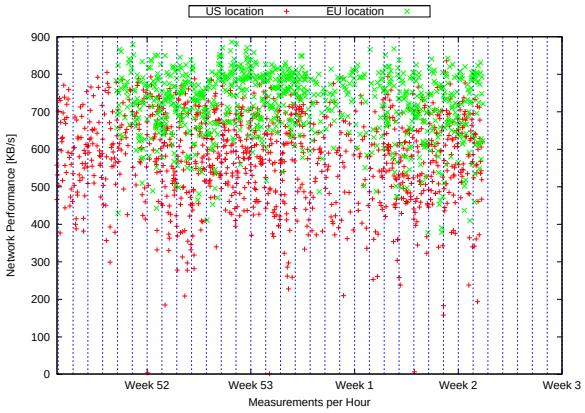


Figure 3.4: EC2: Network perf., $\overline{x}$ =640, COV: 0.19

3.4.5 Network Bandwidth

The results for network performance are displayed in Figure 3.4. The results show that instances in US location have slightly more oscillation in performance than in EU location. The COV for both instances is about 19% which is two orders of magnitude larger than the physical cluster having a COV of 0.2%. As for startup times, the performance variation of instances in US location is more accented than that of instances in EU location. In theory, this could be because EC2 in the EU is relatively new and the US location is more demanded by users. As a result, more applications could be running on US location than on EU location and hence more instances share the network. However, we do not have internal information from Amazon to verify this theory. Again, we observe that the range of measurements is much bigger than for the baseline (Table 3.1): while the range is 6 KB/s in our physical cluster, it is 728 KB/s on EC2.

3.4.6 S3 Upload Speed

As many applications (such as MapReduce applications) usually upload their required data to S3, we measure the upload time to S3. We show these results in

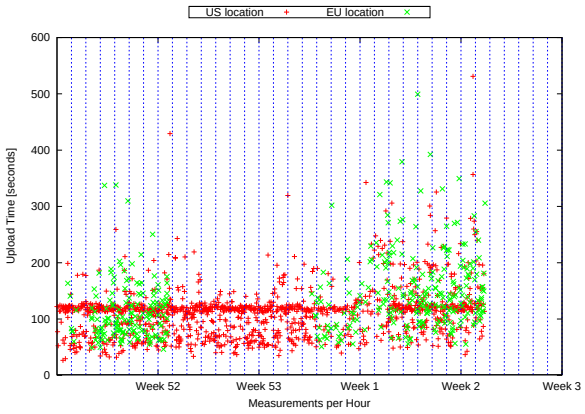


Figure 3.5: EC2: S3 upload time, $\overline{x}$ =120, COV: 0.54

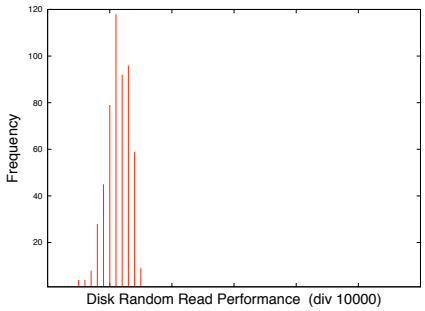
		Cutoff	percent of measurements in Lower Segment
CPU	**Large**	320,000	22
	Small	75,000	17
Memory	**Large**	250,000	27
	Small	65,000	36

Table 3.2: EC2: Distribution of measurements between two bands for Ubench benchmark

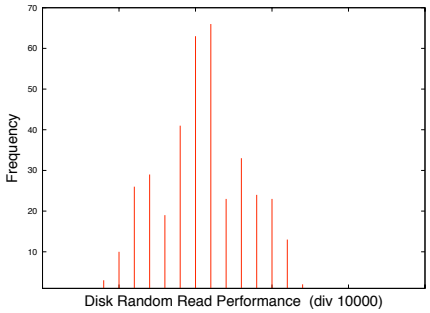
Figure 3.5. The mean upload time is $\overline{x}$ =120 with a COV of 54%. As mentioned above the COV may be influenced by other traffic on the network not pertaining to Amazon. Therefore we only show this experiment for completeness. Observe that during weeks 53 and 1 there is no data point for EU location. This is because Amazon EC2 threw us a bucket[3] exception due to a naming conflict, which we fixed later on.

We observed in previous section that, in general, Amazon EC2 suffers a lot from a high variance in its performance. In the following, we analyse this in more detail. In addition, Section 3.5.7 contains a variability decomposition analysis.

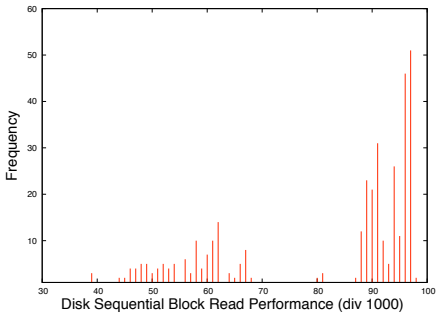
[3]Generally speaking, a bucket is a directory on the Cloud.



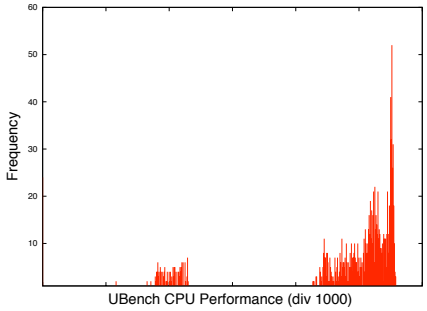
(a) US small instance random I/O.



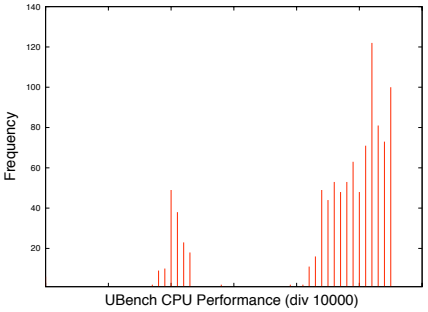
(b) US large instance random I/O.



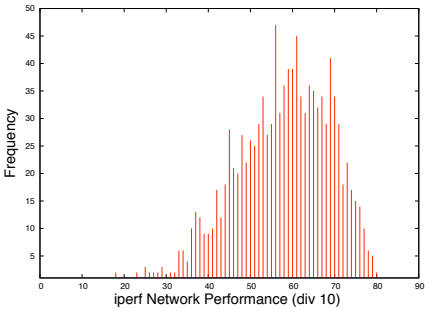
(c) US large instance sequential I/O perf.



(d) US large instance Ubench CPU perf.



(e) EU large instance Ubench CPU perf.



(f) US network performance.

Figure 3.6: Distribution of measurements for different benchmarks

3.5 Analysis

After a first discussion of the micro benchmark results, let us analyse a number of interesting points further.

3.5.1 Distribution Analysis

Bands. A number of previous results (e.g., Figure 3.2(c)) showed two performance bands in which measurements are clustered. In this section we quantify the size of some of those bands. Table 3.2 shows how measurements are clustered when dividing the domain of benchmark results into two partitions: one above and one below a cutoff line. The *Cutoff* column expresses the Ubench unit that delimits the two bands and the *Lower Segment* column presents the percent of measurements that fall into the lower segment. We can see that for CPU on large instances 22% of all measurements instances fall into the lower band having only 50% the performance of the upper band. In fact, several Amazon EC2 users already experienced this problem [4]. For memory performance we may observe a similar effect: 27% of the measurements are in the lower band on large instances, 36% on small instances. Thus, the lower band represents a considerable portion of the measurements.

Distributions. To analyze this in more detail, we also study the distribution of measurements of the different benchmarks. We show some representative distributions in Figure 3.6. We observe that measurements for Figures 3.6(a)& 3.6(b), US random I/O, and Figure 3.6(f), US network performance, are normally distributed. All other distributions show two bands. Most of these bands do not follow a normal distribution. For instance, Figure 3.6(c) depicts sequential I/O for large US instances. We see two bands: one very wide band spanning almost the entire domain from 45,000 to 70,000 KB/s. In addition, we see a narrow band from 87,000 to 97,000 KB/s. None of the individual bands seems to follow a normal distribution. A possible explanation for this distribution might be cache effects, e.g., warm versus cold cache. However, a further analysis of our data could not confirm this. We thus carry out a further analysis in the following sections.

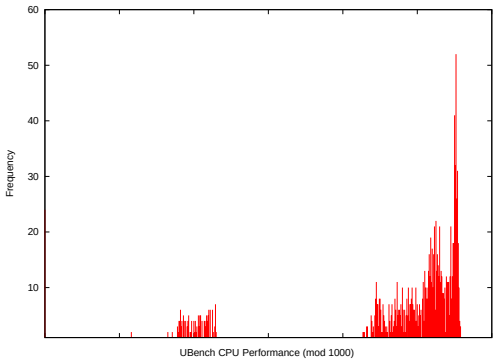


Figure 3.7: CPU Large Histogram.

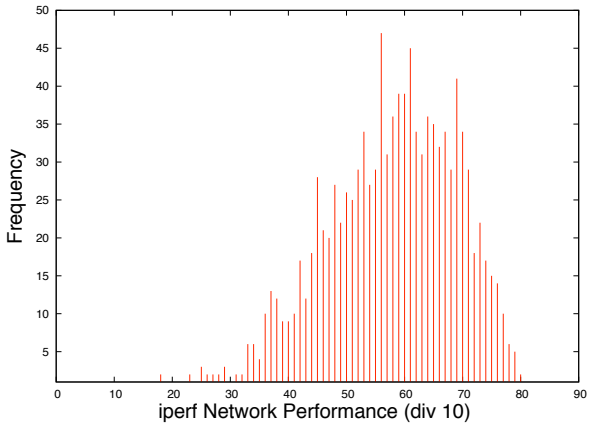


Figure 3.8: Network US Histogram.

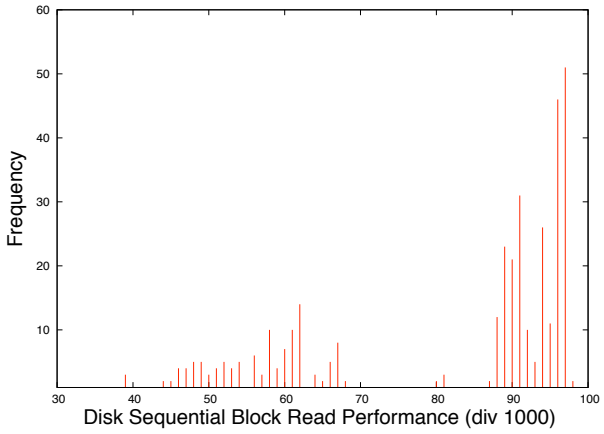


Figure 3.9: Sequential IO US Histogram.

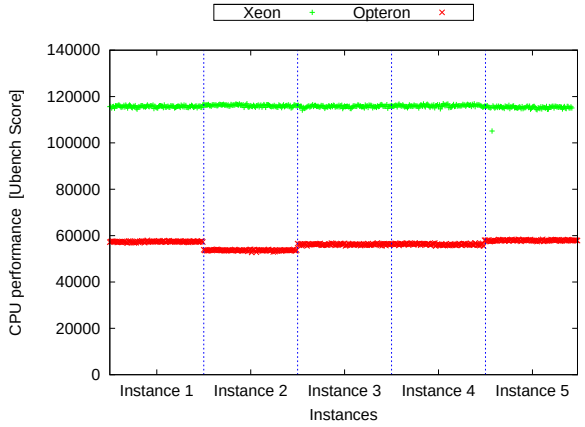


Figure 3.10: Variability of CPU perf. per processor.

3.5.2 Variability over Processor Types

As indicated in [2], the EC2 infrastructure consists of two different systems — at least of two processor types[4]. We conducted an additional Ubench experiment so as to analyze the impact on performance that these different system types might have. To this end, we initialized 5 instances for each type of system and ran Ubench 100 times on each instance.

We illustrate the results in Figure 3.10. These results explain surprisingly in great part the two bands of CPU performance we observed in Section 3.4.2.

[4]This can be identified by examining the `/proc/cpuinfo` file where the processor characteristics are listed.

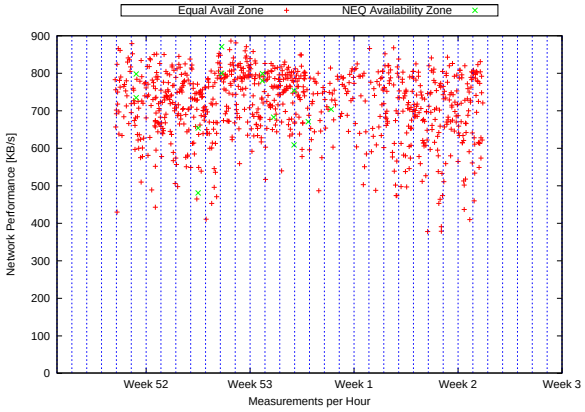


Figure 3.11: Variability of network perf. for EU.

This is surprising because both instances are assumed to provide the same performance. Here, we observe that the Opteron processor corresponds to the lower band while the Xeon processor corresponds to the higher band. The variance inside these bands is then much lower than the overall variation: a COV of 1% for Xeon and a COV of 2% for Opteron — while the COV for the combined measurements was 35%. Furthermore, we could observe during this experiment that the different bands of memory performance could also be predicted using this distinction. The corresponding COV decreased from 13% for combined measurements to 1% and 6%, respectively, for Xeon and Opteron processors. Even for disk performance we found similar evidence for two seperate bands — again depending on processors.

3.5.3 Network Variability for Different Locations

As described in Section 3.3.4, we did not explicitly consider the availability zone as a variable for our experimental setup and hence we did not pay too much attention on it in Section 3.4. Amazon describes each availability zone as "distinct locations that are engineered to be insulated from failures in other availability zones" [2].

In this section we analyze the impact of using different availability zones for the network benchmark. Our hypothesis is that whenever the two nodes running the network benchmark are assigned in the same availability zone, the network performance should be better; vice versa when the two nodes are assigned to

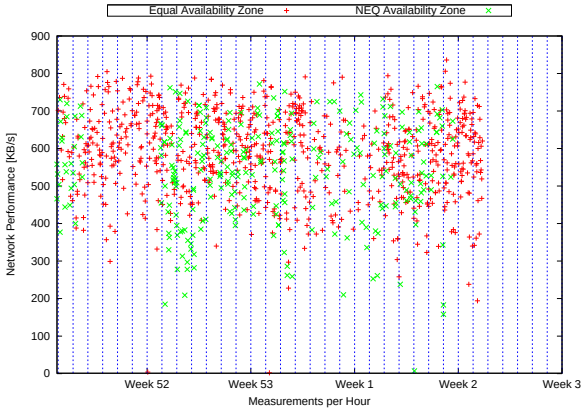


Figure 3.12: Variability of network perf. for US.

different availability zones, their network benchmark result should be worse. If this holds, we could conclude that different availability zones correspond to units (possibly physical units) where the network connections inside units are better than among those units.

Figure 3.11 shows the results for the EU. Here red indicates that both nodes run in the same availability zone, green indicates they run in different availability zones. Unfortunately, we observe that in the EU most instance pairs get assigned to the same availability zone. This changes however if we inspect the data for the US (see Figure 3.12). All measurements vary considerably. However, the measurements inside an availability zone have a mean of 588, the measurements among different availability zones have a mean of 540. Thus inside an availability zone the network performance is by 9% better. We validated this result with a t-test: the null hypotheses can be rejected at $p = 1.1853 \times 10^{-11}$.

We believe that the variability of network performance we could observe so far might stem from the scheduler policy of EC2 — which always schedules virtual nodes of a given user to different physical nodes. This is supported by Ristenpart et al. who observed that a single user never gets two virtual instances running on the physical node [158]. As a consequence, this results in more network contention.

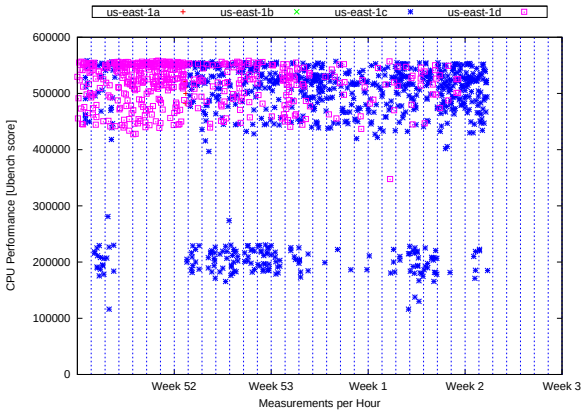


Figure 3.13: CPU distribution over different availability zones for large instances

3.5.4 CPU Variability for Different Availability Zones

In this section we analyze how different availability zones impact CPU performance. Figure 3.13 shows the same data as Figure 3.2(d). However, in contrast to the latter, we only show data from the US and depict for each measurement its availability zone. We observe that almost none of the nodes was assigned to us-east-1a or us-east-1b. All nodes were assigned to us-east-1c and to us-east-1d. Interestingly, we observe that all measurements in the second lower band belong to us-east-1c. Thus, if we ran all measurements on us-east-1d only, all measurements would be in one band. Furthermore the COV would decrease. These results confirm that indeed availability zones influence performance variation. In fact, we also observed the same influence for small instances and for other benchmarks as well, such as network performance. We believe that this is, in part, because some availability zone mainly consist of one processor type only, which in turn decreases the performance variability as discussed in Section 3.5.2. Hence, one should specify the availability zone when requesting its instance.

3.5.5 Variability over Different Instance Types

In this section, we examine the impact of different instance types on performance. Figure 3.14 shows the same data as Figures 3.2(c) and 3.2(d). However, in contrast to the latter we do not differentiate by location. As observed in

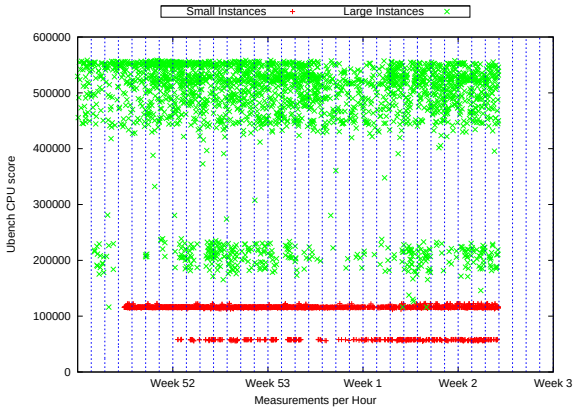


Figure 3.14: CPU score for different instance types

Figure 3.14 the mean CPU performance of a large instance is almost exactly by a factor 4 larger than for small instances. The standard deviation for large instances is about four times higher than for small instance. However, the COVs are comparable as the factor four is removed when computing the COV. The COV is 21% for small instances and 24% for large instances. It is worth noting that the small and large instances are actually based on different architectures (32/64 bit platforms, respectively), which limits the comparability between them. However, we also learn from these results is that for both instance types (small and large) several measurements may have 50% less CPU performance. instances are in the lower band? We answer this question in the next section.

3.5.6 Variability over Time

In previous section, we observed that, from the general point of view, most of the performance variation is independent of time. We now take a deeper look at this aspect by considering the COV for each individual weekday. Figure 3.15 illustrates the COV for individual weekdays for the Ubench CPU benchmark. As the COV values for other components, such as memory and network, are quite similar to those presented here, we do not display those graphs. At least for the US instances we observe a lower variation in CPU performance of about 19% on Mondays and weekends. From Tuesday to Friday the COV is above 26%. For the EU this does not hold. The small COV for US location on Monday strengthen this assumption since in Unite States it is still Sunday

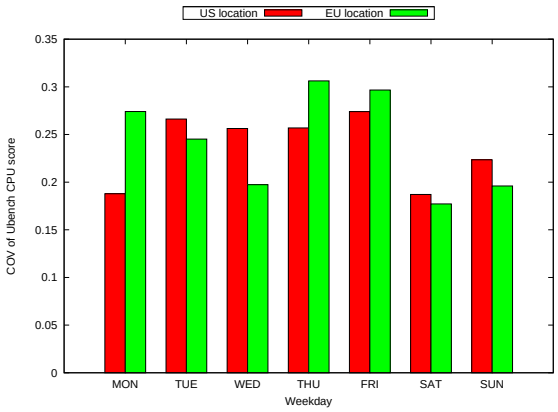


Figure 3.15: Variability of CPU perf. per weekday

(recall we use CET for all measurements). We believe the reason for this is that users mainly run their applications during their working time. An alternative explanation could be that people browse and buy less on Amazon and therefore Amazon assigns more resources to EC2.

3.5.7 Variability Decomposition

We have seen so far that Amazon EC2 suffers from a high variation in its performance. In this section, we decompose the performance variability in order to better understand such a variation in performance. For this, we decompose the COV analysis into two parts with the aim of identifying from where the variance arises. To this end, we analyze the data using four different aggregation-levels: (i) day, (ii) hour, (iii) hour of the day, (iv) day of the week. We partition our measurements $x_1, .., x_N$ into disjoint and complete groups $G_1, .., G_k$, $k \leq N$ where each group corresponds to an aggregate of an aggregation level (i)–(iv). Then, we analyze the aggregates in two ways:

(1.) between aggregation-levels. We compute the mean for each aggregate; then we compute the COV of all means. In other words, for each group G_i we compute its mean $\overline{x}_{G_i}$. For all means $\overline{x}_{G_i}$ we then compute the $\text{COV}_{\overline{x}_{G_i}}$. The idea of this analysis is to show the amount of variation *among* different aggregation-levels. For instance, we may answer questions like 'does the mean change from day to day?'. Table 3.3 gives an overview of the results.

(2.) in aggregation-levels. We compute the COV for each aggregate; then

Small instances								
$\text{COV}_{\overline{x}_{G_i}}$	All		Day		HourOfDay		DayOfWeek	
	US	EU	US	EU	US	EU	US	EU
Startup Time	1.399	0.439	0.306	0.263	0.211	0.121	0.122	0.079
S3 Upload Time	0.395	0.481	0.132	0.210	0.448	0.147	0.371	0.094
Bonnie Seq. Out	0.199	0.136	0.080	0.055	0.049	0.029	0.030	0.015
Bonnie Ran Read	0.102	0.085	0.043	0.018	0.037	0.017	0.019	0.002
CPU	0.237	0.179	0.075	0.033	0.031	0.004	0.032	0.028
Memory	0.097	0.070	0.036	0.028	0.014	0.015	0.013	0.011
Iperf	0.201	0.124	0.045	0.028	0.044	0.026	0.026	0.021

Large instances								
$\text{COV}_{\overline{x}_{G_i}}$	All		Day		HourOfDay		DayOfWeek	
	US	EU	US	EU	US	EU	US	EU
Startup Time	2.022	0.479	0.330	0.231	0.292	0.087	0.163	0.094
S3 Upload Time	0.395	0.481	0.132	0.210	0.448	0.147	0.371	0.094
Bonnie Seq Out	0.226	0.191	0.091	0.069	0.060	0.038	0.070	0.037
Bonnie Ran Read	0.043	0.056	0.043	0.056	0.027	0.030	0.019	0.031
CPU	0.230	0.243	0.078	0.079	0.031	0.033	0.032	0.028
Memory	0.108	0.097	0.038	0.032	0.020	0.016	0.014	0.021
Iperf	0.201	0.124	0.045	0.028	0.044	0.026	0.026	0.021

Table 3.3: between aggregation-level analysis: COV of mean values

we compute the mean of all COVs. In other words, for each group G_i we compute its COV_{G_i}. For all COV_{G_i} we then compute the mean $\overline{x}_{\text{COV}_{G_i}}$. The idea is to show the mean variation *inside* different aggregation-levels. For instance, we may answer questions like 'what is the mean variance within a day?' Table 3.4 shows the results.

For better readability, all results in-between 0.2 and 0.4 are shown in orange text color, and all results greater equal 0.4 are shown in red text color.

We first discuss results in Table 3.3 focussing on some of the numbers marked red and orange. The results for small and large instances are very similar. Therefore we focus on discussing small instances. For small instances we observe that the mean startup time varies considerably for both US and EU: respectively 139% and 43.9% of the mean value. When aggregating by HourOfDay S3 upload times vary by 44.8% in the US but only 14.7% in the EU. When aggregating by DayOfWeek we observe that mean S3 upload times also vary by 37.1% in the US but only by 9.4% in the EU. Thus, the weekday to weekday performance is more stable in the EU. CPU performance for a particular

Small instances								
$\overline{x}_{\mathrm{COV}_{G_i}}$	All		Day		HourOfDay		DayOfWeek	
	US	EU	US	EU	US	EU	US	EU
Startup Time	1.399	0.439	0.468	0.357	0.636	0.429	0.914	0.436
S3 Upload Time	0.395	0.481	0.356	0.406	0.371	0.448	0.383	0.472
Bonnie Seq Out	0.199	0.136	0.177	0.123	0.199	0.132	0.198	0.135
Bonnie Rand In	0.102	0.085	0.091	0.074	0.096	0.081	0.019	0.085
CPU	0.237	0.179	0.207	0.162	0.229	0.173	0.228	0.174
Memory	0.097	0.070	0.085	0.050	0.096	0.065	0.097	0.068
Iperf	0.201	0.124	0.204	0.121	0.196	0.122	0.205	0.123

Large instances								
$\overline{x}_{\mathrm{COV}_{G_i}}$	All		Day		HourOfDay		DayOfWeek	
	US	EU	US	EU	US	EU	US	EU
Startup Time	2.022	0.479	0.330	0.330	0.812	0.416	1.192	0.451
S3 Upload Time	0.395	0.481	0.356	0.406	0.371	0.448	0.383	0.472
Bonnie Seq Out	0.226	0.191	0.227	0.183	0.227	0.192	0.317	0.189
Bonnie Rand In	0.114	0.149	0.112	0.141	0.114	0.148	0.113	0.147
CPU	0.237	0.243	0.208	0.222	0.235	0.242	0.236	0.242
Memory	0.108	0.097	0.093	0.085	0.105	0.096	0.107	0.095
Iperf	0.201	0.124	0.204	0.121	0.196	0.122	0.205	0.123

Table 3.4: in aggregation-level analysis: Mean of COV values

hour of the day is much more stable: 3.1% in the US and 0.4% in the EU. In addition, CPU performance for a particular day of the week is also remarkably stable: 3.2% in the US and 2.8% in the EU. We conclude that means for different hours of the day, and different days of the week show little variation. Thus a particular hour of the day or day of the week does not influence the mean performance value. Table 3.4 shows results of the in-aggregation-level analysis. Here the results for small and large instances differ more widely. We focus again on small instances. For Startup Time we observe that the mean COV for Day is 46.8% in the US and 35.7% in the EU. If we aggregate by HourOfDay or DayOfWeek, we observe very high means of the COVs for both locations, for example up to 91.4% for DayOfWeek in the US. Thus, the mean variance for hours of the day and days of the week is very high. Still, the results in Table 3.3 show that the variance *among* the means of a particular hour of the day or day of the week is far less.

For CPU performance we observe that in the US the mean COV is above 20% when aggregating by Day, HourOfDay, or DayOfWeek. In other words, CPU performance varies considerably *inside* each aggregate over all aggregation-levels. The variation *among* different aggregation-levels is not that high any-

more as observed in Table 3.3. This indicates a certain stability of the results: the measurements *inside* a single aggregate may be enough to predict the mean CPU performance with little error. However, recall that for our data each aggregate consists of several individual measurements: Day: 84, HourOfDay: 62, and DayOfWeek: 96 measurements.

3.5.8 Revisiting Cloud Performance Variance

As the original variance measurements were taken in 2010 we were curious whether the variance situation had changed. So we measured the variance again between end of march and beginning of may 2013. Using the same setup as before, we allocated a small instance every hour at AWS. The results for CPU and IO performance are shown in FIgure 3.16 and the respective COV measures in Table 3.7. When comparing to the 2010 results we see that the variance for CPU and Write performance performance has decreased from a COV of 0.237 to 0.05 in case of the CPU performance and from 0.199 to 0.110 for the IO write performance. But the the IO read performance incurs with the new measurement even more variance than in 2010 (the COV increased from 0.154 to 0.45). Despite the lower CPU performance variance we can still clearly see different performance bands in Figure 3.16(a). For both the IO read in Figure 3.16(b) and write performance in Figure 3.16(c) there is a clear performance baseline and other measurements being spread around this baseline. Having identified the machine type[5] explaining a large factor of the variance in Section 3.5.2 we examined how different systems affected the performance for our new measurements. By checking the processor type we found a total of 4 different system configurations using the following processor types: E5-2650 with 2.00GHz, E5430 with 2.66GHz, E5507 with 2.27GHz , and E5645 with 2.40GHz. As shown in Table 3.6 the distribution between these different types is fairly uniform. We show the benchmark results by different processor type in Figure 3.17. In Table 3.7 we show the COV and average values for each processor type. For the CPU performance in Figure 3.17(a) the different processor types perfectly explain the three performance bands. The upper band contains only measurements of E5645 processor type.The medium band contains both measurements from the E5430 and E5507 processor type. All measurements from the E5-2650 processor type fall in the lower band. Considering the different processor types also decreases the CPU variance even more; when considering the different processor types the COV values are between

[5]The machine type is identified by the CPU type.

COV			
COV	CPU	Seq. Block Write	Seq Block Read
2010	0.237	0.199	0.154
2013	0.05	0.11	0.45

Table 3.5: Variance in 2010 and 2013

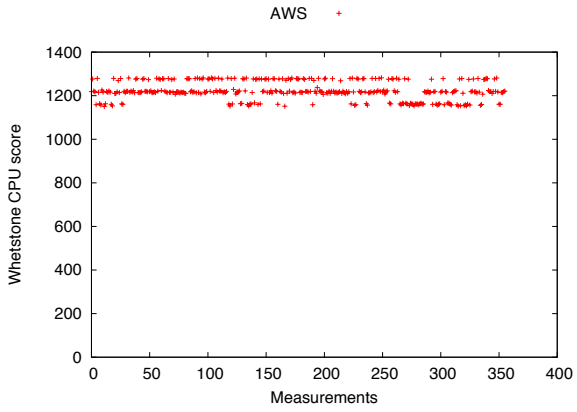
Processor Types		
CPU	GHz	Percentage
E5-2650	2.00	20.2%
E5430	2.66	25.2%
E5507	2.27	26.4%
E5645	2.40	27.8%

Table 3.6: Distribution of Processor Types

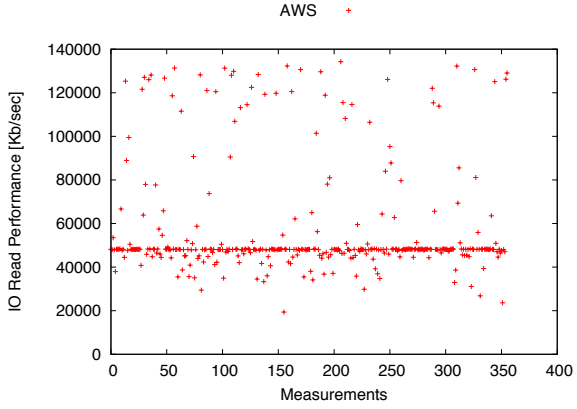
0.002 and 0.0033 as compared to an overall COV of 0.05. As we view the processor type as an identifier for the entire system we also look at the effect the clustering by processing type has on the IO read and write variance. We show the measurement by processor type in Figure 3.17(b) and Figure 3.17(c). Recall that in the IO read variance is higher for the 2013 measurement s than for the 2010 measurements. Using Figure 3.17(b) we can easily explain this behavior: For the E5-2650, E5507, and E5645 system the read performance is relative uniform (COVs between 0.06 and 0.11), but the E5430 measurements have a quite different characteristic. First of all the variance is much higher with a COV of 0.366 when compared to the other systems. As a second factor also the average performance is quite different. We can see in Table 3.7 that the average read performance is about twice as large as for the other systems. The likely explanation for this observation is that the E5430 systems use SSDs instead of HDDs. The performance variance in this case would result from the fact that the bonnie++ accesses a lot of individual blocks which can reduce the performance of SSDs.

Next, we look at the network bandwidth variability between two different instances. This is especially important for distributed processing frameworks such as MapReduce as they need to shuffle large amounts of data between nodes. In Figure 3.18 we show the results of measuring the network bandwidth among two different nodes. These nodes are for both providers always

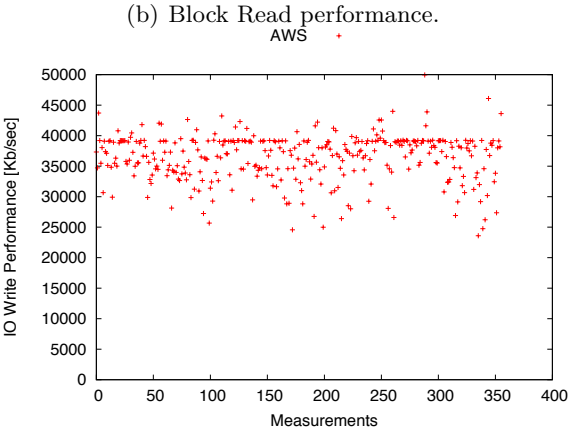
allocated in the same region. The bandwidth measurements from AWS show a common performance band between 300 to 400 Mbit/sec but also a large number of outliers. Note that most outliers have a higher bandwidth than the common performance band. On the other hand the measurements from Azure look more randomly distributed. This intuition is also confirmed by the COV measurements in Table 3.8. AWS's new COV value (0.355) are also much higher than the COV value from Azure (0.177) and also higher than the COV value (0.201) from the initial benchmarks in 2010. In order to understand this strange behavior we have to look into the definition of a region. For AWS a region consists of different Availability Zone which are equivalent to the notion of data center [20]. For Microsoft on the other hand a region is currently (as of autumn 2013) not further divided and hence corresponds to a datacenter [22]. This means in the case of AWS we are sometimes measuring the network bandwidth between different datacenter while in the case of Azure we are always measuring network bandwidth inside a single datacenter. If we only consider AWS measurements where both instances are located inside the same datacenter we obtain a COV of 0.169 which comparable to Azures COV value. Considering traffic between different datacenter also explains the common band we saw in Figure 3.18 for the AWS measurements. The common band between 300 to 400 Mbits is actually the traffic between two different datacenter while the measurements above are measurements inside the same datacenter. Still, why is AWS's COV value from the 2013 measurements higher than the ones from 2010. The most likely explanation is that Amazon simply build more datacenter [20] and hence the chance for two instance being allocated in different availability zones/datacenter inside the same region is much higher.



(a) CPU performance.

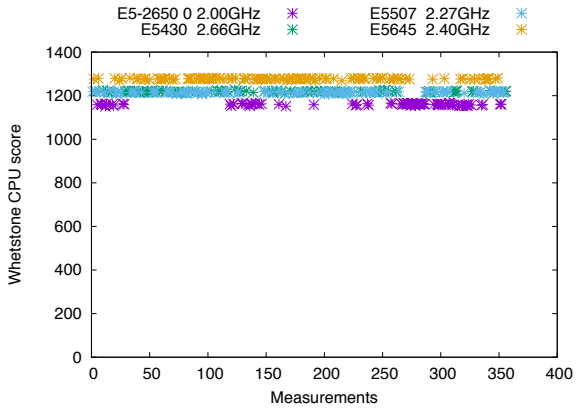


(b) Block Read performance.

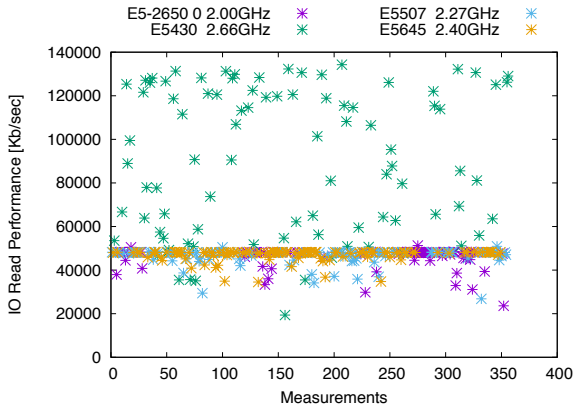


(c) Block Write performance.

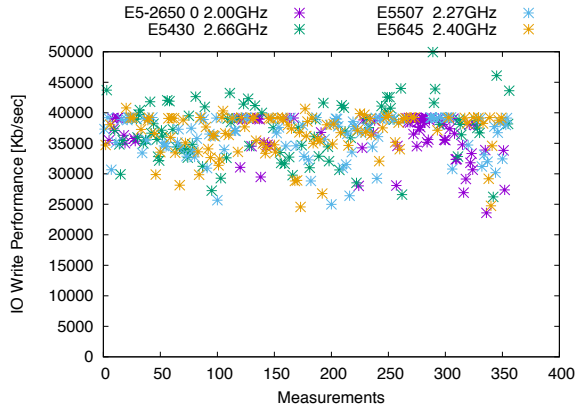
Figure 3.16: AWS rerun measurements



(a) CPU performance.



(b) Block Read performance.



(c) Block Write performance.

Figure 3.17: AWS rerun measurements by Processor Type

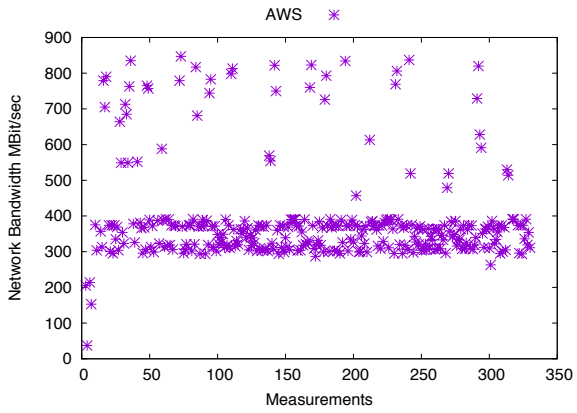
Measurements by Processor Type

COV	CPU				Seq. Write				Seq Read			
	E5-2650	E5430	E5507	E5645	E5-2650	E5430	E5507	E5645	E5-2650	E5430	E5507	E5645
	0.0033	0.0030	0.0027	0.0020	0.1008	0.1238	0.0955	0.0967	0.1153	0.3666	0.0876	0.0623
Average	CPU				Seq. Write				Seq Read			
	E5-2650	E5430	E5507	E5645	E5-2650	E5430	E5507	E5645	E5-2650	E5430	E5507	E5645
	1160	1218	1215	1277	35947	37056	36116	36399	45621	90349	46261	46857

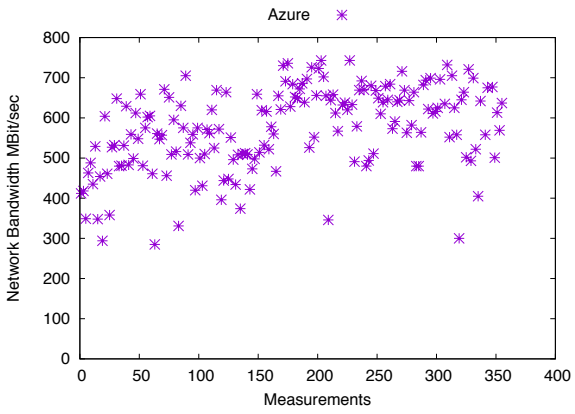
Table 3.7: Variance between different instances

COV	Network Bandwitdth
AWS 2013 (region)	0.355
Azure 2013 (region)	0.177
AWS 2010 (region)	0.201
AWS 2013 (availability zone)	0.169

Table 3.8: Network Bandwidth Variance



(a) AWS.



(b) Azure.

Figure 3.18: Network bandwidth performance

3.5.9 Long-Running Instance Variance over Time

In this section we examine the variance of a single long-running instance in contrast to the previous experiments where we looked at the variance between different instances. Therefore we allocated a small EC2 instance and measured continuously executed the standard set of benchmarks (see Section 3.3.2) every two hours over the timespan of one month from 13th of march to 13th of april 2013. The results for CPU performance are shown in Figure 3.19(a). The CPU performance is very homogeneous. This is also evident when looking at the COV measures in Table 3.9 directly. Here we can see that the COV for CPU performance is only at 0.002. This indicates that once a user acquires an instance the CPU performance is quiet reliable. This is mostly due to the fact that the CPU itself can be easily split between different virtual instances by either physical partitioning (each virtual instance receives a fixed number of cores) or by time wise splitting of CPU cycles. The performance CPU variance we saw in Section 3.4.2 is mostly caused the between-instance variance of different instances which can be partially explained by different hardware, especially processors (see Section 3.5.2).

The results for I/O read and write performance shown in Figure 3.19(c) and Figure 3.19(b) are slightly different. Also here we have a smaller COV of 0.065 and 0.045 when compared to the between-instance variance with a COV of 0.199 and 0.154, but the difference is not several orders of magnitude as in the CPU case. A potential explanation for this difference is, that I/O devices (either hard-disks directly or network attached storage) is shared between different instances. Hence, the I/O contention is a cause for performance variance.

When observing the I/O Write pattern over time one recognize again two different bands of performance. Recall that these bands cannot be explained by different underlying platforms we consider the same running instance [6]. The most likely cause here is I/O contention of other co-located instances. Interestingly we do not observe such distinct bands of performance for i/O read performance, despite a large variance of read performance. Note, that we omit the results for memory performance as there was little performance variance (COV of 0.097) in the initial experiments.

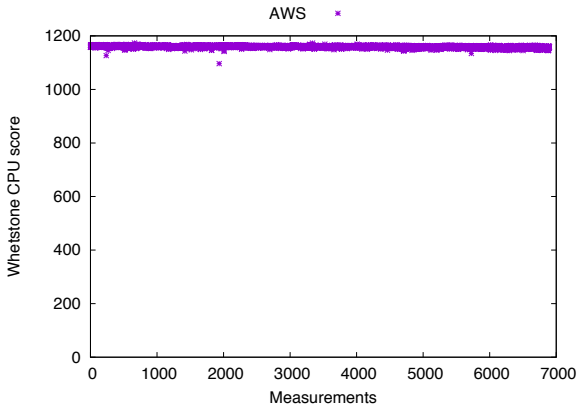
We repeated the same setup with a small instance allocated at Microsoft Azure. Here the timeframe was only a week from 13th to 19th of march 2013 as in

[6] Some virtualization solutions such Vmware Esx [78] allow the movement of running instances between nodes, but as of summer 2013 no major cloud vendor is known to move running instances.

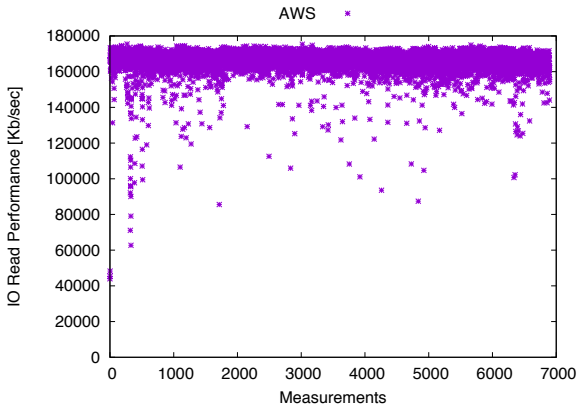
Single instance COV			
COV	CPU	Seq. Block Read	Seq Block Write
AWS EC2	0.002	0.065	0.045
Microsoft Azure	0.001	0.375	0.105
Across instance COV for comparison			
AWS EC2	0.237	0.199	0.154

Table 3.9: COV for different measures

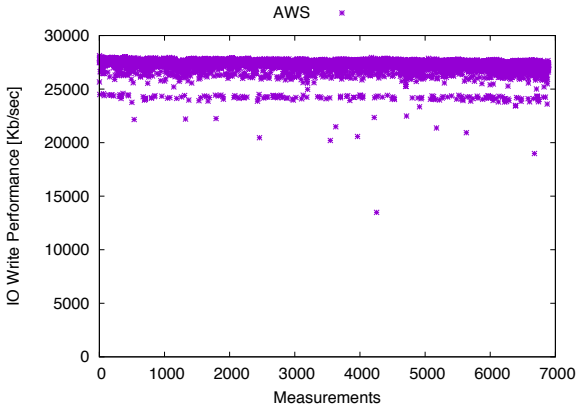
contrast to EC2 where we used a research we incurred actual cost here. The results for CPU, I/O Write performance and I/O Read performance are shown in Figure 3.20. We present the COV measures again in Table 3.9. When looking at the CPU variance the results for Azure are similar to those from AWS EC2. The COV is with 0.001 even lower. The picture changes when looking at the I/O performance: Here Azure offers an overall higher throughput, but at the cost of increased variance. The COV measures for I/O performance are 2-6 times larger than the same measures from AWS EC2. Its seems that Azure is employing a best effort approaches as we sometimes see outliers in both directions. For Azure we can recognize two different performance bands for the I/O Read performance in Figure 3.20(b). Similar the explanation for this behavior is most likely contention from other co-located instances. Recall that again we are dealing with two different Hypervisors, each having different I/O scheduling policies [148].



(a) CPU performance.

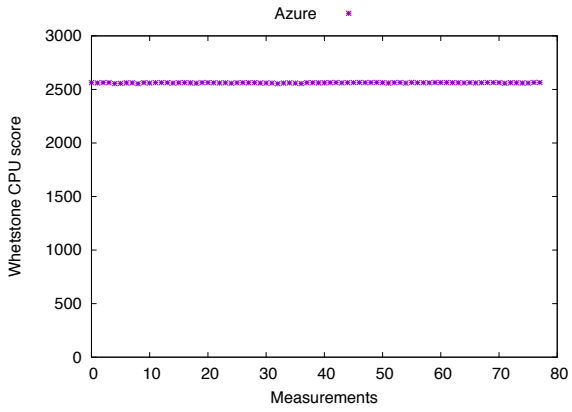


(b) Block Read performance.

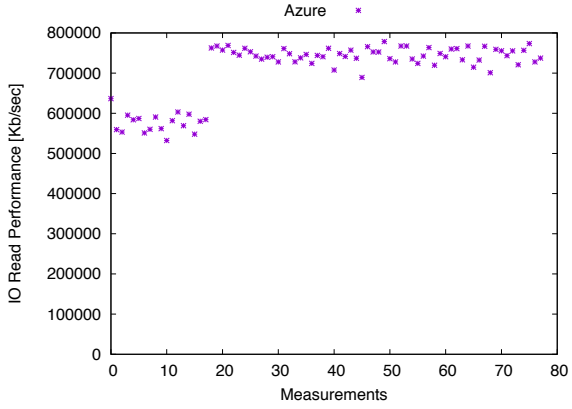


(c) Block Write performance.

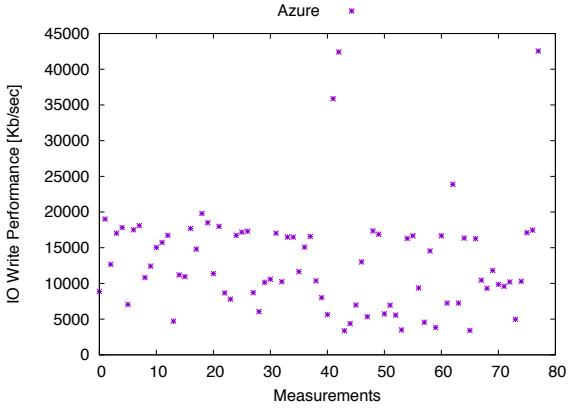
Figure 3.19: AWS fixed instance long term measures



(a) CPU performance.

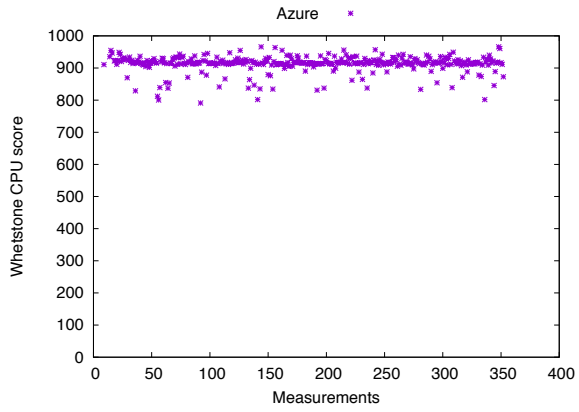


(b) Block Read performance.

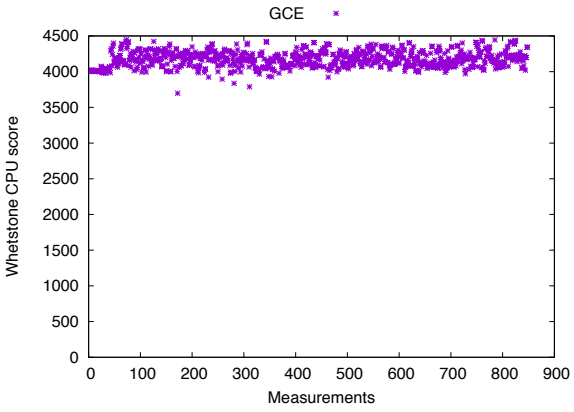


(c) Block Write performance.

Figure 3.20: Azure fixed instance long term measures



(a) Microsoft Azure.



(b) Google Compute Engine.

Figure 3.21: CPU Performance Variance across Vendors

3.5.10 Different Cloud Provider

As today there exists a large number of cloud providers, the reader might vendor whether the performance variance is solely a problem of Amazon EC2 or is present across all cloud providers. Therefore we examine Microsofts Azure [58] and Google Compute Engine [42] using the same set of benchmarks as described in Section 3.3.4. We present the CPU performance variance in Figure 3.21. As the reader might notice, Google's Compute Engine seems to incur a higher variance in CPU performance. This is confirmed by the COV measurement in

COV	CPU
AWS EC2 (total)	0.05
AWS EC2 (by CPU Type)	0.003
Microsoft Azure	0.0034
Google CE	0.0264

Table 3.10: CPU Performance Variance across Vendors

Table 3.10. Here the GCE CPU performance COV is an order of magnitude higher than for Microsoft's Azure. When comparing these COV values to the ones obtained at AWS we notice that the GCE CPU performance variance is comparable with the variance at AWS across all CPU types. On the other hand the MS Azure CPU performance variance is comparable to the AWS CPU performance variance when considering the different CPU types. Does that imply that GCE is using different CPU types while MS Azure is not? Unfortunately it is not that easy. As far as we can tell from inspecting of the /proc/cpu files both vendors use the same CPU type across all instances we allocated. GCE's CPU type is the Intel Xeon CPU with 2.60GHz and MS Azure's CPU type is the AMD Opteron 4171 with 2.094 GHz. Does this imply Microsoft and Google are more consistent than AWS by giving the user always the same system configuration? Not necessarily, because one has to consider the age of all three vendors. While both Google and Microsoft started their IaaS cloud services in June 2006, AWS is running publicly since 2006. Therefore AWS has gone though a number of upgrades for their existing data centers and also build new data centers equipped with the state of the art systems at the time of construction. Recall that the different providers are using different virtualization technologies and the results are also influenced by the different Hypersivors: AWS uses Xen [25], Microsoft Azure uses the Windows Azure hypervisor (which is based on HyperV [54]) and the Google Compute Engine uses KVM [49].

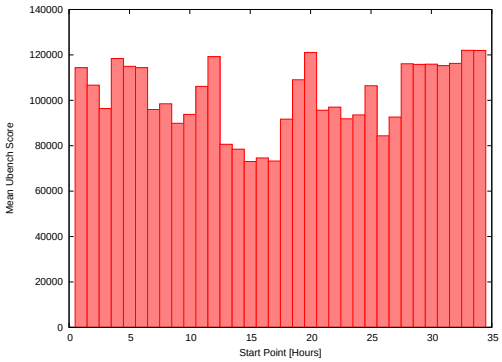


Figure 3.22: 50 Nodes Mean CPU Performance.

3.6 Impact on a Real Application

So far, we ran microbenchmarks in small clusters of virtual nodes. Thus, natural next steps are to analyze (1) whether the performance variability observed in previous sections will average out when considering larger clusters of virtual nodes and (2) to which extent this micro variances influences actual data-intensive applications. As MapReduce applications are frequently performed on the Cloud, we found them to be a perfect candidate for such analysis.

For a random variable such as measurement performance one would expect that the average cluster performance will have less variance due to the larger number of 'samples'. Therefore we experimented with different virtual cluster sizes up to 50 nodes. However, we could not observe a significant relationship among the number of nodes and the variance. Note however, that also for the cluster 20%-30% of the measurements fall into the low performance band. Here we only show results for the largest cluster of 50 nodes we tried. As for previous experiments, we reallocated the cluster every hour. We performed this measurement for 35 hours in a row. For each hour we report the mean CPU performance of the cluster.

Figure 3.22 shows the results. As we may observe from the figure even when running 50 instances concurrently, the mean performance still varies considerably. Thus, a large cluster does not necessarily cancel out the variance in a way that the performance results become stable (see Figure 3.23). This is because performance still depends on the number of Xeon processors that composes a cluster of virtual nodes as discussed above for single virtual instances (Figure 3.10). It might of course be that the variance of the means will be reduced for larger clusters of several hundred nodes. Whether the means are then use-

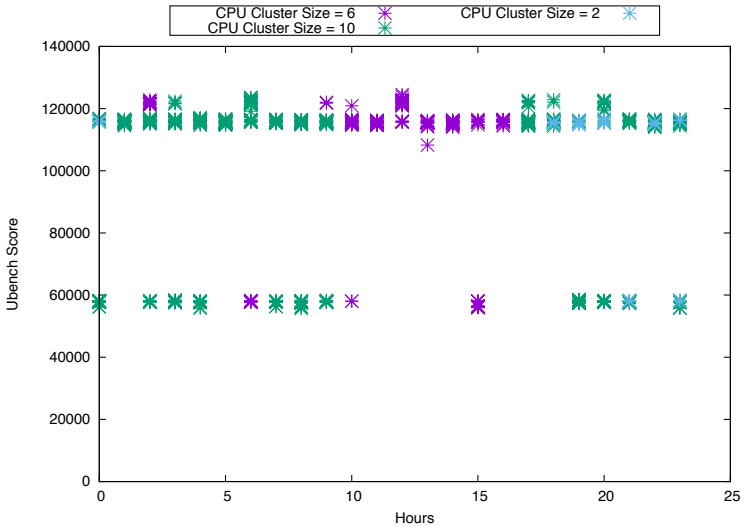


Figure 3.23: CPU Performance for variable Cluster Size.

ful for any practical purposes or realistic applications for smaller systems as Yahoo! or Google is left to future work.

A natural question at this point is: *How does all this variance impact performance of applications?* To answer this question we use a MapReduce job as benchmark for two main reasons. First, MapReduce applications are currently one of the most executed applications on the cloud and hence we believe this benchmark results would be of great interest for the community. Second, a MapReduce job usually makes intensive use of I/O (reading large datasets from disk), network (shuffling the data to reducers), CPU and main memory (parsing and sorting the data).

For this, we consider the following relation (as suggested in [150]), `UserVisits(UV)=(sourceIP, visitedURL, visitDate, adRevenue)`, which is a simplified version of a relation containing information about the revenue generated by user visits to webpages. We use an analytic MapReduce job that computes the total sum of `adRevenue` grouped by `sourceIP` in `UserVisits` and executed it on Hadoop 0.19.2. We consider two variations of this job: one that runs over a dataset of 100GB and other that runs over a dataset of 25GB. We ran 3 trials for each MapReduce job every 2 hours over 50 small virtual

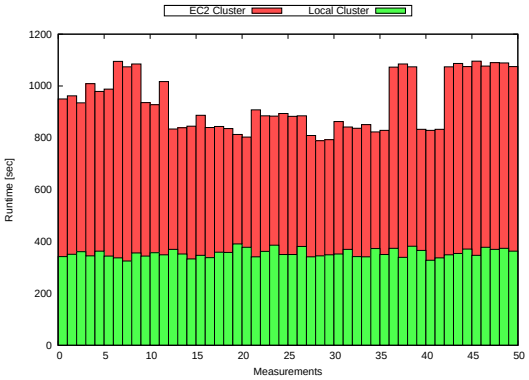


Figure 3.24: Runtime for a large MapReduce job.

nodes (instances). For EC2, we created a new 50-virtual nodes cluster for each series of 3 trials. In our local cluster, we executed it on 50 virtual nodes running on 10 physical nodes using Xen.

As results for the small and large MapReduce jobs follow the same pattern, we only illustrate the results for the large MapReduce job. We showed in fact these results in Figure 3.24 as motivation in the introduction. We observe that MapReduce applications suffer from much more performance variability on EC2 than in our cluster. More interesting, we could again observe during these experiments that both MapReduce jobs perform better on EC2 when using larger percentage of Xeon-based systems than Opteron-based systems. For example, when using more than 80% Xeon-based systems the runtime for the large MapReduce job is 840 seconds on average; when using less than 20% Xeon-based systems the runtime is $1,100$ seconds on average. This amounts to an overall COV of 11%, which might significantly impact experiments repeatability. However, even if we consider a single system type, performance variability is by an order of magnitude higher than on our local cluster.

3.6.1 MapReduce and Cluster Heterogeneity

One of the main benefits of using Hadoop is that users can easily execute their jobs on large clusters without having to worry about the challenges of distributed computing such as scheduling, network communication, or synchronization. The Hadoop framework hides the fact that the computation is executed on several nodes. Unfortunately, the Hadoop framework assumes

that the performance of the cluster nodes is relatively homogeneous. This assumption is for example reflected in the scheduler and the straggler detection components of Hadoop [122, 174]. While this assumption of homogeneous node performance might hold to certain degree in the data centers of Google or Facebook it is certainly not valid for cloud settings as we have seen in Chapter 3. In Section 3.6.2 we examine how the observed variance effects the runtime of MapReduce. Note that here we are not concerned with the question of the runtime variation of MapReduce jobs but rather with the question whether cluster heterogeneity can cause a general slowdown of MapReduce jobs. In addition, we also examine the effects of cluster variance on HDFS performance in Section 3.6.6. For this purpose we consider several aspects: On the one hand we consider effects on the HDFS upload performance. On the other hand we also consider how heterogeneity in node performance affects the block placement which basically defines the data locality for the later running Map Reduce jobs.

Next, we examine how the observed performance degradation is effected by the performance variance in several subcomponents, i.e., we try to analyze how much of the slow down is due to heterogeneity in I/O performance, CPU performance and network performance respectively. After quantifying the effects of cluster heterogeneity on MapReduce performance, we are interested in how one can reduce the cluster heterogeneity in a cloud computing environment where we cannot control the co-placement of virtual instances on physical hardware in Section 4.1. Our solution aims at directly allocating a more homogeneous cluster instead of modifying the Hadoop framework and is hence usable for other applications. Our solution is based on Whirr [13] which allows for a cloud neutral way to run Hadoop. In order to provide a more homogeneous cluster we allocate a slightly higher number of nodes than actually required and then select the subset of nodes having the least performance variance among them. We show that especially for long running MapReduce jobs the initial overhead of allocating more nodes and benchmarking the performance is actually negligible compared to the improved query runtimes.

3.6.2 Variance Problems for MapReduce

Let us first examine how the performance heterogeneity of a seemingly homogeneous cluster (consisting of nodes having the same instance type) effects the runtime of MapReduce jobs in a cloud setting. To answer this question we allocate several clusters of 25 small instances each at Amazon EC2. We then

use the same micro-benchmarks as in Chapter 3 (unixbench [77] for measuring CPU performance and bonnie++ [23] for measuring I/O performance) to measure the performance of each node. Next, we measure the response times of MapReduce jobs implementing TPC-Query 6 [82] over a 100GB dataset on each of cluster.

We show the resulting runtimes averaged over three runs in Figure 3.25. As one can see the average runtimes are varying from 2,996 seconds up to 3,403 seconds.

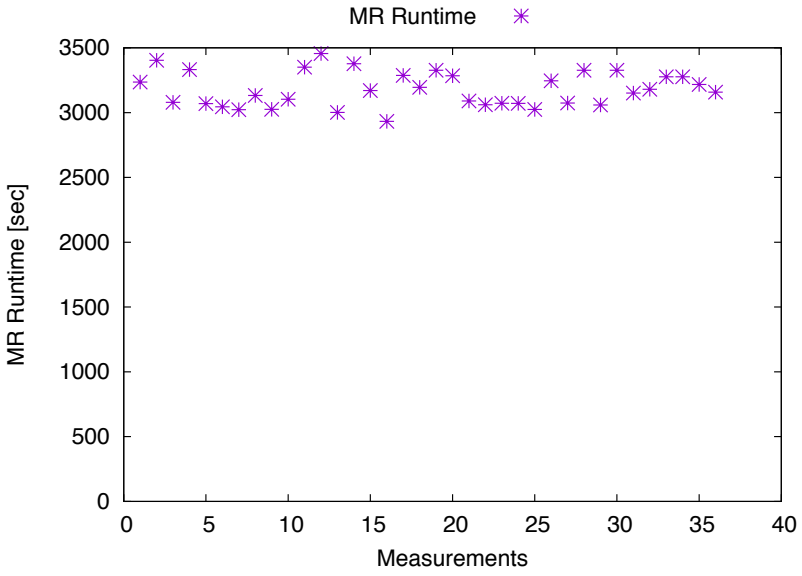


Figure 3.25: MR Job Runtime.

As we would like to examine the relation of cluster heterogeneity to MapReduce performance we first have to establish a measure $HET_{cluster}$ for cluster heterogeneity. Note that, while so far we considered the performance variance of individual nodes, $HET_{cluster}$ measures the performance variance between different nodes in a cluster. In Chapter 3 we already used the *Coefficient of Variation* (COV), which is defined as the ratio of the standard deviation to the mean to quantify the heterogeneity of a single performance characteristic (e.g., CPU performance). In order to extend this to the notion of cluster heterogeneity we use a linear combination of the individual performance characteristics:

$$HET_{cluster} = a_1 * COV_{CPU} + a_2 * COV_{IO-Read} + a_3 * COV_{IO-Write} + a_4 * COV_{Network}$$

where COV_{CPU} is the COV of the CPU performance, $COV_{IO-Read}$ the COV of the IO Read performance and $COV_{IO-Write}$ the COV of IO Write performance respectively. The definition of $COV_{Network}$ is a little more challenging in a cluster setting: As in Hadoop each worker potentially has to share intermediate results with all workers we would have to consider the pointwise network performance between all nodes in the cluster. As this is quite expensive (for example a 25 node cluster this would require $\frac{(n-1)*n}{2}$ measurements of network performance. Instead we measure the network performance between the name node and all other nodes (this approach only requires 24 measurements of network performance). Note that we did not include the heterogeneity in memory performance into $HET_{cluster}$. This is because our measurements from Chapter 3 indicate that memory performance is relatively stable across instances. First we consider the influence of all performance characteristics as equal, meaning that $a_1 = a_2 = a_3 = a_4 = 1$. We return to this question in Section 3.6.4 and examine the influence of different application characteristics (e.g., IO bound application vs CPU bound application) on these weights.

Also we want to differentiate between the influence of the cluster heterogeneity $HET_{cluster}$ as defined above and the cluster performance. Therefore we have to define a measure for the average cluster performance $PERF_{cluster}$. We define $PERF_{cluster}$ as the sum of the average performance indicators (CPU, IO-Read, IO-Write, and Network) over all cluster nodes. We normalize each of the average values in order to give each indicator an equal influence despite its relative magnitude. As normalization constant $GAVG(indicator)$ we use the average for the given performance indicator across all clusters i.e., across all measurements. Hence

$$PERF_{cluster} = \frac{AVG_{CPU}}{GAVG(CPU)} + \frac{AVG_{IO-Read}}{GAVG(IO-Read)} + \frac{AVG_{IO-Write}}{GAVG(IO-Write)} + \frac{AVG_{Network}}{GAVG(Network)}$$

Figure 3.26 and Figure 3.27 provide an overview of the distributions for $HET_{cluster}$ and $PERF_{cluster}$ for each cluster. Here we see that both measures vary significantly between different clusters. So the question remains which influence these measures have on the MapReduce runtime.

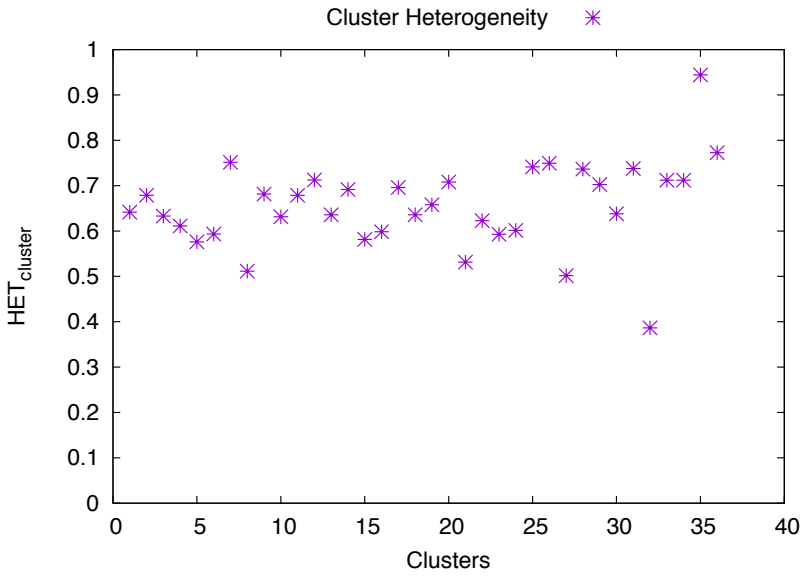


Figure 3.26: Distribution of $HET_{cluster}$.

Next, we want to quantify the correlation between $PERF_{cluster}$ and average MapReduce runtime, between $HET_{cluster}$ and average MapReduce runtime, and between $PERF_{cluster}$ and $HET_{cluster}$. As a measure of correlation we use pearson's coefficient of correlation. Pearson's coefficient of correlation r is defined to be the covariance of two variables normalized by the product of their standard deviations, see also Equation 3.1. Note that pearson's coefficient of correlation is always between -1 and 1, where -1 indicates a perfect negative correlation, 0 indicates no (linear) correlation at all, and 1 indicates a perfect positive correlation.

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{3.1}$$

Figure 3.28 shows the distribution of average MapReduce runtime across the different $PERF_{cluster}$ measurements. As one would expect the correlation between $PERF_{cluster}$ and average MapReduce job runtime is with $r = -4.007036503 * 10^{-1}$ significantly negative: hence a better cluster per-

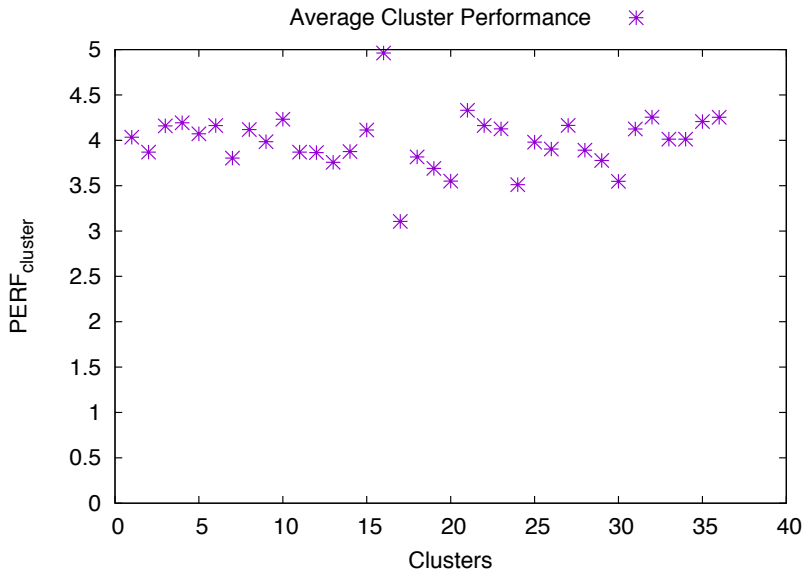


Figure 3.27: Distribution of $PERF_{cluster}$.

formance indicates less runtime. The correlation also validates our choice for $PERF_{cluster}$ to be a good indicator for the overall cluster performance.

Next we consider the correlation between $HET_{cluster}$ and average MapReduce job runtime as an indicator for how the heterogeneity affects application performance. We measure a coefficient of correlation of $r = 2.096194253 * 10^{-1}$. This positive correlation confirms our assumption that cluster heterogeneity results in longer MapReduce job runtimes. We examine this phenomena in more detail in more detail throughout this chapter.

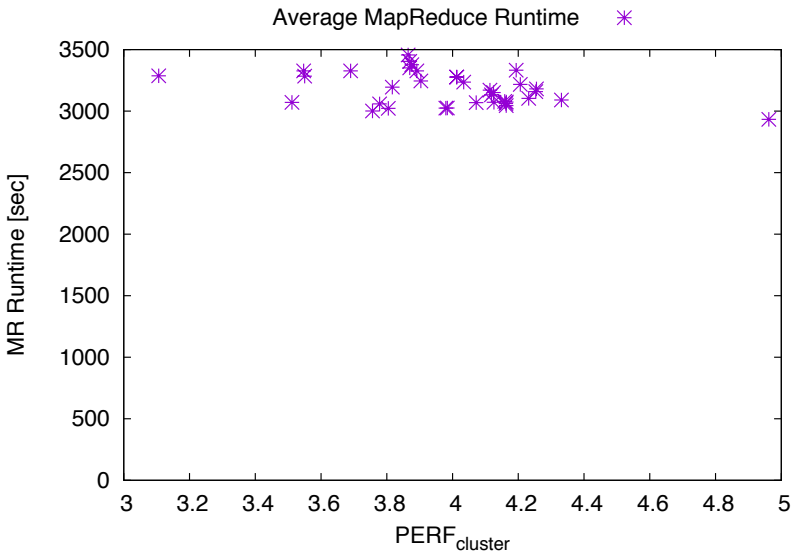


Figure 3.28: Average Runtime versus $PERF_{cluster}$.

Interestingly, if we look at the correlation between $PERF_{cluster}$ and $HET_{cluster}$ in Figure 3.29, we obtain a significant negative correlation with $r = -4.084864569 * 10^{-1}$. This means that a high cluster heterogeneity is an indicator for low cluster performance. A first intuitive explanation could be that this correlation might be caused to different system types as discussed in Section 3.3. But when examining the system distribution in these clusters we found that most clusters had a pretty equal distribution of system with around 95 percent of the nodes running with the E5-2650 2.00GHz processor. The reason for the significant negative correlation is more likely the influence of co-located instances on the same physical node. The more instances are sharing the same physical hardware (this can even be a network switch and is hence not constrained to a single physical node) the less performance each single virtual instance receives. A cluster containing such underperforming nodes also incurs then a higher heterogeneity as the performance varies more between different instances.

As we are considering the correlation between $PERF_{cluster}$, $HET_{cluster}$, and MapReduce runtimes, it is an interesting questions whether $PERF_{cluster}$ and

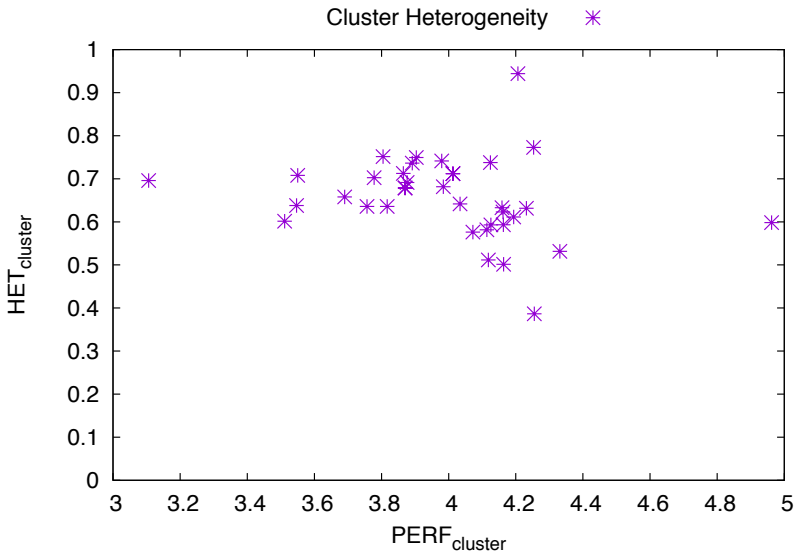


Figure 3.29: Distribution $PERF_{cluster}$ over $HET_{cluster}$.

$HET_{cluster}$ are stable measures over time. We have already seen in Section 3.5.9 that the individual performance measure are relatively stable over time. Only I/O performs might suffer a little bit due to contention from co-located instances, but is still an order of magnitude less then between instance I/O performance variance. But what about $HET_{cluster}$? Is the heterogeneity also stable over time?
To answer this question we reran the measurements and recomputed $HET_{cluster}$ for the same cluster. In order to simulate a very long running job, we started one measurement run per hour for a total of 10 runs. As we can see from Figure 3.30 $HET_{cluster}$ is quite stable over time for a given cluster. Note that for comparison we also show measurements for $HET_{cluster}$ for 10 different clusters. There is one notable outlier with a much lower $HET_{cluster}$. This was actually caused by an overall drop in I/O read performance, which was then equally low on all nodes. Hence the assumption that an initial performance measurement is also a good predictor for later points in time is justified by these numbers.

So far we have looked at the high level effects of cluster heterogeneity onto MapReduce performance. Here we have seen a significant positive correlation

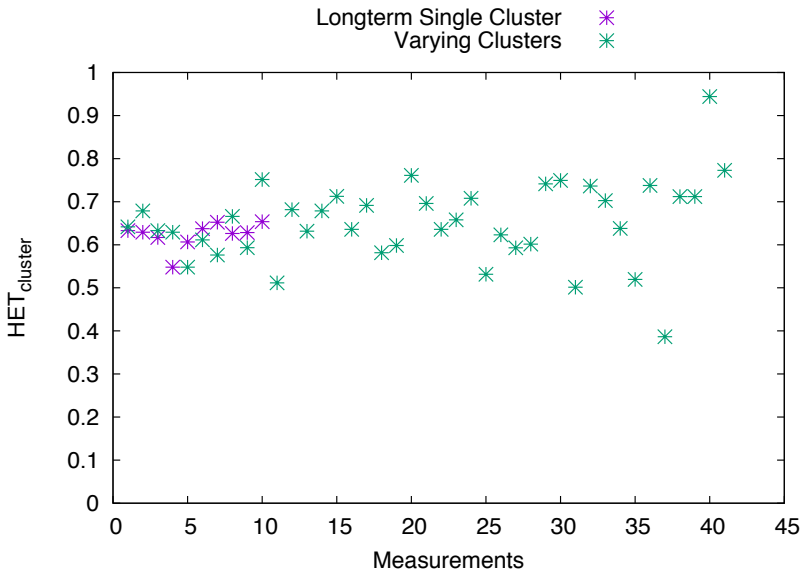


Figure 3.30: Stability of $HET_{cluster}$.

between $HET_{cluster}$ and MapReduce runtimes. In the following sections we will examine this effect in more detail.

3.6.3 CPU Steal Time

Let us consider another interesting metric for instances running in virtualized environments: *cpu steal*. According to the *mpstat* manpage, cpu steal is the percentage of time spent in involuntary wait by the virtual cpu(s) while the hypervisor was servicing another virtual processor. So basically how much of the cpu wait time was caused by the hypervisor choosing not to schedule the respective virtual cpu(s). Note that on a physical system running without hypervisor this metric will always be 0.

But what are the causes for a high cpu steal percentage? The first one is due to contention with other virtual instances running on the same hypervisor (i.e., usually same physical hardware). In such cases the sum of requested cpu capacity across all instances is higher than the actual cpu capacity of the physical hardware. This causes involuntary wait states for the virtual cpu(s) and hence a high cpu steal percentage. Still a high cpu steal percentage is not always due to contention by other virtual instances. In many virtualized settings there is a maximum limit for the cpu usage for each instance, for example the instance is assigned 30% of the overall cpu capacity for that system. Then if the instance tries to use more cpu cycles the hypervisor might force the virtual cpu to wait despite whether there might be actual free capacity or not. So in this case the high cpu steal percentage is caused by the instance trying to use more cpu capacity than its assigned share.

Netflix [59], a large online video provider uses the cpu steal metric in order to determine virtual instances which perform poorly due to contention with other collocated virtual instances. Netflix actually kills such virtual instances and recreates them on a hopefully different physical machine.

So the question arises whether the cpu steal percentage might also be an indicator for variance caused by contention with other collocated virtual instances. To examine this question we first consider the cpu steal percentage across different instance types. In particular we consider the m1.micro, m1.small, m1.large, and m1.xlarge as they are the most commonly used instance types and represent a wide range of performance and isolation guarantees. Recall that smaller instance types have less performance and a higher probability of other collocated instances on the same physical machine. This holds especially for the micro instance types, where AWS specially states that cpu cores are likely to shared among instances to provide additional performance bursts [19].

We used the unix `top` command to measure the user based user based cpu

utilization and the cpu steal percentage. For this we allocated one instance per instance type and measured both values 1000 times in each instance. The resulting resolution is about one measurement every two seconds.

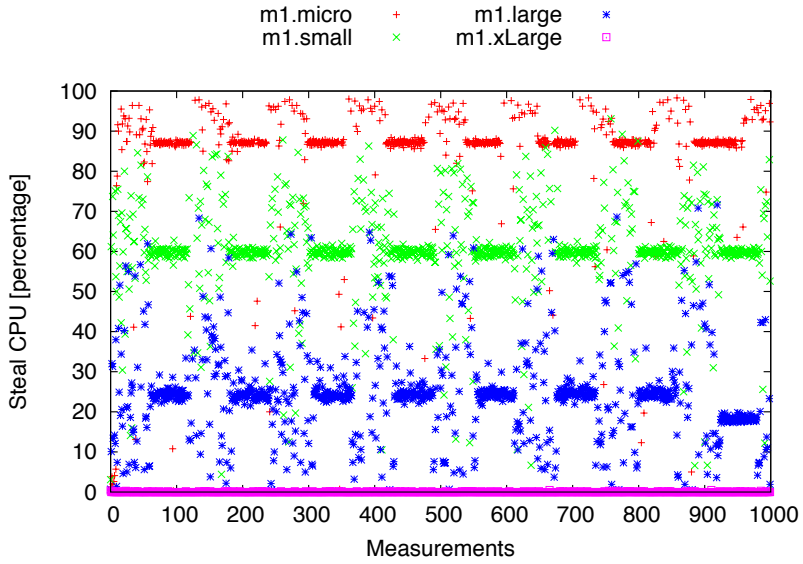
In Figure 3.32 we show these measurements on otherwise empty instances, not running any additional workload. As we would expect both the user cpu percentage in Figure 3.32(b) and steal time in Figure 3.32(a) are close two zero. This is because there no potential for involuntary waiting for cpu capacity (and therefore cpu steal) if there is no demand for cpu capacity.

More interestingly are the results in Figure 3.32. Here we created demand for cpu capacity by running a cpu intensive benchmark on each instance. We choose uBench [76] for this purpose, mainly because uBench has two distinct phases. First uBench generates a cpu intensive workload with one thread per core. This ensures that we fully utilize several virtual cpu cores available on the large instance types. After this first phase uBench generates a memory I/o intensive workload in order to measure the memory performance. This phase requires also a fair amount of cpu capacity but less compared to first phase. These two distinct phases of cpu utilization allow us to differentiate between the two causes for cpu steal. During the cpu intensive phase it might be that the steal time is simply caused by demanding more cpu capacity than assigned to the virtual instance. During the memory I/O intensive phase the instance demands some cpu capacity but due to the I/O bound characteristic of the workload this will usually be less than the maximum cpu capacity assigned to the virtual instance. Hence we can observe contention related cpu steal times in this case.

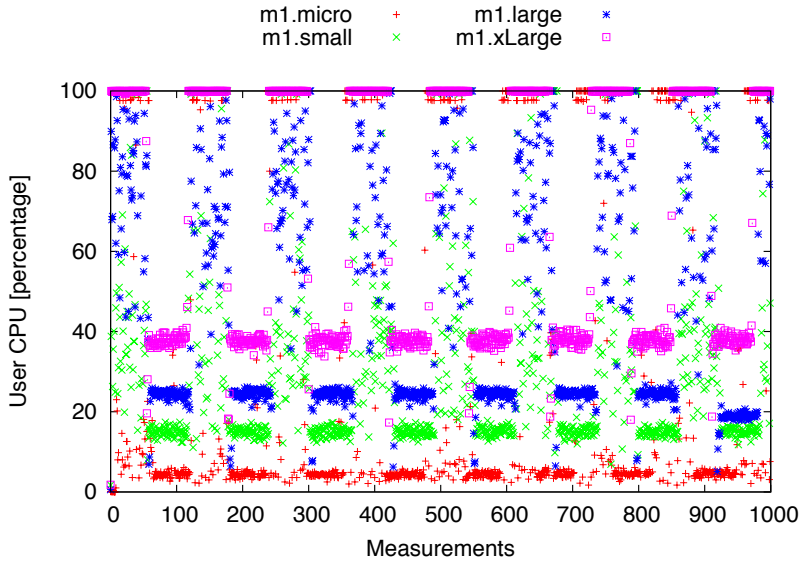
These two phases of uBench can nicely be observed in Figure 3.31(b) which shows the user cpu utilization. We can see that there are phases of mostly 100% cpu utilization (e.g., the first 80 measurements) followed by phases with lower cpu utilization (e.g., measurements 80 to 120). Interestingly the spread of potential percentage values is quite high during the cpu intensive phases and quite stable during the memory intensive phases. Moreover during the cpu intensive phase the smaller the instance type the more and lower percentage values we measure. This is an indicator that the cpu has maximum assigned cpu capacity. Hence once the instance or virtual cpu has exceeded it assigned share, the cpu will be idle if not required by another virtual instance. As for the m1.xLarge instance type only one virtual instance is scheduled per physical machine, the maximum assigned cpu capacity is equal to the maximum physical cpu capacity. Hence there is no need to have a virtual cpu idle because it exceeded its maximum share on the m1.xLarge instance type. In the second

memory intensive uBench phase we usually require less cpu capacity than the assigned maximum capacity. Therefore the cpu utilization is stable across all instance types as no virtual cpu exceeds it maximum assigned share.

In Figure 3.31(a) we show the corresponding cpu steal percentage measurements. First we observe that the smaller the instance type the higher the cpu steal percentage. This is an indicator that there is either a smaller assigned maximum share or a higher contention for smaller instance types. For the m1.xLarge the cpu steal percentage is almost always 0 as a m1.xLarge virtual instance corresponds to a physical machine, Again, we observe the two distinct phase. During the cpu intensive phase (i.e., measurement 0-80) we observe a high variance between measurements per instance types. The reason for this that the virtual cpu attempt to allocate as much CPU capacity as possible. During the memory I/O intensive phase (i.e., measurement 80-120) the cpu steal percentages per instance type are more stable. This is caused by the lower overall cpu consumption.

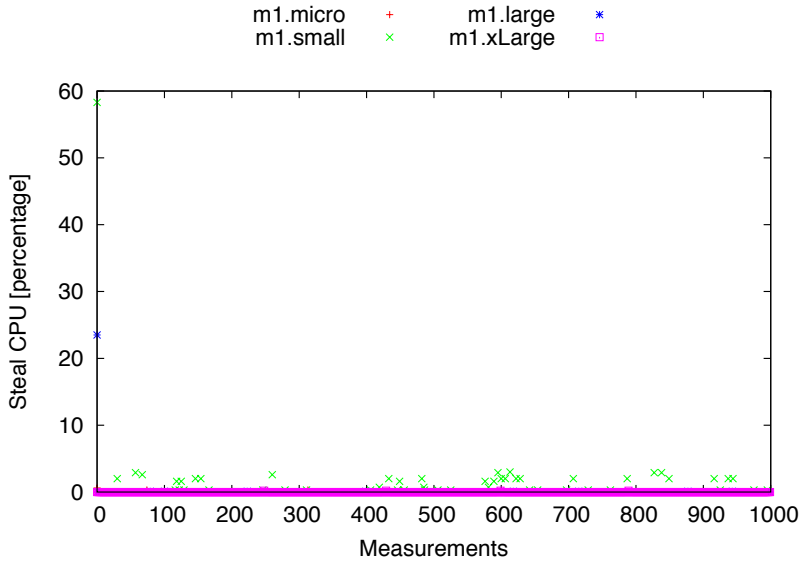


(a) Steal CPU Percentage.

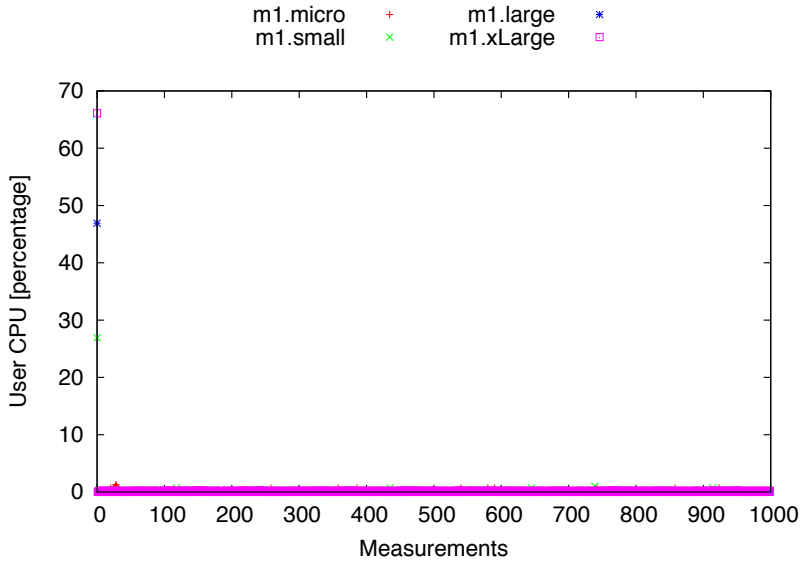


(b) User CPU Percentage.

Figure 3.31: `top` measurements while running uBench



(a) Steal CPU Percentage.



(b) User CPU Percentage.

Figure 3.32: `top` measurements on otherwise empty instance

So the question is whether a high steal CPU percentage might be an indicator for performance problem. To answer this question we collected steal CPU percentages in addition to our other benchmarks. As before we distinguish three different phases for collecting the CPU steal percentage: while CPU benchmarking where the CPU load is very high, while memory benchmarking where CPU load is medium, and while no additional load is on the system. As before we used uBench [76] for both the CPU benchmark and memory benchmark. During each of these three phases we take a total of 50 measurements on each node and then consider the average. Note that while no additional load was on the system the CPU steal percentage was as expected always below one percent. Therefore we do not show detailed graphs for this phase.

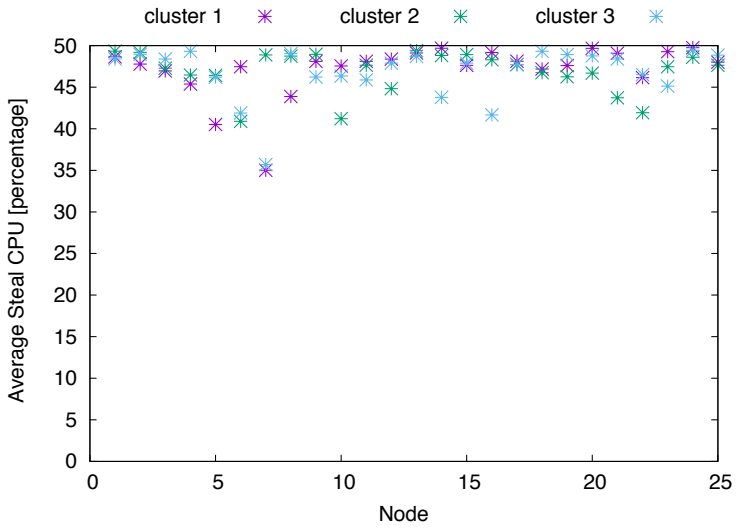
In Figure 3.33 we show the steal percent numbers for the individual nodes of three sample clusters Here we see that the average performance while running the uBench CPU benchmark is between 50 and 60 percent which is similar to our initial measurements in Figure 3.31. The difference between individual nodes is not very high, so there are no nodes experiencing massive contention with other nodes. More surprisingly the CPU steal percentage in Figure 3.33(b) while running the uBench memory benchmark is slightly higher.

Next we want to take a look how the cpu steal percentage correlates with the measured CPU performance. Therefore we plotted the same data (i.e., averaged cpu steal measurements for each node of the three sample cluster) against the measured CPU performance in Figure 3.34. Unfortunately there is no significant correlation between the two values for both the CPU benchmark phase in Figure 3.34(a) and also the memory benchmark phase in Figure 3.34(b). These results indicate that a higher cpu steal percentage does not result in an overall less CPU performance.

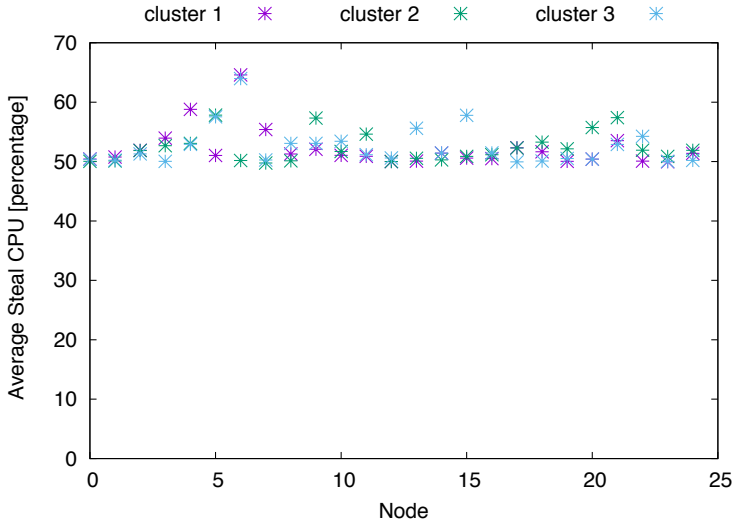
Last, we take a quick look on whether between clusters there are difference in the average cpu steal percentage (i.e., taking the average of all nodes in a given cluster). This could result if a certain cluster is allocated on a highly utilized rack of physical machines. But the measurements in Figure 3.35 indicate that this is not the case, as the cluster-level average values for both phases are very close to each other.

One problem with measuring CPU steal percentage is the problem to distinguish between the two causes: exceeding the CPU share and actual contention with co-located virtual instances. We would recommend to monitor steal performance and drop instances from a cluster which experience a very high cpu steal percentage compared to comparable instances. Comparable instances

refers here to other instances of the same instance type running a comparable workload. Those instances are likely to to experience cpu contention with other co-located instances. As also cpu steal percentage would cause primarily variance in a single node performance (which we saw to be relatively stable in Section 3.5.9). Therefore CPU steal percentage is not useful for measuring the heterogeneity of an cluster. Hence we will continue using $Het_{cluster}$ for measuring heterogeneity.

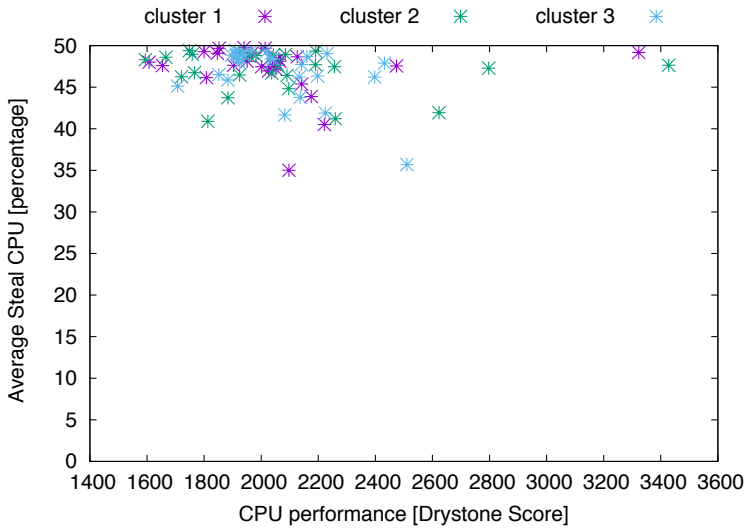


(a) During Memory Benchmark.

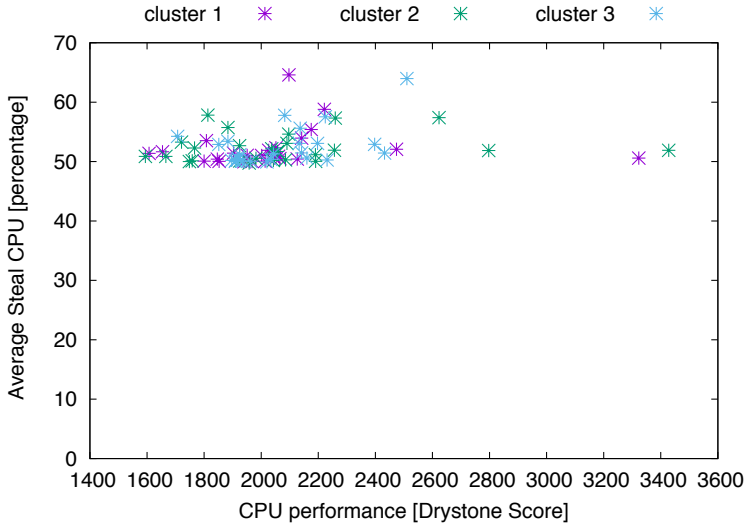


(b) During CPU Benchmark.

Figure 3.33: Average Steal CPU Percentage Measurements for individual Nodes.



(a) During Memory Benchmark.



(b) During CPU Benchmark.

Figure 3.34: Average Steal CPU Percentage Measurements for individual Nodes by CPU Performance.

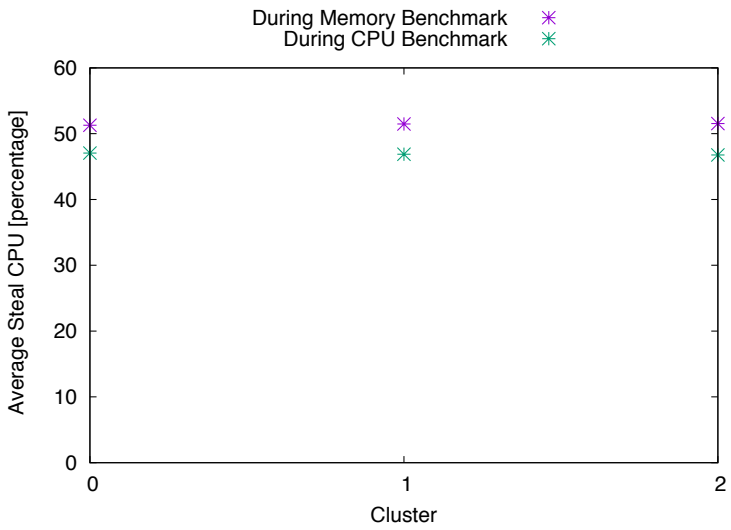


Figure 3.35: Average Steal CPU Percentage Measurements for Cluster.

Coefficient of Correlation	Performance	COV
CPU	-0.6492	0.0735
Network	-0.461	0.4245
IO-Read	0.1393	0.6500
IO-Write	-0.52815	0.2640

Table 3.11: Correlation of individual components and average MapReduce job runtime.

3.6.4 Application Characteristic

Next, let us revisit the assumption that each single characteristic in $HET_{cluster}$ and $PERF_{cluster}$ has an equal effect on the MR runtime.

We therefore consider the correlation between the different performance components (such as CPU performance) and the average MapReduce runtime. We again do this considering two dimensions: performance (i.e., the individual components of $PERF_{cluster}$) and heterogeneity (i.e., the individual components of $HET_{cluster}$). As before we normalize the performance measurements in order to obtain comparable values for the coefficient of correlation. Again we use the global average, meaning the average of all measurements across all clusters as normalization constant. Table 3.11 shows pearson's coefficient of correlation for each of these individual measures with the average MapReduce job runtime. Recall that a negative coefficient of correlation between a performance measure and average job runtime means that higher performance indicates less job runtime (which is what we would expect). On the other hand a positive correlation between a COV measurement and average job runtime means that a higher COV measurement indicates also longer running jobs.

Let us first look at the correlation of the individual performance components with the average job runtime. We see that CPU, Network, and IO-Write have a significant negative correlation. The correlation between CPU performance and runtime seems to be the most significant, quickly followed by IO-Write performance and then Network performance. This is in line with our expectations and the measured correlation between $PERF_{cluster}$ and average job runtime. What is more surprising is the less significant and positive correlation between IO-Read performance and average job runtime. This would mean that a higher IO-Read performance actually corresponds to slower job runtimes. But looking at the significance of this correlation we quickly see that it is way less significant compared to the other performance components. The most likely explanation

for this strange behavior (especially when compared to the significant negative correlation coefficient for the IO-Write performance) can be found in the Amazon storage structure: As the default IO storage device is EBS mapped storage, meaning it is attached via network. Now when all nodes start start reading large amounts data at the beginning of the MapReduce job these attached storage quickly becomes a bottleneck. The writing of the result to disk is stretched over a longer timeframe and also the output of a MapReduce job is usually smaller than its input (roughly factor of 10 in our benchmark job). Next, we consider the correlation of the individual heterogeneity components (i.e., COV for a given component) with the average job runtime. We notice that all coefficients are positive and the variance in IO-Read and network performance have the most significant positive correlation with the average job runtime. This means that the given MapReduce job is especially sensitive to variance in these two performance characteristics. On the other hand variance in CPU performance has no significant correlation with the job runtime. Note, that this behavior might be different for another MapReduce job. Another MapReduce job performing K-means clustering [112] which has a CPU intensive Map function will be more sensitive to variance in CPU performance.

3.6.5 Cluster Heterogeneity and Runtime Variance

Another interesting question is how the the cluster heterogeneity $HET_{cluster}$ effects the variance of our MapReduce job runtimes. This is not directly related to our initial question of how the heterogeneity effects the overall runtime. Still, when considering predictable and robust query performance the runtime variance is a valuable measure for cloud users. For quantifying the runtime variance we use two different measurements: First, we use again compute the COV. But as we only have three runtime measurements for each cluster the standard deviation in the COV computation is limited in its statistical value. Therefore, we also consider the absolute difference between the minimal and maximal runtime $Diff$ as a measure for runtime variance. Note, that $Diff$ is also a relevant measure for the users as it specifies the time range in which they can expect the query to finish. We again use pearson's coefficient of correlation as defined in Section 3.6.2 to quantify the correlation between both values. Note that we use the same set of measurements as in Section 3.6.2. When considering the correlation between $HET_{cluster}$ (i.e., a combination of the $COVs$ for the different performance measurements for the specific cluster) and the COV of the three MapReduce runtimes measurements, we see a rather small correlation with $r = 0.132$. This means that a high cluster heterogeneity

does not directly coincides with a high runtime variance measured by with the COV. We assume this is partly due to the small number of data points used to compute the standard deviation.

On the other hand when considering the $Diff$ (i.e., the longest job runtime for a given cluster minus the fastest job runtime) we obtain a significant positive correlation with $r = 0.423$. This means that for more heterogeneous clusters also the absolute deviation in runtimes is larger. Therefore the user has to deal with less predictable and robust performance.

3.6.6 Variance Problems for HDFS

Next, we consider the effects of performance variance on HDFS performance. Herefore, we also measure the HDFS upload time for a 10GB dataset which was previously partitioned on the different nodes in each cluster (i.e., parallel upload).

As we can see in Figure 3.36 there is also variance the HDFS upload times.

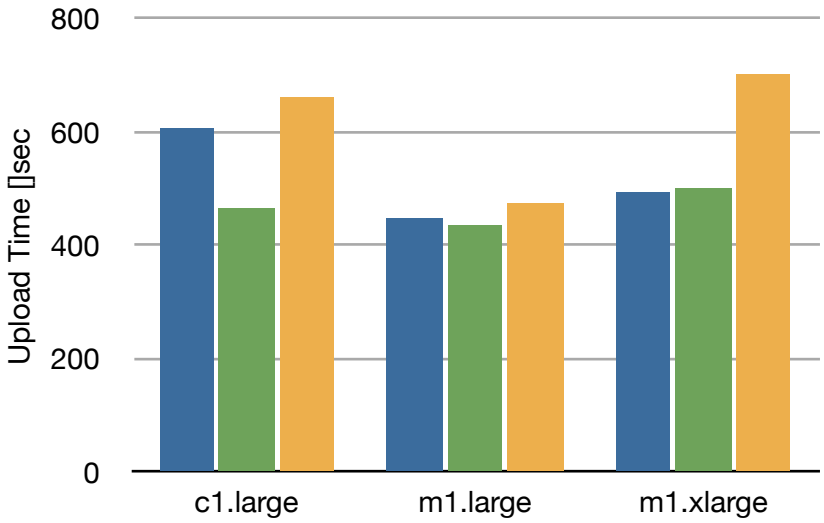


Figure 3.36: HDFS Upload Variance for 100GB Data on 10 Node cluster.

In order to explain the variance further we take a detailed look at a single HDFS upload on a 100 node cluster. Recall that we upload data in parallel from each

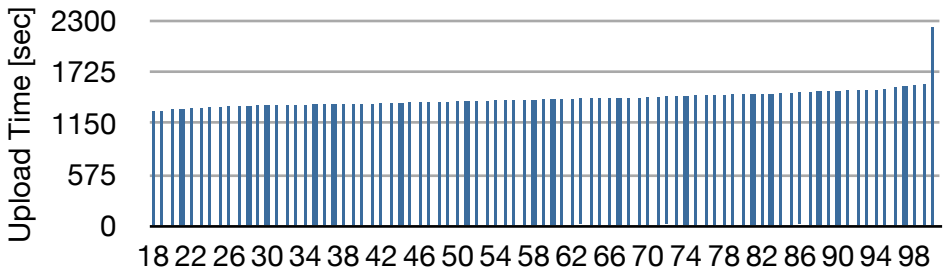


Figure 3.37: HDFS per Node Upload Time on 100 Node cluster .

node in the cluster. Figure 3.37 shows the local upload times per node. As one can see, the overall runtime is dominated by a few outlier nodes which basically limit the overall runtime. The surprising fact is that nodes finishing last are actually the nodes having the best I/O performance. This behavior is a result of the HDFS block placement strategy: When the namenode realizes that a given node takes longer to store blocks, it will eventually stop placing HDFS data blocks on these slow nodes. Hence, the fast nodes have to store more remote HDFS data blocks (i.e., HDFS data blocks from other nodes) and hence take longer to finish their local uploads.

Another important factor for HDFS read performance and therefore MapReduce job runtime is the placement of blocks. HDFS tries to locate the blocks as equally between nodes as possible in a homogeneous cluster, but this is not anymore true for heterogeneous clusters.

3.7 Performance Variance for Software as a Service

So far we focused in IaaS Cloud service offer (i.e., we allocate virtual machines and execute benchmarks on top of these virtual machines). As Cloud vendors also offer Software as a Service (*SaaS*) such as AWS Elastic MapReduce [18], the question arises whether these services suffer from the same performance variance. In theory an SaaS provider has more potential to reduce the variance in SaaS offers as compared to IaaS offers. First the vendor has more control over the environment as he is responsible for most of the application stack including operating system and settings and application settings. Secondly, the workload can be more easily estimated for a given self-controlled software than for the more general IaaS setting where the user might issue any kind of workload. In order to answer this question we execute the same MapReduce job *UV* used in Section 3.3 using AWS Elastic MapReduce. The UV MapReduce job uses the following relation UserVisits *(sourceIP, visitedURL, visitDate, adRevenue)* as input and computes the total sum of adRevenue grouped by sourceIP in UserVisits. On AWS EMR we used 5 small instances in the US East region and 25 GB of UV data uploaded to S3 buckets located in US East. We used the EMR provided Hadoop version 1.03 and otherwise the default settings of EMR. The setup was launched every three hours for a total of 80 runs. Recall that the Job in Section 3.3 was executed over 100GB UV on a 50-virtual cluster with small instances and a custom deployment of Hadoop 0.19.

The end-to-end job runtimes are shown in Figure 3.38. Note that these end-to-end job times include setup and shutdown of the EMR cluster. The variance with COV of 12.5% for the end-to-end job runtimes is comparable to the COV of 10% for the MapReduce job runtimes in Section 3.3. As the reader might have noticed the comparison is not quite fair as we are comparing end-to-end runtimes to MapReduce job runtimes. To make the comparison we decompose the end-time-end job runtimes into the MapReduce job runtimes in Figure 3.39 and the overhead time including cluster setup and shutdown in Figure 3.40. With this decomposition we can see that the MapReduce job runtime incurs a smaller COV of 12.0% compared to the overhead COV of 17.7%. This difference in variance is due to the availability of nodes: As for EMR cluster EC2 tries to allocate the nodes close to each other (similar as if a user request a cluster of nodes) the time until such cluster can be allocated varies. The difference when comparing the MapReduce only COVs (10% for the IaaS self-allocated cluster and 12% for the SaaS cluster) can be partially explained by data loading. In case of the self-allocated cluster the data is stored in the clusters HDFS, while for the SaaS cluster it has to be loaded from S3. The difference in how the data

is loaded is a result of the SaaS nature of EMR. Overall we can see how the architecture of such SaaS influences the response time variance. EMR creates a new virtual cluster for each job. Another option would be to keep a set of virtual or physical nodes and use those nodes to setup the requested Hadoop cluster. An even more extreme option would be to use a large multi tenant MapReduce cluster for all jobs. Besides the privacy concerns of these solutions, the variance profile would look different for each solution. For example the setup time would be reduced in both solutions.

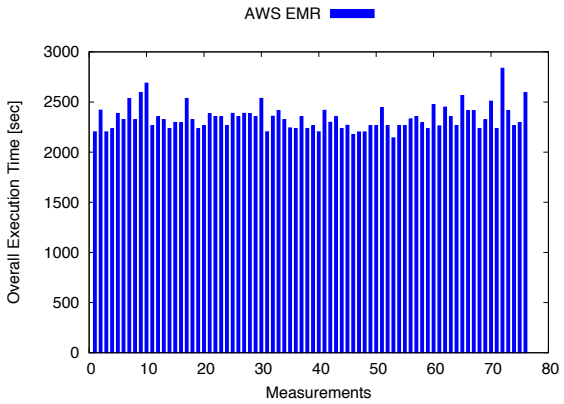


Figure 3.38: EMR overall job runtime including setup

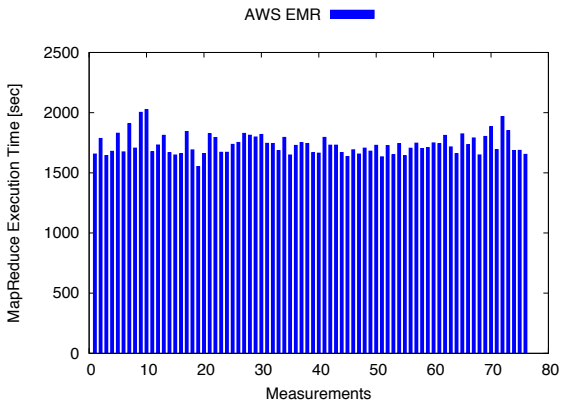


Figure 3.39: EMR MapReduce only runtime

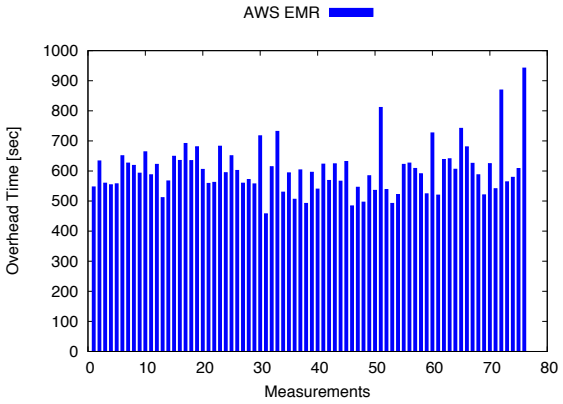


Figure 3.40: EMR Overhead (Cluster Setup)

3.8 Conclusion

Performance variance is a major issue when using cloud infrastructure or applications in the cloud. The promise of cloud computing that users do not have to know about the specifics such as place or system where their application is executed unfortunately also implies a loss of control about performance relevant criteria. We saw in Section 3.3 that when allocating instances having the same specification users can still experience a large performance variance between these "equal" instances. This variance is visible for almost all relevant performance characteristics including CPU, I/O, and network. The only performance characteristics which is relatively stable—also due to the fact that it can be easily virtualized and shared among users—is the memory performance. Also related metaservices such as provisioning or shutdown of instances suffer from performance variance. One can understand and lower the performance variance by not accepting the cloud as a black box but considering the actual—in contrast to the specified—system. This is due to the fact that most cloud providers use internally a mix of different systems which have different performance characteristics.

These differences not only affect low level performance characteristics only captured by performance benchmark tools but also effects real application performance. In Section3.6 we saw that even large cluster systems and applications running on such large clusters are affected by such performance variance.

As for such applications there also exist direct SaaS/PaaS offerings the next step for us was to examine the performance variance of AWS Elastic MapReduce as an example for such offering. In Section 3.7 we show that the performance variance with a COV of 12.5% is comparable to the performance variance in the previous case where we deploy the same application on an IaaS cluster.

We only considered variance between single instances, so the question we consider in Section 3.5.9 is how the performance varies for a single long running instance over time. Here we see that even in such setting performance variance one experiences performance variance with a COV up to 37.5%.

Next in Section 3.5.10, we look at the question whether performance variance is just a problem of Amazon AWS. Therefore we ran a subset of the same experiments on Google's Cloud Service and Microsoft's Azure Cloud. It turns out that performance variance is a problem present across all providers. The

only advantage of new and smaller cloud providers might be that they have less heterogeneity in their systems and data center.

As our initial experiments date back to 2010 we asked ourselves in 2013 whether the situation has changed. In Section 3.5.8 we show that performance variance is still a problem. The most noticeable difference is that today there exits even more different underlying system configuration each having different performance characteristics. So the approach to understanding and lowering performance variance by inspecting the underlying system has become even more important. This holds especially for the placement of instances and applications being sensitive to network I/O performance variance as the number of data centers has grown.

To summarize we would give the following guidelines for cloud users working with performance critical system:

- Be aware of performance variance as it is a consequence of using cloud systems.
- Be aware that the low level system variance affects applications as well.
- In order to reduce the performance variance consider as much information as possible about the underlying systems.
- Keep your systems and applications as close together (at best in the same datacenter) as possible considering your availability constraints.

4 Reducing Cluster Heterogeneity

4.1 Introduction

As we have seen performance heterogeneity between different nodes can prolong the runtimes of MapReduce jobs. So how can we obtain a better, more homogeneous cluster?

One way is to allocate more nodes than actually required, and then remove nodes one by one. Especially for long running jobs in the order of several hours the overhead cost of allocating a few additional nodes is quickly amortized. For the removal process we explore three different approaches: the variance driven approach, the system driven approach, and the performance driven approach. Using the variance driven approach we first remove the node contributing most heterogeneity to the cluster. The system driven approach on the other hand uses the cpu type information and tries to achieve a homogeneous cluster in respect to these CPU types (and the other physical system parameters tied to the CPU). The performance driven approach tries not to optimize the heterogeneity directly, but rather always removes the node with least performance first.

4.1.1 Variance driven Cluster Optimization

For the variance driven approach we execute our established set of micro benchmarks and then terminate the instances contributing most heterogeneity. Hence, after measuring the individual COV and MapReduce job runtime for the entire cluster, we terminate the instance having the largest impact onto $HET_{cluster}$, i.e.,: $argmin_{instance}(HET_{cluster/instance})$. By repeating this process, we obtain a more homogeneous cluster with each step.

To investigate this approach we again used the clusters of 25 instances and ran the above described micro benchmarks. After determining $argmin_{instance}(HET_{cluster/instance)}$ we terminated the identified instance.

Next, we measured the MapReduce job runtime on this reduced cluster. Note, that prior to this measurement we rebalance the data in HDFS. This means that all data blocks will have again the same replication factor and each data node will have roughly the same number of blocks. We repeat this removal of instances and measure MapReduce performance until our cluster consists of 20 nodes. In general we do not have to remove nodes individually. Instead after collecting all performance measurements one can simply remove the all desired nodes at once.

One problem with this approach is how to determine the best target cluster size? In theory a cluster with a single node will have the minimal value $HET_{cluster}$ but then MapReduce runtime will be awful for large datasets. So how many nodes should we drop for optimal performance? We have to consider several factors to answer this question:

- Do we only consider the performance or also the price ration? If we would not care about the price we would allocate a very large cluster and then drop a large percentage of the nodes. But in a realistic setting this would be too expensive.
- The relative performance difference between nodes. If all nodes have by chance a similar performance one does not have to reduce the cluster size at all.
- How is the distribution of performance between nodes? For example, what is the best solution in cases where in a 20 node cluster we have 10 nodes of processor type a and 10 nodes of processor type b?

To deal with these problems our approach considers the relative decline in $HET_{cluster}$ if we remove an instance from the cluster. Consider a cluster with n nodes, where all nodes have almost equal performance and hence contribute the same amount of heterogeneity to $HET_{cluster}$. When removing a random node x, we would expect: $HET_{cluster} - HET_{cluster/x} \approx \frac{HET_{cluster}}{n}$. In such case we should not remove the node as it is too similar to all other nodes. The above experiments showed that a node should be removed if its contribution to $HET_{cluster}$ is more than twice the expected value, hence: $HET_{cluster} - HET_{cluster/x} > \frac{HET_{cluster}}{n}$. We continue to remove nodes until there is no node fulfilling this criteria anymore.

4.1.2 System driven Cluster Optimization

As a second strategy we try to reduce the heterogeneity in the cluster by considering the underlying CPU type. As we saw in Section 3.3 a virtual instance might run on different CPU types (and overall systems) each having different performance characteristics. Therefore this approach tries to obtain a more homogeneous cluster in respect to underlying systems. To achieve this goal we determine the dominant system types (i.e., the CPU types which has the largest share across all instances in the given cluster). When removing instances we choose instances from the least frequent group(s). A central advantage of this approach is that one does not have to run costly benchmarks as for the variance driven approach.

4.1.3 Performance driven Cluster Optimization

As baseline for the improvement of MapReduce performance we also use the performance driven approach. Here one always removes the least performant node. This could for example be a node having a lot contention with co-located instances. As a measure of node performance we choose $PERF_{cluster}$ as a simple linear combination of the individual performance characteristics. Note, that this definition is corresponding to the definition of $PERF_{cluster}$ in Section 3.6.2.

$$PERF_{node} = \frac{CPU}{GAVG(CPU)} + \frac{IO - Read}{GAVG(IO - Read)} + \frac{IO - Write}{GAVG(IO - Write)} + \frac{Network}{GAVG(Network)}$$

4.1.4 Comparison

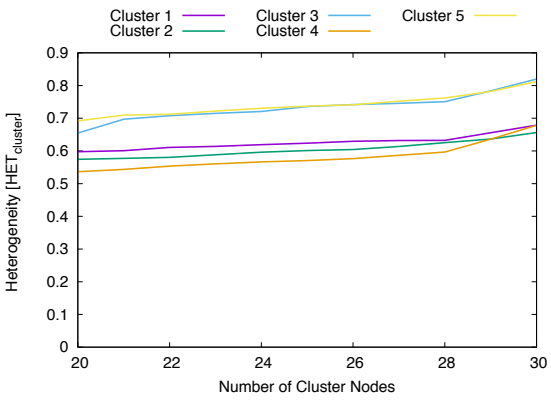
In order to evaluate these approaches we start with 30 nodes cluster and then iteratively remove nodes until the cluster consists of 20 nodes. At each step we recompute $HET_{cluster}$ and measure the average MapReduce job runtime (as before we report the average of three runs). We repeated this procedure for a total of five different initial clusters and the three different approaches.

In Figure 4.1 we show the decrease of $HET_{cluster}$ for each iteration. We can see that the variance driven optimisation achieves the fastest decrease of $HET_{cluster}$ across all clusters. This is expected as we choose the node to drop

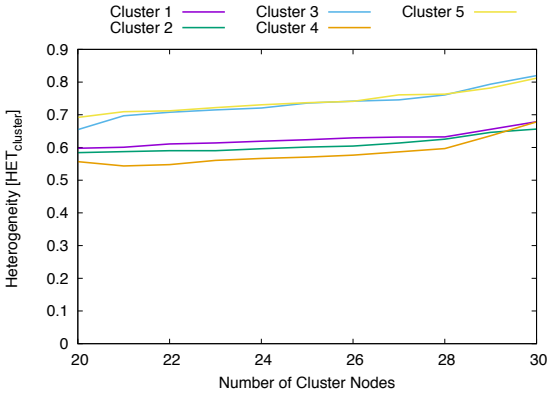
according to this criterium. The other two approaches are only close seconds, at many cluster sizes achieving the same $HET_{cluster}$ score.

Especially, if we look at the $HET_{cluster}$ for the 20 node clusters we see a comparable performance between the three approaches. So for a larger set of dropped nodes the approach does not matter too much.
This is confirmed when comparing the removal decision (i.e., nodes which have been dropped by one approach but not the other). The variance driven and system driven approach differ by only one to three decisions (with an average of 1.8 decisions). When comparing the performance driven approach to the variance driven approach the difference is only slightly worse with one to four dropped nodes and an average of 2 nodes differing between both approaches.

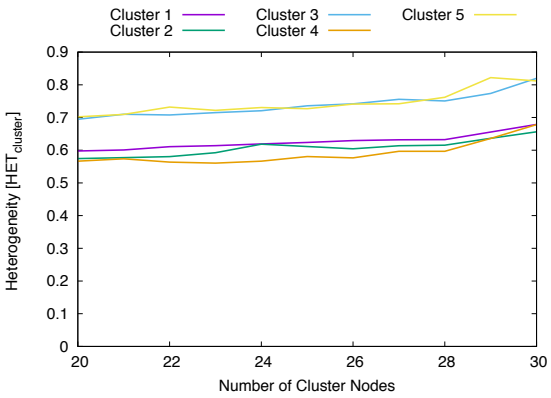
Next, let us compare the MapReduce job performance in Figure 4.2. Here the average runtime does not differ significantly between the three approaches. But interestingly, if we consider the average performance for each cluster size, we obtain the minimum average runtime for a cluster size of 28 nodes with an average improvement of 134 seconds compared to the 30 node initial cluster. This confirms our hypothesis that smaller, but more homogeneous cluster have a better MapReduce performance. The additional overhead we had to allocate is only at six percent of the initial cluster size, especially if we consider that many MapReduce jobs run for several hours and one can schedule several MapReduce jobs on a given cluster this overhead quickly pays off. Allocating more additional nodes—and hence terminating more nodes—decreases the overall MapReduce performance, but the the performance per node i.e., $\frac{\text{number nodes}}{\text{job runtime}}$ still improves.



(a) Variance Driven Optimisation.

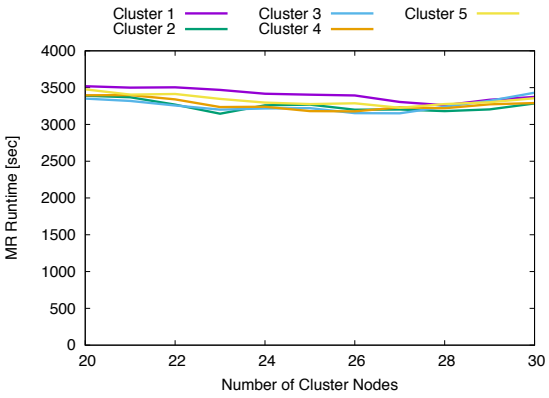


(b) System Driven Optimisation.

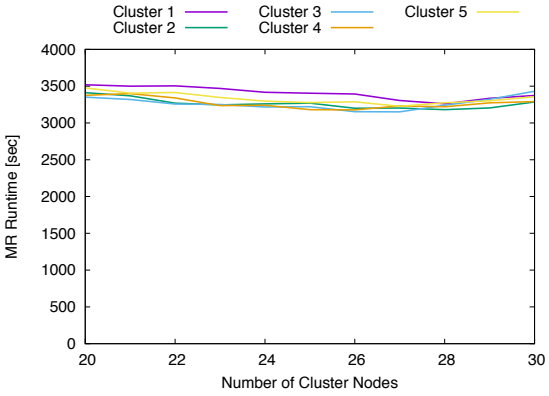


(c) Performance Driven Optimisation.

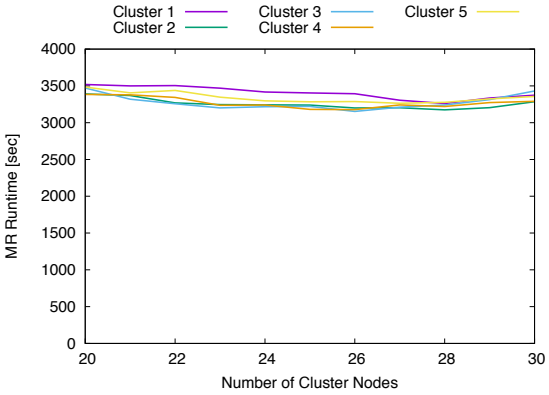
Figure 4.1: Cluster Heterogeneity when Reducing the Cluster Size.



(a) Variance Driven Optimisation.



(b) System Driven Optimisation.



(c) Performance Driven Optimisation.

Figure 4.2: Average MR Job Runtimes when Reducing the Cluster Size.

4.2 Conclusion

As we have seen in Section 3.6 Performance variance among different nodes in a cluster has a negative impact on MapReduce performance at different levels including HDFS upload and job runtime. This is caused by the assumption of homogeneous node performance which is inbuilt into the various scheduling components.

In this Chapter, we considered three different algorithms for reducing the cluster performance. All were based on the idea of allocating a larger cluster than required for a brief period of time and then select the optimal subset of nodes. Variance Driven Optimisation select this subset by trying to reduce $HET_{cluster}$ directly. Variance Driven Optimisation yields the best performance, measured by MapReduce runtime, but requires the upfront execution of benchmarks on each node.
Performance Driven Optimisation also requires the prior execution of benchmarks, but then selects the nodes with the best performance. Still the resulting MapReduce performance is comparable to Variance Driven Optimisation. Both approached can also optimise for more specialised setting, such as IO bound jobs requiring a focus on IO performance.
System Driven Optimisation instead uses the observation made in Chapter 3.6 that the performance variance for a given underlying system type is much low lower than the variance across different system types. This has the advantage that we do not have to run expensive benchmarks upfront but only have to check the /etc/proc file on each node. As the MapReduce runtimes are comparable to the other two approached, we would recommend this approach to settings without additional requirements such as IO bound jobs.

Overall we saw that by acquiring just a few additional nodes in the beginning, we can improve the MapReduce performance by five to ten percent compared to the initial larger cluster. Especially for long running MapReduce jobs or MapReduce cluster executing several jobs the overhead of the additional nodes at start up time can become negligible. The number of additional nodes to allocate is a tradeoff between cost and overall runtime of the cluster, we have seen performance per node, i.e., $\frac{\text{number nodes}}{\text{job runtime}}$ improves even if we drop 1/3 of the initial nodes.

5 Processing Incremental Data using MapReduce

MapReduce has become quite popular to analyse very large datasets. Nevertheless, users typically have to run their MapReduce jobs over the whole dataset every time the dataset is appended by new records. Some researchers have proposed to reuse the intermediate data produced by previous MapReduce jobs. However, these existing works still have to read the whole dataset in order to identify which parts of the dataset changed. Furthermore, storing intermediate results is not suitable in some cases, because it can lead to a very high storage overhead.

In this Section, we propose Itchy, a MapReduce-based system that employes a set of different techniques to efficiently deal with growing datasets. The beauty of Itchy is that it uses an optimizer to automatically choose the right technique to process a MapReduce job. In more detail, Itchy keeps track of the provenance of intermediate results in order to selectively recompute intermediate results as required. But, if intermediate results are small or the computational cost of map functions is high, Itchy can automatically start storing intermediate results rather than the provenance information. Additionally, Itchy also supports the option of directly merging outputs from several jobs in cases where MapReduce jobs allow for such kind of processing. We evaluate Itchy using two different benchmarks and compare it with Hadoop [44] and Incoop [94]. The results show the superiority of Itchy over both baseline systems. In terms of upload time, Itchy has the same performance as Hadoop and it is $\sim$1.6 times faster than Incoop. In terms of job runtime, Itchy is more than order of magnitude faster than Hadoop (up to $\sim$41 times faster) and Incoop (up to $\sim$11 times faster).

Table 5.1: Example Data for running Example

(a) Input to the first-job (`SALES`)

	id	category	price
r1	100	b	4
r2	189	b	6
r3	132	c	2
r4	73	f	9
r5	150	f	9

(b) Appended records (`SALES'`)

	id	category	price
r6	208	g	3
r7	205	c	6

5.1 Introduction

The MapReduce paradigm and especially the open source implementation Hadoop have become the de facto standard for large scale data processing [106, 168, 170, 167]. This success is mainly based on the ease-of-use of MapReduce, which enables non-expert users to process terabyte-sized datasets. In practice, these large datasets frequently change over time, usually by having new data appended. One example for such changing dataset is Google's inverted search index: webpages are added, updated, and deleted all the time. Still, the ratio of changed data is typically only a small fraction compared to the total dataset size [151]. Using the standard Hadoop MapReduce framework, one has to process all the web pages again in order to update a single page in the inverted search index. Indeed, this is not suitable for large datasets as it requires a lot of computing resources and also time until the new search index becomes available.

Let us see through the eyes of a typical Hadoop user (say Alice). Alice is a data analyst at a company and executes queries over several terabytes of data every day. A typical job for her could be similar to the following aggregation query: `SELECT category, AVG(price) FROM SALES GROUP BY category`. Alice runs this query over the data shown in Table 5.1(a). For simplicity, we only consider very few records, but in reality such datasets can have many gigabytes to terabytes of data.

To answer this query with MapReduce, Alice writes a MapReduce job (say *JobAvg*) in which the map function simply emits the `category` attribute as intermediate key and the `price` attribute as intermediate value. The reduce function then just takes the average over the list of intermediate values it received as input. Alice executes JobAvg for the first time on `SALES` (which takes a while for large input datasets) and receives the correct output: {(b,5),(c,2),(f,9)}. We

denote the first run of a given MapReduce job as *first-job* from here on. After a few days, Alice appends the records in `SALES'` Table 5.1(b) to the `SALES` Table 5.1(a). Again, we only consider very few records for simplicity. As Alice wants to retrieve the correct output: $\{(b,5),\underline{(c,4)},(f,9),\underline{(g,3)}\}$, she has to run JobAvg over both the `SALES` table and the appended records in `SALES'`. We denote such later run of a MapReduce job on the grown dataset as *incremental-job*. The main problem for appending data in MapReduce is that typically one has to reconsider some records of the `SALES` table when processing the records in the `SALES'` table. In Alice's example this would be the record $(132, c, 2)$ in the `SALES` table. This record belongs to the same category c as the newly appended record $(205, c, 6)$ and hence Alice has to consider it for computing the correct average for category c. However, using Hadoop MapReduce, Alice has to read the entire `SALES` table again to recompute the correct average for category c. This takes a while as `SALES` is typically very large. In fact, this was one of the main reasons for Google to move away from MapReduce to a new system coined Percolator [151]. Nonetheless, moving to a different system requires Alice to adopt a new programming paradigm and adapt her applications accordingly. Thus, Alice still wonders if there exists a way in MapReduce to process the new records in `SALES'` without reading the entire `SALES` table again.

Some existing works (e.g., Incoop [94] and DryadInc [152]) address this problem by applying memoization to map and reduce tasks. A memoized task caches the intermediate results for the given input while executing a MapReduce job for the first time. When executing an incremental-job, each map task checks whether its input has changed with respect to the first-job. If the input did not the change, the map task from the incremental-job simply returns the cached results without actually processing its input. A disadvantage of these approaches is that they still have to read the entire input dataset in order to identify which parts of the dataset changed. This means that each map task expects to receive the same data input in order to reuse previously produced intermediate results. Furthermore, storing intermediate results can lead to a very high storage overhead. For example, in case of Incoop the overhead can be up to 9 times the initial input size [94]. As datasets usually have several terabytes of data, storing all intermediate results for many jobs is not an option in most Hadoop clusters.

Therefore, Alice still has the problem of dealing with her growing datasets in an efficient manner.

5.1.1 Idea

To make Alice's life easier, we propose Itchy (*Incremental TeCHniques for Yellow* elephants): a MapReduce framework to efficiently process incremental-jobs. The main observation behind Itchy is that different MapReduce jobs require different approaches for processing incremental-jobs. Therefore, in contrast to previous works, Itchy provides an optimiser to automatically choose the best technique for performing each MapReduce job. The main idea of Itchy is to identify the characteristics of each incoming job and accordingly store additional information that allows for executing incremental-jobs efficiently. This additional information can either be the provenance of intermediate results, i.e., which input record produces which intermediate key, or the intermediate results themselves, i.e., the output produced by map tasks.

For MapReduce jobs producing large-size intermediate results or having map functions that are not CPU intensive, Itchy stores a mapping from intermediate keys to input records in the form of *Query Metadata Checkpoints* (QMCs) [156]. However, in contrast to RAFT [156] that stores QMCs at a map task level (for failover purposes), Itchy stores QMCs at a intermediate key level. Hence, Itchy can recompute a given individual record independently of the map task that originally processed the individual record.

Still, QMC information might be larger than intermediate results themselves. Thus, for MapReduce jobs producing very small-size intermediate results or having CPU intensive map functions, Itchy stores intermediate results. This is similar to the memoization approaches presented in [94, 152]. However, Itchy differs from these previous works in that it keeps track of intermediate results at the intermediate-key level rather than at the task level. Again, this allows Itchy to selectively recompute any given intermediate key.

Itchy can also decide to not store any additional information regarding intermediate results in cases that MapReduce jobs allow for merging final outputs. For example, consider a MapReduce job computing the total revenue of selling a given product. In this case, Itchy can merge the final outputs of the first-job and an incremental-job. As a result, Itchy does not have to process again any record in the map phase or intermediate results in the reduce phase.

5.1.2 Research Challenges

The idea behind Itchy triggers many interesting challenges when running a first-job and an incremental-job:

- **First-Job.** Which different kinds of QMCs can be used for improving MapReduce jobs over growing datasets? What is the tradeoff between storing QMCs and intermediate results? How can we efficiently store QMCs or intermediate results during the first-job without any changes to users' jobs?
- **Incremental-Job.** How can we efficiently utilize QMCs or intermediate results to recompute only some records of the `SALES` table? How can we do so with minimal changes to users? How can we process both the `SALES'` table and parts of the `SALES` table in a single MapReduce job for efficiency? How can we efficiently merge the outputs of the first-job and an incremental-job?

5.1.3 Contributions

We present Itchy, a framework to efficiently deal with growing datasets. The main goal of Itchy is to execute incremental-jobs by processing only relevant parts from the input of the first-job (e.g., the `SALES` table) together with the input of incremental-jobs (e.g., the `SALES'` table). We make the following contributions:

- We first identify different classes of incremental MR jobs. Each of these classes allows for different optimizations while processing growing datasets. We then propose three different techniques to efficiently process incremental-jobs, namely Itchy QMC (which stores provenance information), Itchy MO (which stores the Map Output, hence intermediate data), and Itchy Merge (which combines the output of MapReduce jobs).
- We show that deciding between Itchy QMC and Itchy MO is basically a tradeoff between storage overhead and runtime overhead. Thus, we present a decision model that allows Itchy to automatically balance the usage of QMCs and intermediate data to improve query performance. Thereby, Itchy can decide the best option for each incoming MapReduce job considering both job runtime and storage overhead.

- We present a framework that implements the Itchy ideas in an invisible way to users. This framework uses Hadoop and HBase to store and query both QMCs and intermediate data. The Itchy implementation includes many non-trivial performance optimizations to make processing incremental-jobs efficient. In particular, Itchy runs map tasks in two map waves: one map wave to process the incremental dataset (containing the appended records) and one map wave to process the initial dataset (the input dataset to the first-job). This allows Itchy to perform incremental-jobs in a single MapReduce job instead of two MapReduce jobs. As a result, Itchy avoids reading, writing, and shuffling growing datasets twice.
- We present an extensive experimental evaluation of Itchy against Hadoop MapReduce framework and Incoop. Our results demonstrate that Itchy significantly outperforms both Hadoop MapReduce and Incoop when dealing with incremental-jobs. We also show that Itchy incurs only negligible overhead when processing the first-job. Furthermore, we provide a detailed comparison between using QMCs or intermediate data in different settings.

5.2 Hadoop MapReduce Recap

Let us first provide some background knowledge for the remainder of this Chapter. First of all, since Itchy is based on Hadoop MapReduce, we discuss the Hadoop MapReduce workflow in more detail. Then, we discuss the main problem of processing incremental-jobs with Hadoop MapReduce.

5.2.1 Hadoop MapReduce Workflow

Let us look in detail at what happens when Alice executes her MapReduce job `JobAVG`, which is shown below in pseudocode:

```
map(Key k, value v):
   emit(v.category, v.price);
reduce(IntermediateKey ik, values v[]):
   emit(ik, avg(v));
```

Before starting her MapReduce job, Alice uploads the SALES table (i.e., Table 5.1(a)) into the *Hadoop Distributed File System* (HDFS). Once her dataset

is uploaded to HDFS, Alice can execute JobAVG using Hadoop MapReduce. In turn, Hadoop MapReduce executes JobAVG in three main phases: the *map phase*, the *shuffle phase*, and the *reduce phase*.

Map Phase. Hadoop MapReduce first partitions the input dataset into smaller horizontal partitions, called *input splits*. Typically, an input split correspond to an HDFS block. Hadoop MapReduce creates a map task for each input split. Then, the Hadoop MapReduce scheduler is responsible of allocating the map tasks to available computing nodes. Once a map task is allocated, the map task uses a *RecordReader* to parse its input split into key-value pairs. For Alice's MapReduce job, the RecordReader produces key-value pairs in the form `(SALES.id;(SALES.category,SALES.price))` for each line of input data. The map task then executes a map-call for each of key-value pair independently. The output of a map-call might be zero or more intermediate key-value pairs. For Alice's MapReduce job, the map output is in the form `(SALES.category;SALES.price)`. Hadoop MapReduce stores the map output in local disk.

Shuffle Phase. Hadoop MapReduce partitions the intermediate key-value pairs by intermediate key and assigns each partition to a different reduce task. The number of reduce tasks is user-defined. Then, Hadoop MapReduce shuffles the partitions through the network to its respective reduce task. This means that the intermediate results produced by all map tasks that have the same intermediate key will end up in the same reduce task.

Reduce Phase. The reduce task executes a reduce-call for each group of intermediate key-value pairs. The output of a reduce-call might be zero or more final key-value pairs. Finally, Hadoop MapReduce stores the output of reduce tasks in HDFS.

5.2.2 Incremental-Jobs in Hadoop MapReduce

A few days after Alice executed JobAvg, business continues and Alice receives additional `SALES'` records shown in Table 5.1(b)). At this point, Alice's boss wants to see the output of JobAvg including the records in Sales'. In an attempt to update the output from her first-job processing `SALES`, Alice tries to run an incremental-job only over the `SALES'` table. The result for category g from her incremental-job is correct since `SALES` contains no records for category g. However, the result for category c is incorrect as her incremental-job did not consider the record $(132, c, 2)$ from the `SALES` table. Therefore, using normal

Hadoop MapReduce, Alice has to run again her incremental-job over both the SALES and SALES' tables. This is because she has no way to specify the records from SALES that are relevant when processing SALES'. Notice that, records from the SALES table are only relevant, if they produce same intermediate key as some records from the SALES' table. Indeed, performing again an incremental-job over the input dataset of a first-job (the SALES table in Alice's example) is not a good idea since such dataset is typically in the order of terabytes.

5.3 Classes of Incremental-Jobs

After a while, Alice notices that there are different classes of incremental-jobs. First, appended records can contain *inserts* (i.e., new records) or *updates* (i.e., overwritten records). While inserts have a previously unseen primary key, which in Alice's dataset is the id attribute, updates overwrite previously seen records having the same primary key. We classify incremental-jobs whose input datasets contain updates as being in the *Update Class*. For datasets containing only inserts, we have first to know if we can simply merge the output of an incremental-job (*incremental output*) processing only the appended records with the output of the first-job (*base output*). If so, we can then process the first- and incremental-jobs independently and merge the base output with the incremental output. Therefore, whenever an incremental-job allows for such processing, we classify it as being in the *Merge Class*. Otherwise, we have to consider records from the SALES table when processing the appended records. We classify such incremental-jobs as being in the *Recompute Class*. In the following, we discuss each of these three different classes in more detail as each allows for different ways to process incremental-jobs.

(1.) Recompute Class. When running MapReduce jobs in the Recompute class, we have to consider records in the original input (SALES table) while processing appended records in Sales'. For example, consider we append the record $(7, c, 6)$ to Alice's SALES table, which belongs to the same category c as the record $(3, c, 2)$ in the SALES table. Hence, we have to consider the record $(3, c, 2)$ when computing the correct result $\{(c, 4)\}$. Then, Alice has to be aware that the incremental output might overwrite some values from the base output. In this example, the incremental output overwrites the base output for category c.

(2.) Merge Class. MapReduce jobs in this class allow us for merging the incremental output directly with the base output. This means that, for this kind of MapReduce jobs, we do not have to reconsider the input of the first-job.

Instead, we directly access the base output. Itchy Merge requires users to specify a *merge* function to combine the differential output with the base output. For example, assume we vary the query of `JobAvg` from Section 5.1 as follows: `SELECT category, SUM(price) FROM SALES GROUP BY category`. Here, we can simply add the base output and the differential output for each intermediate key. In general, we can specify such a merge function, if the reduce function is a distributive aggregation function [120]. In other words, the reduce function should have have the following property:

$$\exists \text{ function } G() : reduce(\{X_i\}) = G(reduce(\{X_{i,j}\})|j = 1, ..J)$$

Where $\{X_i\}$ is the set of all intermediate values, and $\{X_{i,j}\}$ is the set of intermediate values in partition j for a given partitioning of intermediate values.

(3.) Update Class. Similar to the Recompute Class, for the MapReduce jobs in the Update Class, we have to consider records from the input of the first-job in order to update the output produced by such records. However, in contrast to MapReduce jobs in the Recompute Class, we also have to consider that results computed from overwritten records need to be changed. For example, assume we append the record $(132, b, 10)$ to the `SALES` table. This appended record overwrites the record with key 132 in `SALES`. Since the category b of the newly appended record is different from the old category c, the output for both categories b and c changes and hence we have to update both outputs. Again, the incremental output overwrites the categories b and c from the base output.

5.4 Itchy

In this section, we present the ideas for efficient incremental processing in Itchy (Incremental TeCHniques for Yellow elephants). The central idea of Itchy is to store additional data while running a MapReduce job for the first time (i.e., when running the first-job). Then, Itchy uses this stored data to speed up incremental-jobs when processing appended records. The type of additional stored data varies among different incremental-jobs. We discuss in detail how Itchy processes incremental-jobs that are in the Recompute Class and in the Merge Class. Discussing how Itchy deals with updates is out of the scope of this thesis.

In the following, we first discuss the two techniques Itchy uses for incremental-jobs in the Recompute Class: *Itchy QMC* and *Itchy MO*. Generally speaking, while Itchy QMC stores *Query Metadata Checkpoints* (QMC) to recompute only a part of the input to the first-job, Itchy MO stores fine-granular intermediate results (i.e., Map Outputs) to not have to touch at all the input to the first-job.

Then, we present how Itchy supports merging different outputs for incremental-jobs in the Merge Class. Since Itchy can use all these three techniques for processing incremental-jobs, we thus discuss the tradeoffs among them. Based on these observations, we present a model that allows Itchy to automatically decide which of the three techniques to use for a given incoming MapReduce job.

5.4.1 Itchy QMC

Alice's problem is that using Hadoop she has to reprocess the entire terabyte-sized `SALES` table each time she appends a few records. Recomputing the entire terabyte-sized dataset is a serious performance overhead. Hence, the challenge is to identify those records from the `SALES` table that are relevant when processing the appended records. With this in mind, Itchy exploits the fact that relevant records in the `SALES` table can be identified by the intermediate key they produce. Indeed, records in the `SALES` table are only relevant to an incremental-job if they produce an intermediate key that is also produced by some appended record. Therefore, Itchy first processes the appended records using the map function. This allows Itchy to obtain the set of intermediate keys IK_{inc} produced by the appended records. Then, Itchy uses QMC to identify the relevant records from the input to the first-job. Notice that, RAFT [156] uses QMC to selectively recompute failed tasks. In contrast, here we show how Itchy generalises QMC to deal with incremental-jobs. Itchy introduces a mapping QMC_{ik} that maps each intermediate key ik to offsets in the input to the first-job (i.e., QMC_{ik}: $ik \rightarrow \{\text{offsets}\}$). An offset identifies a record in a dataset and consists of two parts: (1) an identifier of the dataset[1] and (2) the actual offset in the dataset. Thus, an offset requires 12 bytes of storage in total.

Figure 5.1 shows how Itchy collects the QMC_{ik} mapping when running the first-job. First, Itchy processes the `SALES` table using the user-defined map function as in standard Hadoop MapReduce (1). Next, Itchy stores a QMC_{ik} mapping for each intermediate key ik (2). For simplicity, we consider just the line number as offset in this example. Hence, the complete QMC_{ik} mapping for the `SALES` table looks as follows $\{(b \rightarrow r1, r2), (c \rightarrow r3), (f \rightarrow r4, r5)\}$. Then, as in Hadoop MapReduce, Itchy processes the intermediate results using the user-defined reduce function in order to produce the output of the MapReduce job (3).

[1] Itchy assigns an increasing integer value whenever a new dataset is encountered for the first time.

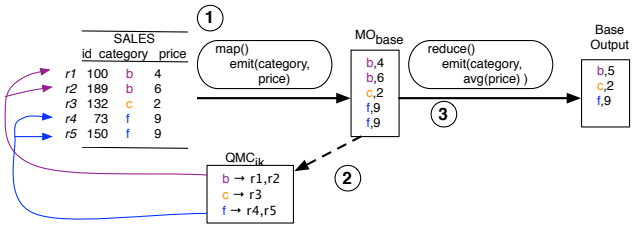


Figure 5.1: First-Job storing QMC_{ik}

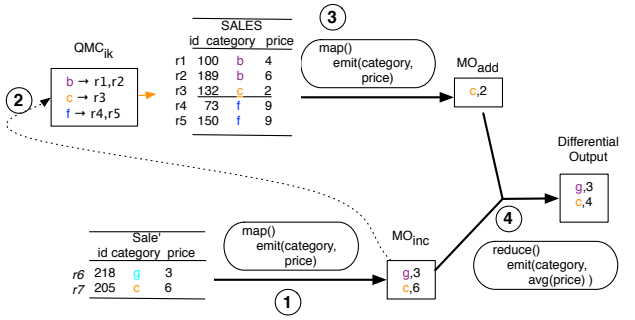


Figure 5.2: Incremental-Job using QMC_{ik}

Now, let us focus on how Itchy processes the appended records in the `SALES'` table (incremental-job). Figure 5.2 shows how Itchy processes an incoming incremental-job using previously collected QMC_{ik}. Notice that, Itchy uses the QMC_{ik} for identifying the relevant records in `SALES`. Thus, as a first step, Itchy executes the user-defined map function over the additional records in `SALES'` and computes the set of relevant intermediate keys IK_{inc} (1). This allows Itchy to identify the set of relevant records in `SALES` for processing the incoming incremental-job. In our example, the map output when processing only the records in Sales' MO_{inc} is $\{(g,3),(c,6)\}$ and IK_{inc} is $\{g,c\}$. Then, Itchy retrieves the associated offsets from the QMC_{ik} for each intermediate key ik_x in IK_{inc} (2). Notice that, the intermediate key $g \in \text{IK}_{inc}$ is not part of QMC_{ik}, because g is not produced by any record in `SALES`. In contrast, the record $(132, c, 2)$ in the `SALES` table produces the intermediate key c, which is in IK_{inc}. Therefore, Itchy retrieves only the relevant offset $\{r3\}$ from QMC_{ik}. This means that Itchy additionally has to process the record at position $r3$ in the `SALES` table. Itchy processes this relevant record $(132, c, 2)$ using again the user-defined map function and produces one map output from the original `SALES` table, $\text{MO}_{add} = \{(c,2)\}$ (3). Notice that, the user-defined map function might produce several intermediate keys for a single record. Thus, Itchy might end up with non-relevant intermediate keys in the reduce phase. This is because Itchy is actually only interested in one of the emitted inter-

mediate keys. To avoid this problem, Itchy post-filters the MO_{add} output so as to consider only relevant intermediate keys in the reduce phase. At this point, we already have all the intermediate results required for the correct result. Thus, Itchy just has to execute the user-defined reduce function over $\text{MO}_{inc} \cup \text{MO}_{add} = \{(g,3),(c,6),(c,2)\}$ ④. As a result, the reduce phase yields the correct differential output: $\{(g,3),(c,4)\}$.

5.4.2 Itchy MO

As discussed in the previous section, Itchy QMC recomputes some records from the `SALES` table when processing incremental-jobs. This is not suitable for jobs having expensive map functions, for example K-means clustering [112].

An alternative solution for these cases is to store the intermediate values directly instead of the offsets. For this, Itchy can also store the intermediate values for each intermediate key while processing the `SALES` table. The set of all mappings from intermediate keys to intermediate values is the map output (MO) itself, i.e., $ik \rightarrow \{intermediate\_values\}$. In our running example, the map output contains the following values for the `SALES` table: $\text{MO}_{base} = \{(b \rightarrow \{4,6\}),(c \rightarrow \{2\}),(f \rightarrow \{9,9\})\}$. Then, when processing the appended records in `SALES'`, Itchy retrieves the required intermediate values from MO_{base}.

Algorithm 1 shows how Itchy processes incremental-jobs using previously computed intermediate values. Like Itchy QMC, Itchy MO first computes the output of the map phase for the records in `SALES'`, i.e., $MO_{inc} = \{(g,3),(c,6)\}$ (line 2). Next, Itchy retrieves the relevant intermediate values from MO_{base}, previously computed by the first-job (lines 3-7). Recall that the intermediate key g was not produced by any record in `SALES`. Thus, Itchy MO only retrieves the intermediate value $(c,2)$ from MO_{base} and add it to MO_{add}. Thus, after this operation, $\text{MO}_{add} = \{(c,2)\}$. Then, Itchy can execute the user-defined reduce function over the $\text{MO}_{inc} \cup \text{MO}_{add} = \{(g,3),(c,6),(c,2)\}$ (line 8). As a result, Itchy again yields the correct differential output: $\{(g,3),(c,4)\}$ (line 9).

An important design decision of Itchy is that, in contrast to other systems such as Incoop [94], Itchy stores intermediate results at an intermediate key level rather than at a task level. This allows Itchy to avoid the *all-or-nothing* problem of other systems: other systems require running tasks from incremental-jobs to have exactly the same input as previous tasks from the first-job to be

Algorithm 1: MO-based processing of incremental-jobs

```
1 MO_add = {};
2 compute MO_inc;
3 forall the ik in MO_inc do
4     if ik ∈ MO_base then
5         MO_add.add(MO_base.getIntermediateValues(ik))
6 execute reduce phase for MO_inc ∪ MO_add;
7 output differential file ;
```

able to reuse previously computed results; Itchy can selectively reuse only those parts of the previously computed results having the required intermediate keys. Furthermore, Itchy is also flexible in terms of task scheduling as it can reuse previously computed results from any task regardless of where previous tasks were executed.

5.4.3 Itchy Merge

Let us now focus on how Itchy deals with jobs in the Merge Class (see Section 5.3). As an example of such job, we vary `JobAvg` from Section 5.1 to the following query `JobSum`[2]: `SELECT category, SUM(price) FROM SALES GROUP BY category`. Here, Itchy merges the output from processing only the appended records (i.e., the differential output) with the output of the first-job (i.e., the base output) using a user-defined *merge function*. For `JobSum`, this merge function simply adds, for each intermediate key, the results in the differential output with the results in the base output.

The problem here is to identify the parts in the base output to merge with while processing the appended records. Since the base output can have a size of several hundreds gigabytes, scanning all base output would indeed hurts performance. Dryad [152] uses for this purpose a Cache Server, which allows for key-value access. However, caching to this Cache Server in addition to writing to HDFS results in a higher cost for each storage operation. Therefore, Itchy uses another type of QMC for identifying the relevant parts in the base output, called QMC_{out}. QMC_{out} is a mapping from intermediate keys to offsets in the base output (i.e., $ik \rightarrow \{\text{offsets}\}$). Notice that, an offset in QMC_{out} represents the location of the final output produced by a given intermediate key.

Figure 5.3 shows the workflow when running `JobSum` for the first time (i.e., the first-job). Itchy first uses the user-defined map (1) and reduce (2) functions to

[2]The `AVG()` function in `JobAvg` is not suitable for merging.

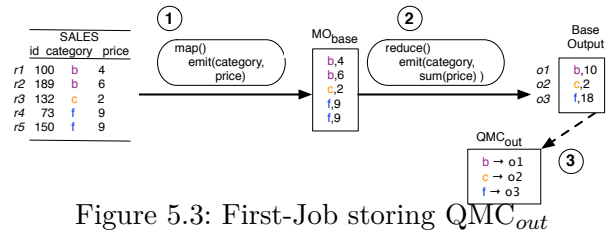


Figure 5.3: First-Job storing QMC_{out}

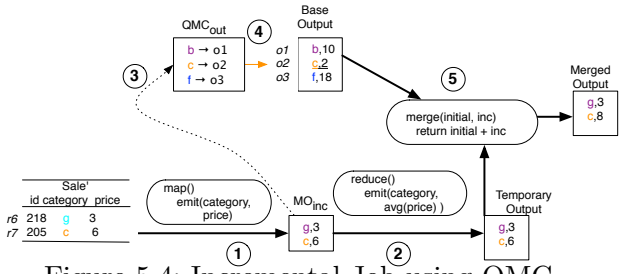


Figure 5.4: Incremental-Job using QMC_{out}

obtain the base output. When writing the base output to HDFS, Itchy also persists QMC_{out} (3). In this example, the QMC_{out} mappings are $\{(b \rightarrow o1), (c \rightarrow o2), (f \rightarrow o3)\}$. Again, we show only the line number as offset for simplicity.

Figure 5.4 shows Itchy's workflow when processing the appended records in `SALES'` (incremental-job). First, Itchy again uses the user-defined map (1) and reduce (2) functions to obtain the temporary output. Next, Itchy needs to merge the temporary output with the relevant parts of the base output. Therefore, for each intermediate key in MO_{inc}, Itchy retrieves the offset of the output record that was produced by the same intermediate key (3). For example, consider the intermediate key `c` in the incremental-job (see Figure 5.4): As the intermediate key `c` also occurred during the first-job, Itchy retrieves the entry $(c \rightarrow o2)$ from QMC_{out}. Next, Itchy needs to retrieve the actual output produced by the first-job for each offset in the retrieved QMC_{out}. For this, Itchy directly retrieves the record at the relevant offsets in the base output (4). In our example, this is the record $(c, 2)$ at offset $o2$. Itchy then uses the user-defined merge function to merge the retrieved records from the base output and the temporary output (5). In our example, the merge function receives the record $(c, 2)$ from the base output and the records $(g, 3)$ and $(c, 6)$ from the temporary output as input. The merge function then computes the sum outputs the merged output records $(g, 3)$ and $(c, 8)$. The output of Itchy Merge is again a differential file.

5.4.4 Decision Model

At this point, the reader might have one natural question in mind: *which of these three techniques (Itchy QMC, Itchy MO, or Itchy Merge) should Itchy use for a given job?* As Itchy Merge requires a user-defined merge function, let us first look at the main differences between Itchy QMC and Itchy MO. On the one hand, Itchy QMC implies that we have to recompute values, while Itchy MO allows us to simply retrieve those values. On the other hand, the intermediate results stored by Itchy MO can become quite large, while Itchy QMC only requires storing the intermediate key and an offset of 12 bytes size. Thus, Itchy has to trade storage overhead with recomputation time.

Let us first consider the storage overhead in more detail. The storage size overhead for intermediate results depends on the size of the attributes we require in the reduce phase. For example, Alice's example in Section 5.1 requires only the category and price attribute in the reduce phase. To show the impact of the size of attributes on the storage overhead, we vary the number of attributes in the reduce phase. Therefore, we extend the `SALES` table 5.1(a) by ten additional text attributes (`att_1`, `att_2`, ..., `att_10`). We assume that each text attribute has an average size of 25 bytes and the total dataset size is 1TB. We also modify the MapReduce Job `JobAvg` from Section 5.1 to include these ten new attributes as follows: `SELECT category, AVG(price), COUNT(DISTINCT att_1),..., COUNT(DISTINCT att_10) FROM SALES GROUP BY category`. As the counting of distinct values is performed in the reduce phase, the attributes are included in intermediate results. Then, we vary the number of attributes we included in the query. Figure 5.5 shows the resulting storage overhead for storing intermediate results when running the modified `JobAvg` job over the extended `SALES` table. For comparison, the purple line indicates the required storage size for Itchy QMC, which is independent of the number of included attributes. As one can see, the space overhead for storing intermediate results can be several orders of magnitude higher than for Itchy QMC.

The second factor impacting Itchy's tradeoff is the time for recomputing relevant records, which mainly depends on the average map runtime. The longer it takes the map function to process one record, the higher is the runtime overhead for recomputing relevant records in Itchy QMC. In contrast, Itchy MO avoids such recomputation costs by directly retrieving the respective values.

In contrast to both Itchy QMC and Itchy MO, Itchy Merge requires neither to recompute relevant records nor to store intermediate results. Thus, Itchy Merge seems to be the best option for processing incremental-jobs. However, it is not

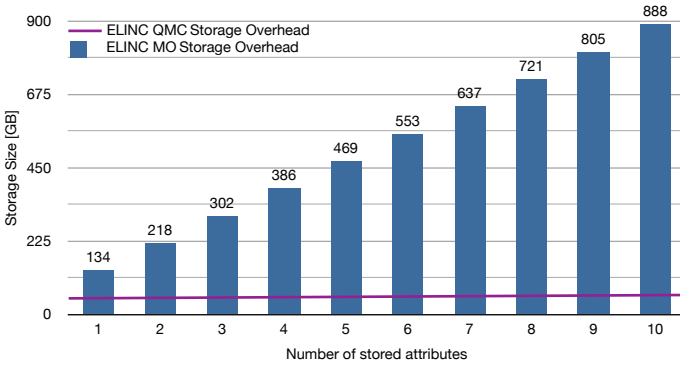


Figure 5.5: Storage overhead for intermediate results when varying the number of additionally included attributes

always possible to provide a merge function, e.g., for a join query. Therefore, we overall observe that all three techniques have their entitlement for existence as it is not always possible to specify a merge function and there is a tradeoff between storage overhead and runtime overhead.

This is the reason why Itchy uses all three techniques and includes a decision model to choose among them. Generally speaking, Itchy decides to use Itchy Merge whenever users specify a merge function, otherwise Itchy decides between Itchy QMC and Itchy MO based on the following measurements collected while running a MapReduce job for the first time:

- *Average Map Function Runtime* (*avgRuntime*). How long does the map function take to process an individual record? This is an estimate for the cost of recomputing a given intermediate key-value pair for Itchy QMC.

- *Average Intermediate Value Size* (*avgValSize*). How large is each intermediate value? This estimates the storage overhead when storing intermediate results.

- *Average Map Selectivity* (*avgSelectivity*). How many input records produce intermediate results? Not every input record necessarily produces intermediate results (e.g., a selection predicate in a SQL like query). Highly selective queries will reduce the amount of intermediate results to store. Hence, this measurement improves Itchy's estimates for storage overhead and runtime overhead.

Algorithm 2 describes in detail how Itchy decides which technique to apply. For each incoming MapReduce job, Itchy samples the input variables (average value size, average map selectivity and average map runtime) when processing the first map task. Itchy first checks, if the Itchy's third technique Itchy Merge (see Section 5.4.3) can be used. Therefore, the decision model considers

Algorithm 2: Itchy's Decision Model

```
1  if isMergeFunctionSpecified() then
2  |  Itchy Merge
3  else if avgValSize <=sizeof(QMC_ENTRY) then
4  |  Itchy MO
5  else
6  |  intermediateResultSize = avgValSize * avgSelectivity * #input records ;
7  |  if (avgValSize ) / avgRuntime < threshold then
8  |  |  Itchy MO
9  |  else
10 |  |  Itchy QMC
   |  |
```

whether a merge function was specified by the user (line 1). If so, Itchy uses Itchy Merge (line 2). The rational for this decision is that Itchy Merge has to store no intermediate values (as in Itchy MO), but at the same time it requires no recomputation of relevant records (as in Itchy QMC see Section 5.4.3 for details). If no merge function is specified, the decision model checks whether the estimated storage size for intermediate results is smaller than or equal as for QMC_{ik} (line 3). This is the case when the intermediate value is only an integer value. In such a case, Itchy uses the Itchy MO technique (line 4). Otherwise, Itchy estimates the storage size for storing intermediate results (line 6). Then, Itchy bases its decision on the ratio of average intermediate value size and average map function runtime (line 7). If such a ratio is smaller than a given threshold and hence intermediate results are small compared to the map function runtime, Itchy uses the Itchy MO technique (line 8). Otherwise, Itchy uses the Itchy QMC technique (line 10). Notice that, users can specify this threshold according to their needs.

5.4.5 Applicability to other MapReduce Jobs

So far we discussed the applicability of Itchy to individual single relational queries. Naturally the question arises the question how Itchy 's ideas extend to more more complex MapReduce jobs such as multiple relation as input or multistage jobs. Itchy can process MapReduce jobs having multiple relations without any alterations. Itchy MO and Itchy Merge work with the intermediate results or the final output respectively and hence are not effected by having multiple input relations. Itchy QMC considers offsets in the input, but as Itchy QMC stores a file identifier along with each offset it can handle multiple input relations. Another question is how Itchy supports Multistage MapReduce jobs consisting of several individual MapReduce jobs (usually because the require

disjoint shuffle phases). When dealing with such jobs Itchy faces the problem that usually in between job results are not persisted (or deleted after the job has finished). Hence Itchy QMC can only access the input data for the first job and Itchy Merge can only access of the last job's output. This leaves us with Itchy MO for dealing with such workloads as here we persist the intermediate results for each job. Itchy MO then processes the first job in the chain as is, next Itchy MO takes the differential output of the previous stage as the incremental input to the next stage until it outputs the final differential output.

5.5 Correctness

We now prove that Itchy produces indeed the correct differential output when processing incremental-jobs. We focus on incremental-jobs in the Recompute Class, i.e., we focus on proving the correctness of Itchy QMC and Itchy MO. It is worth noting that we do not prove the correctness of Itchy Merge as it depends on the user-defined merge function. We compare the differential output with Hadoop's output when processing both the input to the first-job (in the following denoted as *base input*) and the input to incremental-jobs (in the following denoted as *incremental input*) in a single MapReduce job. We denote the output of this single Hadoop job as $out_{Hadoop-merged}$, the base output produced by Itchy as out_{base}, and the differential output produced by Itchy as out_{inc}. Respectively, we denote the input to the reduce phase as $RI_{Hadoop-merged}$ (for Hadoop processing the merged input) and $RI_{Itchy-inc}$ (for Itchy processing the incremental input).

In the following, we first show in Lemma 1 that Itchy produces the correct intermediate values. Next, we show in Lemma 2 that Itchy also produces the correct set of intermediate keys. We then combine both lemmas to show the overall correctness of Itchy when processing MapReduce jobs in the Recompute Class.

Lemma 1. For each intermediate key ik in the intermediate results produced by Itchy (i.e., $\mathrm{RI}_{Itchy-inc}$), the associated intermediate values are the same as in Hadoop (i.e., $\mathrm{RI}_{Hadoop-merged}$).

Sketch. Let us consider an arbitrary intermediate key $ik_x \in RI_{Itchy-inc}$. Now, depending on the technique used by Itchy, there are two different options to obtain the intermediate values:

(1.) Itchy QMC recomputes all records from the base input that produce the same intermediate key ik_x. Itchy then combines these intermediate values with intermediate values produced by the incremental input to obtain $\text{RI}_{Itchy-inc}$. The intermediate values for intermediate key ik_x in $\text{RI}_{Hadoop-merged}$ are computed by processing all records (in particular, all records producing ik_x). Therefore, in both cases, the same set of records produces the same intermediate values for ik_x. Hence, a deterministic map function will produce the same set of intermediate values.

(2.) In case of Itchy MO, $\text{RI}_{Itchy-inc}$ is the union of the stored intermediate results from processing the base input and from processing only the incremental input. On the other hand, the records producing $\text{RI}_{Hadoop-merged}$ are the union of records from the base input and the incremental input. Hence, the intermediate values for $ik_x \in RI_{Itchy-inc}$ and $ik_x \in RI_{Hadoop-merged}$ are produced by the same records. Thus, assuming a deterministic map function, the intermediate values for $ik_x \in RI_{Itchy-inc}$ and $ik_x \in RI_{Hadoop-merged}$ are equal.

Lemma 2. Let $IK_{changed}$ be the subset of intermediate keys in $\text{RI}_{Hadoop-merged}$ for which the intermediate values differ from the output produced by processing the base input. The set of intermediate keys $RI_{Itchy-inc}$ and the set of intermediate keys $IK_{changed}$ are equal. Formally, $ik_x \in RI_{Itchy-inc} \iff ik_x \in IK_{changed}$.

Sketch. To demonstrate $ik_x \in RI_{Itchy-inc} \iff ik_x \in IK_{changed}$, we show that each sides of the equation implies each other.
(1.) $ik_x \in IK_{Itchy-inc} \implies ik_x \in IK_{changed}$: if $ik_x \in RI_{Itchy-inc}$, then there exists at least one appended record r_x that produces ik_x. Recall that Itchy applies post-filtering to emit only relevant intermediate keys. As the record r_x is an appended record and not part of the base input, the produced intermediate key ik_x will be part of $IK_{changed}$.

(2.) $ik_x \in IK_{changed} \implies ik_x \in IK_{Itchy-inc}$: if $ik_x \in IK_{changed}$, then $\text{out}_{Hadoop-merged}$ and out_{base} contain different intermediate values for ik_x. Hence, there must exists at least one appended record r_x producing ik_x. As r_x is part of the Itchy's incremental input, the produced intermediate key ik_x is then also part of $IK_{Itchy-inc}$.

Now, we combine both lemmas to show that $\text{out}_{Itchy-inc} = \text{out}_{Hadoop-merged} \setminus \text{out}_{base}$. Recall that the reduce phase, which further processes these intermediate key-value pairs to obtain the output, is the same for Itchy and Hadoop. Therefore, for the same set of intermediate key-value pairs given as input to

the reduce phase, Itchy and Hadoop produce the same output. Thus, the combination of both lemmas yields: $\text{out}_{Itchy-inc} = \text{out}_{Hadoop-merged} \setminus \text{out}_{base}$.

5.6 Implementation

We now discuss, how the ideas from Section 5.4 are implemented in Itchy. The implementation follows two goals: (i) hiding the incremental processing from users and (ii) providing high efficiency in terms of runtime and storage overhead. In the following, we present the nontrivial changes to the Hadoop MapReduce framework for implementing Itchy's ideas. First, we discuss where and how to persist the QMC mappings or respective intermediate results. Then, we present the implementation for Itchy QMC and Itchy MO in detail. For this, we show the workflow for processing the first- and incremental-jobs for both techniques. Then, we give details of Itchy Merge for merging different outputs.

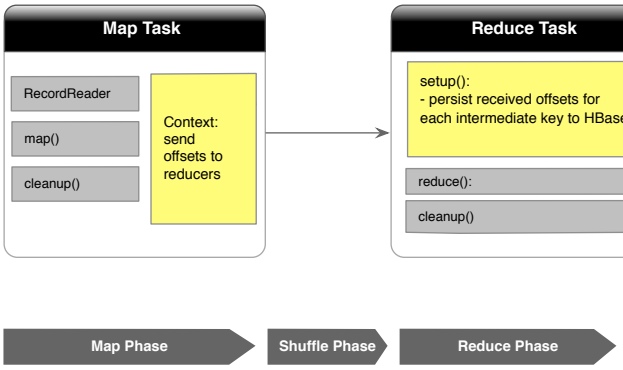
5.6.1 Persisting incremental Data

One key design decision for Itchy, is where and how to persist the *incremental data*, meaning the QMC_{ik}, MOs, or QMC_{out} efficiently. One crucial aspect is that Itchy should be able to retrieve the incremental data from all nodes in the cluster. Otherwise, Itchy would have to consider the storage locations of the incremental data for task scheduling. This is because, the scheduler would have to allocate each task on the node storing the incremental data required by the task. Thus, to make the data highly available, we evaluated a number of alternatives allowing for distributed data access including plain HDFS files and different NoSQL systems (such as MongoDB [100], Project Voldemort [67], and HBase [8]). We decided to use HBase as it offers reliable and scalable performance in combination with MapReduce jobs.

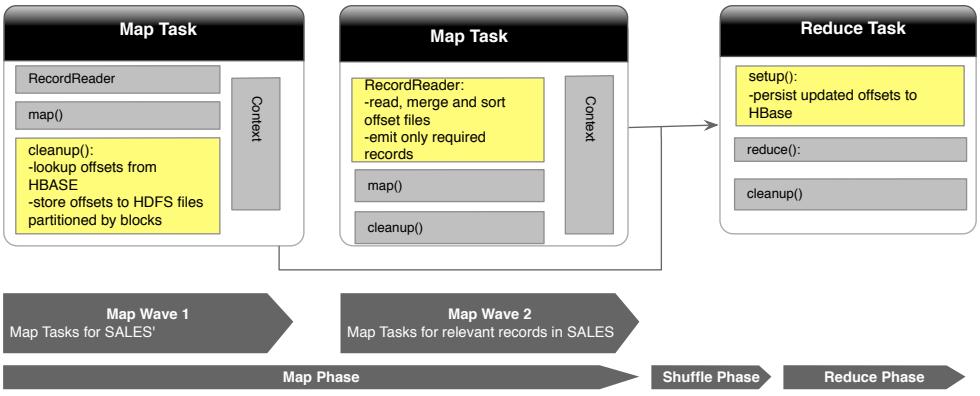
Itchy's HBase layout contains a single table for storing all incremental data. The table uses a binary representation of the intermediate key as `row key`, allowing for efficient lookups based on the intermediate key. In addition, the layout defines for each of the three techniques a single column family[3]. This layout allows Itchy to store all incremental data for a single intermediate key in a single row. In case of Itchy QMC, the column family only contains two attributes: `offset` and `datasetID`. The `offset` attribute stores a concatenated list of all

[3] A logical and physical grouping of several attributes.

the offsets in the dataset for the specific intermediate key. The `datasetID` attribute stores the concatenated binary representation of the dataset identifier for each offset. The column family for Itchy MO also contains two attributes: `intermediateValue` and `length`. Here, the `intermediateValue` attribute is a binary concatenation of all intermediate values for that intermediate key. As intermediate values can have variable lengths, the length attribute stores the lengths of the intermediate values. For storing QMC_{out} the respective column family looks similar to the case for storing QMC_{ik} and contains two attributes: `offset` and `datasetID`. These attributes again store a list of offsets and dataset ids. In contrast to Itchy QMC, the `offset` and `datasetID` both refer to the output of the first-job.



(a) Processing the First-job



(b) Processing the Incremental-job

Figure 5.6: Itchy QMC

5.6.2 Itchy QMC

Processing the first-job. While processing the first-job, Itchy has to store the QMC_{ik} mappings to HBase for each intermediate key. The challenge here is to do this efficiently by minimising the number of storage operations. Recall that Itchy can obtain the intermediate key as well as the current offset in the input dataset during the map phase. Therefore, the reader might think that the right time to store the QMC_{ik} mappings is during the map phase. However, this approach requires many HBase storage operations as each intermediate key-offset pair would be stored individually. In addition, this approach requires HBase to concatenate the offsets for each given intermediate key, which also increases the cost of each storage operation. Instead, Itchy QMC proceeds as shown in Figure 5.6(a). Itchy concatenates the offsets at the end of the intermediate values during the map phase. Itchy then utilises the shuffle phase to piggy back the offsets on top of the data send from the map tasks to the reduce tasks. The shuffle phase automatically groups and partitions the intermediate values and offsets by intermediate key. When deserializing the intermediate values before calling the reduce function, Itchy extracts the offsets from the intermediate values. In this way Itchy has all offsets for a given intermediate key and can store them in a single HBase storage operation. The remainder of the reduce function is then executed as in the Hadoop MapReduce workflow.

Processing the incremental-job. When processing incremental-jobs, Itchy uses the QMC_{ik} mappings to identify the records in the base input as discussed in Section 5.4. Itchy requires two steps to do so. First, Itchy has to retrieve the relevant offsets from the QMC_{ik}mappings. Second, Itchy has to process the appended records together with the relevant records in the base input. Itchy identifies the relevant records using the previously retrieved offsets as discussed in Section 5.4.1. A simple and naive implementation can achieve this by using two separate MapReduce jobs. Unfortunately, this approach is not efficient as it incurs the overhead of initialising and running two MapReduce jobs. Therefore, Itchy processes the appended records and parts of the base input in a single MapReduce job. This is challenging as Itchy only knows which parts of the original input it has to process after processing all appended records. In the standard Hadoop MapReduce framework all map tasks are scheduled independently and there is no control for such scheduling conditions.

Therefore, to deal with this requirement, Itchy introduces the concept of *map waves* as shown in Figure 5.6(b). The idea is to assign each map task to a map wave. In particular, Itchy assigns all map tasks processing the appended

records to map wave 1 and all map tasks processing relevant records from the base input to map wave 2. During the map phase Itchy only schedules those tasks that belong to the current map wave. Only after all task of a map wave have finished, Itchy moves to the next map wave. Hence, both datasets are processed strictly sequential. Note, that the concept of map waves is independent of the scheduling policy (such as Delay [173] scheduler) used by Hadoop as map waves are implemented on a per Job basis (i.e., in the JobInProgress class).

When a map task processing the appended records (in map wave 1) finishes, Itchy looks up the offsets in HBase for the intermediate keys produced by this map task. One problem here is that several map tasks can produce the same intermediate key ik_x. Thus, looking up the offsets for all intermediate keys in every map task can lead to a large number of unnecessary HBase lookups. To reduce the number of unnecessary HBase calls, Itchy caches the set of already seen intermediate keys between several map tasks. For this, Itchy reuses the Java Virtual Machine (JVM) in one node across several map tasks. This allows all map tasks running in the same JVM to share a cache for already seen intermediate keys. As a result, a map task queries HBase only for intermediate keys that did not occur before in any previous map task that ran inside the same JVM. Itchy then sorts the retrieved offsets and partitions them into HDFS files according to the blocks of the base input. This means that Itchy writes all offsets from a map task that belong to a specific block in the base input into a single file. Notice that each map task could potentially produce one file for each block in the base input.

When Itchy starts the second map wave, we face another challenge: Itchy should schedule only map tasks for those blocks containing at least one relevant offset; the other map tasks points to blocks containing only irrelevant records. However, Hadoop MapReduce (and therefore Itchy as well) creates map tasks when initialising a MapReduce job and Itchy does not know the relevant records till it finishes executing all map tasks in map wave 1. Therefore, to deal with this problem, Itchy will unschedule all map tasks that are not required. Unscheduling means directly marking a map task as finished without executing it. Itchy also ensures that no intermediate results from unscheduled map tasks are expected in the reduce phase. Thus, during the shuffle phase, Itchy only copies the map output from scheduled map tasks. For scheduled map tasks in map wave 2, Itchy reads, merges, and sorts all files containing relevant offsets. With this sorted set of offsets, the RecordReader of a map task can pass only relevant records to the map function. Notice that, as a map-call might emit several intermediate keys for one record, the map function can emit

intermediate keys that are not produced by the appended records. These intermediate keys are not required by reduce tasks to produce the correct results. In fact, these intermediate keys can actually lead to incorrect results for the such intermediate keys. Therefore, Itchy applies post-filtering of the not required intermediate keys at the end of each map task in map wave 2. As post-filtering is expensive, Itchy keeps track whether a MapReduce job on a specific dataset has *multiple emits* (i.e., if one record produces several intermediate keys). As a result, Itchy applies post-filtering only if the first job produces multiple emits. Then, the shuffle and reduce phase will progress as in the standard Hadoop MapReduce framework.

5.6.3 Itchy MO

Processing the first-job. In case of storing map outputs, Itchy has to persist the intermediate results while processing the first-job. The intermediate results are grouped and partitioned during the shuffle phase by default. Thus, the only difference from the standard Hadoop MapReduce workflow is that Itchy persists the intermediate results to HBase before starting each reduce task.

Processing the incremental-job. When processing the incremental-job, Itchy uses only the appended records as input. Itchy retrieves the required intermediate values just before each reduce-call. Notice that, each reduce-call is responsible for a single intermediate key. Therefore, Itchy requires only one HBase query to retrieve the intermediate values per intermediate key. Itchy appends the retrieved intermediate values to the intermediate values produced by the appended records. The reduce function processes all these intermediate values as in the standard Hadoop MapReduce workflow.

5.6.4 Itchy Merge

In contrast to Itchy QMC and Itchy MO, Itchy Merge requires users to specify a *merge function*. This is because the specification of such merge function requires some knowledge about the semantics of the MapReduce job. As the map and reduce functions are usually user-defined functions, these functions are black boxes to Itchy . Hence, the user-defined merge function signals Itchy (i) that a job is in the merge class and (b) how the base and incremental outputs can be merged. Users specify the merge function by using the `job.setMerge(class)` function when specifying their MapReduce jobs.

Processing the first-job. If Itchy encounters a job in the merge class, it will process the first-job almost as in the standard MapReduce framework. Except at the point where reduce tasks emit the output: Itchy additionally persists the QMC_{out} mappings to HBase. Recall, that QMC_{out} is a mapping from intermediate key to offset in the output.

Processing the incremental-job. When processing the incremental-job, Itchy again behaves as the standard Hadoop MapReduce workflow up to the point where reducers emit the final output. Here, Itchy retrieves the offsets in the base output using QMC_{out} for each intermediate key. Then, Itchy reads the corresponding parts from the base output at those offsets. Next, Itchy merges the retrieved parts of the base output with the just produced temporary output. Finally, Itchy writes the merged differential output to HDFS.

5.7 Experiments

We now evaluate whether Itchy can efficiently solve Alice's incremental data problem. We mainly perform this evaluation in order to answer the following four questions: (i) *What is the runtime for processing the incremental jobs?*, (ii) *What is the performance overhead when processing the first job?*, (iii) *What is the tradeoff between storing Itchy MO and Itchy QMC?*, and (iv) *How does Itchy deal with this tradeoff?*.

5.7.1 Experimental Setup

Hardware. We use a local 10 node cluster where each node has one 2.66GHz Quad Core Xeon processor running 64-bit platform Linux openSuse 11.1 OS, 4x4GB of main memory, 6x750GB SATA hard disks, and three Gigabit network cards. The advantage of running our experiments on this local cluster, rather than on the cloud, is that the amount of runtime variance is limited [161].

Datasets. We use two different datasets: 300GB of `Lineitem` data as specified in the TPC-H benchmark [82] and 300GB of `Wikipedia` dumb files as described by [53]. Notice that for the `lineitem` dataset, we use our own data generator as we wanted to vary certain characteristics, such as the cardinality of intermediate keys. We generate 300GB of base input and 1GB of incremental input for both datasets.

Systems. In our experiments, we use Hadoop [106] as the standard MapReduce system baseline and Incoop [94] as baseline for dealing with incremental data. We evaluate Itchy in all its three modes to operate, namely: Itchy QMC, Itchy MO, and Itchy Merge. Notice that, we use Itchy Merge only for the `Wordcount` job, because `Job1` and `Job2` are not in the Merge Class.

MapReduce Jobs. We use two different jobs for `lineitem`. The first job (`Job1`) is similar to Alice example query, while the second job (`Job2`) is based on the first query of TPC-H [82]. In other words, `Job1` and `Job2` implement the following SQL queries:

Job1 (in SQL)

```
SELECT l_partykey, AVG(l_extendedprice)
FROM  lineitem
GROUP BY l_partid
```

Job2 (in SQL)

```
SELECT l_returnflag,
l_linestatus, sum(l_quantity), sum(l_extendedprice),
sum(l_extendedprice*(1-l_discount)),
sum(l_extendedprice*(1-l_discount)*(1+l_tax)),
avg(l_quantity), avg(l_extendedprice),
avg(l_discount), count(*)
FROM lineitem
WHERE l_shipdate <= date '1998-12-01'
GROUP BY l_returnflag, l_linestatus
```

For the `Wikipedia` dataset, we use the `WordCount` job, which is frequently used for benchmarking MapReduce systems [106, 94]. Notice that, since Incoop requires the use of combine functions for its contraction phase, `Job1` and `Job2` then use a combine function as well (see Appendix 5.9). For all experiments, we run the jobs three times and report the average.

5.7.2 Upload Time

The first thing users have to do to run their MapReduce jobs is to upload their datasets to HDFS. In this respect, Itchy works in the same way as Hadoop, i.e., no special or extra process is required at upload time. In contrast, Incoop requires a special upload pipeline as it split the datasets in a content-wise manner. Thus, we benchmark the upload process for each of the three systems.

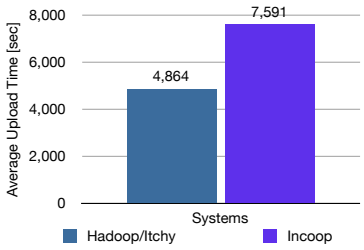
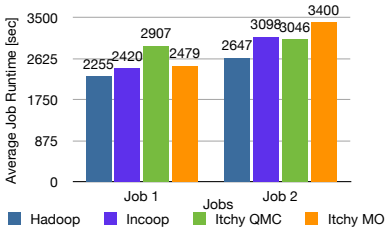
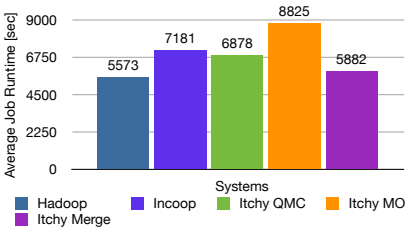


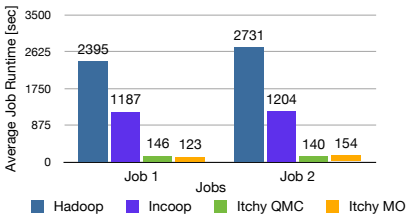
Figure 5.7: Uploading 300GB of Lineitem data.



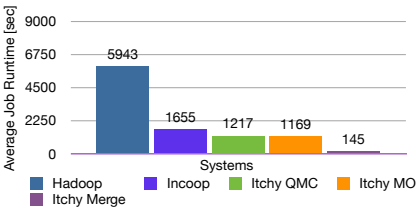
(a) 300GB of `Lineitem` data.



(b) 300GB of `WordCount` data.



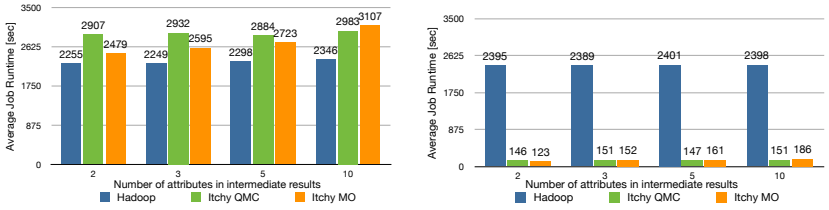
(c) 1GB of `Lineitem` data.



(d) 1GB of `WordCount` data.

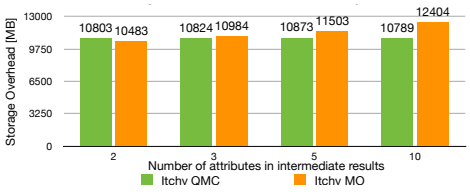
Figure 5.8: Job Runtimes: (a)-(b) execution times for running the first-job; (c)-(d) execution times for running the incremental-job.

Figure 5.7 illustrates the upload times for all three systems when uploading 300GB of `Lineitem` data. Notice that, we plot one bar for Hadoop and Itchy since Itchy performs the same process to as Hadoop to upload a dataset. In these results, we observe that both Hadoop and Itchy are ∼1.6 times faster than Incoop. This is because while Hadoop and Itchy performs a simple byte copy of the input dataset, Incoop have apply a content-wise splitting of data blocks. To do so, Incoop computes fingerprints to identify the boundaries between data blocks, which slow down the upload process.



(a) First-job.

(b) Incremental-job.



(c) Storage overhead w.r.t. Hadoop.

Figure 5.9: Tradeoff between QMC and MO while varying the size of intermediate values for Lineitem.

5.7.3 First-job Performance

Once their datasets are uploaded to HDFS, users can run their MapReduce jobs. When running MapReduce jobs for the first time, Itchy has to some store additional information: either QMC or MO information. In this section, we measure the overhead for storing such additional information.

Figure 5.8(a) illustrates the execution times for `Job1` and `Job2` when processing 300GB of `Lineitem` base input data. As expected, we observe that both Itchy and Incoop incur some overhead over the execution time of Hadoop as they have to store additional information. In particular, we observe that Itchy QMC has an overhead of ∼29% for `Job1` and and overhead of ∼15% `Job2`. Interestingly, we observe that Itchy QMC incurs a higher overhead than Incoop for `Job1`, but it has a lower overhead for `Job2`. This is because the intermediate data size for `Job1` is smaller than the QMC information, while the intermediate data size for `Job2` is bigger. This is why Itchy MO also has a lower overhead (∼10%) than Itchy QMC for `Job1` and a higher overhead (∼28%) for `Job2`. Here, we observe that Itchy MO has roughly the same performance as Incoop for `Job1`, but Itchy MO performs slightly slower than Incoop for `Job2`. This is because Itchy MO stores the intermediate data at the intermediate key level while Incoop does it at the task level.

Figure 5.8(b) shows the execution times for `WordCount` when processing 300GB

of `Wikipedia` base input data. Notice that, we also plot the results for Itchy Merge as `WordCount` allows for merging different outputs. On the one hand, we observe that Itchy QMC incurs an overhead of ∼23% over Hadoop, but Itchy QMC is slightly faster than Incoop. On the other hand, we observe that Itchy MO is ∼1.6 slower than Hadoop and ∼1.2 slower than Incoop. This is because the `WordCount` job produces a large number of intermediate keys[4]. In contrast, we observe that Itchy Merge performs 1.22 faster than Incoop, 1.13 faster than Itchy QMC, and 1.5 faster than Itchy MO. Still, Itchy Merge is slightly slower than Hadoop, because it has to store the mapping of intermediate keys to offsets in the output (QMC_{out}) to HBase.

5.7.4 Incremental-job Performance

We now focus on evaluating how well Itchy can help Alice with her problem, i.e., we evaluate the performance of Itchy when performing incremental-jobs. For this, we consider that the `Lineitem` and `Wikipedia` datasets have been appended 1GB new data. Note, that for Itchy and Hadoop we only have to upload the 1GB of new records, while for Incoop we have to upload a 301GB dataset including all input from the first-job. This is because, Incoop can currently handle only a single file as input to a job while Itchy and Hadoop can use several files as input to a job.

Figure 5.8(c) shows the results for `Lineitem`. We observe that Hadoop performs quite similar as when running the first-job. This is because Hadoop has to reprocess the entire 300GB base input in addition to the 1GB of appended records. In contrast, Itchy does not have to recompute the entire 300GB base input. As a result, we observe that Itchy is up to ∼ 20 times faster than Hadoop and ∼10 times faster than Incoop. In particular, we observe that the performance gap between Itchy and both Hadoop and Incoop is higher for `Job2`. This is because the intermediate data produced by `Job2` has a larger size. This is why Itchy QMC also slightly outperforms Itchy MO for `Job2`. Moreover, we also observe during our experiments that Itchy is close to the performance of Hadoop when performing only 1GB incremental input (e.g., Itchy MO incurs an overhead of only ∼12%). This shows the high efficiency of Itchy to deal with incremental data.

Figure 5.8(d) illustrates the results for `WordCount` when processing 1GB of new `Wikipedia` data. In these results, we plot the results for Itchy Merge as

[4]One could apply dictionary compression to reduce the intermediate data size, but this is out of the scope of this thesis.

`WordCount` is in the Merge class. Again, Hadoop reprocess the entire 300GB input to the first-job in addition to the 1GB appended records. As a result, Hadoop has a similar performance as when performing the first-job. For Itchy, we observe that Itchy QMC and Itchy MO are ∼4.6 times faster than Hadoop and ∼1.3 times faster than Incoop. However, recall, that Incoop requires the user to upload the entire 301GB of data while Itchy only requires the user to upload the 1GB of new records. Here, the reader can observe that the relative performance difference between Incoop and Itchy QMC and Itchy MO is not as large as in the `Lineitem` benchmark. This is because the `WordCount` incremental-job requires many intermediate values to be recomputed as a single frequent word can occur in a large number of documents. In contrast, Itchy Merge exploits the fact that `WordCount` performs a simple sum of frequencies per word to simply add the incremental output with the base output. This results in an improvement factor of ∼41 over Hadoop and of ∼11 over Incoop.

5.7.5 Varying intermediate Value Size

In the previous two Sections, we show that high superiority of Itchy over Hadoop and Incoop. From now, we study the tradeoff between Itchy QMC and Itchy MO, using only Hadoop as baseline. Recall from Section 5.4.4 that Itchy QMC requires a higher cost for recomputing some records from the first-job, while Itchy MO might require a lot of space for storing the intermediate values. To examine this tradeoff in more detail, we first look at runtime overhead when processing the first-job while varying the intermediate value size. For this, we alter `Job1` to include additional attributes in the intermediate values.

Figure 5.9(a) shows the results for the first-job when varying the intermediate value size. We observe that the job runtime for both Hadoop and Itchy QMC does not increase by much when increasing the intermediate value size. The increase in runtime from the job projecting 2 to the job projecting 10 attributes in the intermediate values is of 4% for Hadoop and of 3% for Itchy QMC. In contrast, Itchy MO incurs a 25% increase in runtime from projecting 2 attributes to projecting 10 attributes. This is because Itchy MO cost for storing the MO mappings strongly depends on the of intermediate values, while Itchy QMC cost for storing the QMC mappings is independent of the intermediate value size.

Figure 5.9(b) shows the results for the incremental-job when varying the intermediate value size. In contrast to previous results, we observe that the job runtime for all three systems in consideration does not vary much. This is because the number of intermediate values is far less than for the first-job. As a result, the additional effort for storing larger intermediate values is hidden by the overhead of Hadoop framework. Overall, we see that Itchy MO performs about 4% faster than Itchy QMC, because it does not have to recompute the intermediate values but it can retrieve them directly.

5.7.6 Storage Overhead

After looking at the runtime characteristics between Itchy QMC and Itchy MO, we now consider the other side of the tradeoff, i.e., the storage overhead. Recall that both systems have to store additional information: while Itchy QMC stores QMC mappings, Itchy MO stores the intermediate results. Here, we consider as storage overhead the size of the entire HBase table storing either the QMC or MO mappings.

Figure 5.9(c) illustrates the storage overhead incurred by both systems when varying again the size of intermediate values. As expected, we observe that the overhead incurred by Itchy QMC does not vary with respect to intermediate value size. This is because Itchy QMC stores offsets and hence the QMC storage does only depend on the number of intermediate keys and not on the size of intermediate values. In contrast, Itchy MO stores the intermediate values directly and hence we can see the storage size for MO increase when including additional attributes in the intermediate values. Notice that, for 2 attributes the storage size for Itchy QMC is actually larger than for Itchy MO. This is due the fact that Itchy QMC has to store the filename identifier and offset which is larger than the size of the two attributes Itchy MO has to store as intermediate value. However, as soon as `Job1` start to project more attributes, the overhead incurred by Itchy MO starts getting larger than Itchy QMC.

5.7.7 Effectiveness of Itchy's Decision Model

Let us now consider how well the Itchy's decision model (see Section 5.4.4) deals with the different benchmarks. Table 5.2 shows sampled values for the average runtime to process a record in the map function (*avgRuntime*), the average size for an intermediate value (*avgValSize*), and the average map selectivity (*avgSelectivity*) together with the output of the decision model for each benchmark.

Job	avgRuntime [msec]	avgValSize [B]	avg Selectivity	Decision
`Job1`	0.008	16	1	MO
`Job2`	0.016	113	0.83	QMC
`WordCount`	0.021	18	9	Merge

Table 5.2: Decision Model for Benchmark Queries

Recall that these values are sampled while running the first wave of map tasks in the first-job. We observe that in `Job1` the average intermediate value size (consisting of two long values) is as large as the size for storing a QMC mapping (also two long values). Therefore, the decision model decides to use Itchy MO as for the same storage requirements it expects a better performance for the incremental-job. As we can see from Figure 5.8(c) this assumption holds true. For `Job2`, we see that the average intermediate value size is much larger as it contains more and larger attributes. In such case, the decision model considers the ratio $\frac{avgValSize}{avgRuntime}$, Since this ratio is above the given threshold, Itchy decides to use the Itchy QMC technique. Again, we see in Figure 5.8(c) that this holds true. For the `WordCount` job, Itchy has the option to execute the user defined merge function, Thus, Itchy decides for using Itchy Merge. As we can see in Table 5.2, both the average map function runtime and required storage size[5] are not favorable for Itchy QMC and Itchy MO. The results presented in Figure 5.8(d) confirms that this decision was once more again the right decisions to take.

5.8 Related Work

MapReduce was originally proposed by Google for building their inverted search index [106]. However, Google faced problems with the incremental nature of their web index and hence moved on to Percolator [151]. Percolator operates on top of BigTable [97] and has departed from the idea of MapReduce. Most other works, such as Incoop [94] and DryadInc [152], propose the idea of memoization. Here, the idea is to cache outputs of map and reduce tasks. These cached intermediate results can be returned when the same task is later run over the same input. However, these two approaches has to read the entire dataset every time the dataset changes in order to identify the parts

[5] Even if the average intermediate value size is not very high, the selectivity is such that each input record (line of text) produces an average of nine intermediate values

of the dataset that changed. DryadINC additionally proposes the idea of *Mergable Computation* similar to idea of Itchy Merge. However, in contrast to DryadINC can merge the output stored in standard HDFS.

Another approach is Restore [111], which considers workflows of several MapReduce jobs as produced by high-level languages, such as Pig [117]. Restore persists the outputs of individual jobs in such a workflow and can reuse these outputs for later jobs in cases where the physical plans match. However, a previous plan (and therefore its stored output) is only considered a match for a new plan if both input datasets are unchanged. Hence, it is not applicable to Alice's problem of growing datasets. Ramp [126] considers the related problem of *selective refresh*. Here, the focus is on a particular output record. In other words, given some changed input data, how can we update this single output tuple efficiently? Itchy focuses on a different problem: *given a set of appended records, how can we update the entire result set?*. [125] also explores the concepts of provenance in MapReduce similarly to Itchy and measure the overhead for capturing provenance. However they do not apply the concept to the idea of incremental computation. CBP [137] implements incremental bulk processing on top of MapReduce by considering stateful grouping operators. Nevertheless, users has to decide and specify which state should be persisted and used.

One can view the result of a MapReduce job as a materialized view on the input data. Therefore, we also consider the literature on relational incremental materialized view maintenance [86, 83]. Here, the problem is slightly different: usually the materialized views are specified by declarative SQL queries. Hence, the semantics of queries are known and used for view maintenance. This is not the case for MapReduce, where queries are usually specified by black box map and reduce functions. Still, some of the ideas of relational incremental materialized view maintenance could be applied when working with high-level query languages on top of MapReduce such as HiveQL [167] or PigLatin [117].

5.9 Combiners

In practice, users can specify a *combine* function for some of their MapReduce jobs in order to reduce the amount of intermediate data to shuffle. Typically, users use the reduce function itself as a combine function. For example, one can do so for MapReduce jobs computing a *sum* or a *count* aggregation. However, in many other cases, the combine function is not as straight forward. For instance,

for our `JobAvg` example in Section 5.1, a combine function has to keep track of the count of records in addition to the sum of all price values. In some other cases, it is even impossible to specify a combine function, e.g., for a reduce side join [170]. In the following, we describe how Itchy supports combine functions for the different techniques: Itchy QMC, Itchy MO, and Itchy Merge.

Itchy QMC. When using combine functions with Itchy QMC, the combine function has to ensure that all offsets of intermediate values in the input to this combiner are kept. Itchy QMC keeps a list of all offsets in addition to the intermediate data and persist this list of offsets to HBase in the reducer.

Itchy MO. Here, we have two different options to exploit combiners: either we (i) store the intermediate values directly to HBase or (ii) keep a list of all intermediate values. While the first option requires a combine function to perform a higher number of HBase write operations, the second option requires a combine function to simply appends the different intermediate values. Therefore, Itchy uses the second option as the standard way to support combiners in Itchy MO. Figure 5.10 shows that, for `JobAVG`, the second option indeed performs better than the first option.

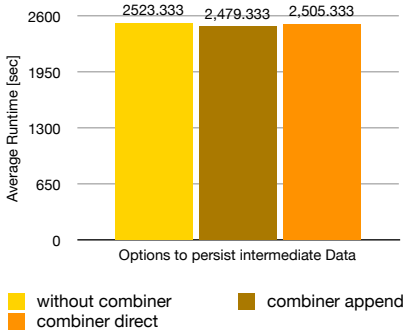


Figure 5.10: Itchy MO processing 300GB for JobAVG

Itchy MERGE. Itchy Merge does not require any changes for supporting combiners, as Itchy Merge works with the final output which does not change when using combiners.

5.10 Conclusion

Many current applications produce large amounts of data in a daily basis that has to be analysed as new data arrive. Thus, analytical tasks have to deal with incremental datasets in an efficient manner. We presented Itchy (*Incremental TeCHniques for Yellow elephants*) for supporting MapReduce jobs over growing datasets. As we identified that different MapReduce jobs require different approaches to deal with incremental datasets, Itchy contains a suite of three different techniques: Itchy QMC, Itchy MO, and Itchy Merge. Itchy QMC maintains a mapping from intermediate keys to input records while running a MapReduce job for first time over a given dataset (base input). If new records arrive, Itchy QMC then uses this mapping to selectively recompute records from the base input in order to produce the right output when running consecutive MapReduce jobs. Itchy MO avoids recomputing records from the base input (which can be costly) by storing a different kind of mapping. Itchy MO stores a map from intermediate keys to intermediate values directly. However, we showed that this advantage is outweighed by large intermediate values. Therefore, we presented a cost-based decision model that allows Itchy to automatically chooses the right technique between Itchy QMC and Itchy MO for each incoming MapReduce job. In contrast to Itchy QMC and Itchy MO, Itchy Merge merges whenever it is possible the output two MapReduce jobs using a user-defined merge function. On of the beauties of Itchy is that it chooses the right technique to use in an almost invisible ways for users. Hence, Itchy does not require users to change their MapReduce jobs.

We evaluated Itchy using two different benchmarks and compared it with Hadoop (the most popular MapReduce framework) and Incoop (a state-of-the-art system to deal with incremental datasets). The results showed the superiority of Itchy over both Hadoop and Incoop: Itchy runs up to ∼20 times faster than Hadoop and up to ∼10 times faster than Incoop. In cases where users can specify a merge function, Itchy can improve the performance up to a factor of ∼41 compared to Hadoop up to a factor of ∼11 compared to Incoop. Still, in terms of data upload, Itchy performs exactly as Hadoop and up to 1.6 times faster than Incoop. A series of additional experiments also showed the high effectiveness of the Itchy decision model.

6 Conclusion

In this thesis we considered two important topics in todays IT landscape: Cloud Computing and Big Data Analyses. Especially the combination of both techniques allows a large group of users to solve previously impossible big data problems, but also yields new problems. In this thesis we showed that performance variance between different nodes in an IaaS Cloud Setting is a severe problem. IaaS cloud providers such as Amazon EC2, Microsoft Azure, or Google offer instance types with different performance. But, when looking at the performance among equal instances (i.e., having the same instance types) in Chapter 3 we experienced large performance variance between them. Using micro-benchmarks we measured the Coefficient of Variance (COV) for CPU, IO Read, IO Write, Memory, and Network performance. We identified significant performance variance where the COVs ranged from 0.15 for IO Write performance to 0.35 for CPU performance. Such high performance variance makes it very difficult for users to repeat experiments measuring performance or predict the performance of their given IaaS cloud instance.

Next, we identified the underlying hardware for the same virtual instance type had huge impact on performance in Section 3.5.2. When considering the performance variance per system type of underlying hardware we could reduce the COV from 35% to 1% to 2%. We saw similar reductions in performance variance for memory and even disk performance. These results how important it is for users to not blindly rely on the cloud notion of virtual instance, but where possible check for the underlying physical hardware.

Even considering the underlying physical system, IaaS virtual instances incur a high performance variance compared to our baseline measurements on a physical system. We confirmed this by measuring the performance for a single long running instance over a time period of one month. We saw very low CPU variance, but a very high variance for IO performance (especially for the Azure cloud). This remaining variance can be understood when considering that in an IaaS setting usually several virtual instances are executed on the

same physical hardware. Resource isolation, i.e., the illusion that each virtual machines owns the physical hardware is an important goal for the hypervisors managing the virtual machines. But this is more or less challenging for different hardware components. While resource isolation for CPU performance (excluding the caches) is quite simple, the isolation for IO is more challenging. The actual isolation properties differ between different hypervisors (for example between Amazon EC2 and Microsoft Azure), unfortunately as we did not have direct access to the underlying physical hardware we could not further evaluate such effects by for example measuring the cache miss rates.

As we first published these finding in 2010, we wondered whether the situation had changed in 2013. By then IaaS cloud computing had become a valuable tool for many users and also the cloud vendors included other well known companies such as Google and Amazon. The result show that performance variance is still at he same level as in 2010 or even slightly above. For Amazon EC2 we saw a growing variety of underlying system types (four as compared to only two in 2010). The growing physical system diversity is logical considering their growing number of new data centres while older hardware is still used for a certain time period.

Next, we examined whether the performance variance we measured would effect to application performance, especially Big Data Analytics using MapReduce. Here we saw that the cluster heterogeneity (i.e., how different are nodes in a cluster with nodes having the same instance type) has a significant impact on MapReduce job performance. As MapReduce—similar to many other distributed systems—makes the assumption of homogeneous node performance, we saw a performance degradation for increasing performance difference between virtual nodes in the cluster. We proposed three different techniques for reducing the cluster heterogeneity of a cluster by shortly allocating more nodes than required and then choose an optimal subset of nodes for the actual job. Here we saw that considering simply the system type achieves a similar reduction variance compared to the more complex measuring of the cluster heterogeneity directly which involves running several benchmarks on each of the cluster nodes.

As cloud computing also has different offers besides IaaS, we also the performance variance for Amazon's Elastic Map Reduce (EMR) Platform as a Service (Paas) offer. Here we saw that on our EMR cluster we achieved a

similar performance variance consider job runtimes than on a similar cluster in an IaaS setting.

Next, in Section 5 we considered another problem for large scale data analytics: dealing with incremental datasets in an efficient manner. Many current applications produce large amounts of data in a daily basis that has to be analysed as new data arrive. We proposed ITCHY (*Incremental TeCHniques for Yellow elephants*) for supporting MapReduce jobs over growing datasets including three different approached: Itchy QMC, Itchy MO, and Itchy Merge. We presented a cost-based decision model that allows Itchy to automatically chooses the right technique between Itchy QMC and Itchy MO for each incoming MapReduce job. In contrast to Itchy QMC and Itchy MO, Itchy Merge merges whenever it is possible the output two MapReduce jobs using a user-defined merge function. Hence, Itchy does not require users to change their MapReduce jobs. Our results showed the superiority of Itchy over both Hadoop and Incoop(a state-of-the-art system to deal with incremental datasets): Itchy runs up to $\sim$20 times faster than Hadoop and up to $\sim$10 times faster than Incoop. In cases where users can specify a merge function, Itchy can improve the performance up to a factor of $\sim$41 compared to Hadoop up to a factor of $\sim$11 compared to Incoop. Still, in terms of data upload, Itchy performs exactly as Hadoop and up to 1.6 times faster than Incoop.

6.1 Outlook

With the increasing relevance of cloud computing also understanding the effects of performance variance becomes more and more important. As we have seen with Amazon EC2, the number of underlying physical systems increases over the lifetime of cloud vendors due as new physical systems are introduced. As we have seen this effects the performance of the IaaS virtual instances and hence customer experience.
Another interesting development is the growing shift towards container based virtualization [164] (please refer to Section 2.2.3 for details). Amazon EC2 Container Service [15] and Google Container Engine [43] are already offering such container based virtualization Cloud infrastructures. As with container based virtualization we have even less performance isolation compared to traditional virtual machines, it will become important for distributed applications to drop the assumption of homogeneous performance across all cluster nodes. The good news is the growing ecosystem around such container virtualization

includes Operating Systems such as CoreOs [33] and special scheduler such as Apache Mesos [9]. As these system have the control of the Operating System in case of CoreOs or the Cluster in case of Mesos, they can already consider performance interference in their scheduling decisions.

Even though continuing popularity of Big Data Analytics the traditional MapReduce is rapidly overtaken by more flexible processing models such as Apache Spark [10], Apache Flink [7], or Apache Tez [12]. This trend is also reflected by Hadoop 2.0 [44] supporting a number of different processing engines besides the traditional MapReduce engine. Use cases for big data analytics such as shift from read only datasets towards more dynamic datasets. This trend is also reflected by the number of streaming system such as Spark Streaming [74] or Apache Storm [11]. Such highly dynamic datasets make incremental processing even more important.

Bibliography

[1] 3tera. http://www.3tera.com. retrieved: 16.07.2013.

[2] Amazon AWS. http://aws.amazon.com. retrieved: 16.07.2013.

[3] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2. retrieved: 02.07.2014.

[4] Amazon Forum. http://developer.amazonwebservices.com/connect/thread.jspa?threadID=16912. retrieved: 16.07.2013.

[5] AMD-V. www.amd.com/virtualization. retrieved: 15.07.2013.

[6] Apache Deltacloud. http://deltacloud.apache.org. retrieved: 15.07.2013.

[7] Apache Flink. https://flink.apache.org/. retrieved: 12.03.2015.

[8] Apache HBase. http://hbase.apache.org/. retrieved: 15.07.2013.

[9] Apache Mesos. http://mesos.apache.org/. retrieved: 14.07.2014.

[10] Apache Spark. https://spark.apache.org/. retrieved: 12.03.2015.

[11] Apache Storm. https://storm.apache.org/. retrieved: 20.04.2015.

[12] Apache Tez. https://tez.apache.org/. retrieved: 12.03.2015.

[13] Apache Whirr. http://whirr.apache.org. retrieved: 15.07.2013.

[14] AppNexus. http://www.appnexus.com. retrieved: 16.07.2013.

[15] AWS EC2 Container Service. http://aws.amazon.com/ecs/. retrieved: 20.04.2015.

[16] Aws Ec2 Dedicated Instances. http://aws.amazon.com/de/dedicated-instances. retrieved: 23.07.2013.

[17] Aws Ec2 GovCloud. http://aws.amazon.com/de/govcloud-us. retrieved: 23.07.2013.

[18] AWS Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/. retrieved: 14.03.2014.

[19] AWS Micro Instance Type. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html. retrieved: 20.12.2011.

[20] AWS Regions. http://aws.amazon.com/about-aws/globalinfrastructure/. retrieved: 20.09.2013.

[21] Aws Virtual Private Cloud. http://aws.amazon.com/de/vpc. retrieved: 23.07.2013.

[22] Azure Regions. http://joranmarkx.wordpress.com/2012/01/16/microsoft-azure-data-center-locations-worl-wide/. retrieved: 20.09.2013.

[23] Bonnie. http://www.textuality.com/bonnie/intro.html. retrieved: 03.08.2009.

[24] Bundesdatenschutzgesetz. http://www.gesetze-im-internet.de/bdsg_1990. retrieved: 15.07.2013.

[25] Citrix Xen Server. www.xensource.com. retrieved: 15.07.2013.

[26] Cloud Harmony. http://cloudharmony.com. retrieved: 10.03.2014.

[27] Cloud Spectator. http://www.cloudspectator.com. retrieved: 10.03.2014.

[28] Cloud Suite 2.0. http://parsa.epfl.ch/cloudsuite/cloudsuite.html. retrieved: 14.03.2014.

[29] CloudClimate. http://www.cloudclimate.com. retrieved: 20.12.2011.

[30] CloudKick. https://www.cloudkick.com. retrieved: 20.12.2011.

[31] Container at Google. http://googlecloudplatform.blogspot.de/2014/06/an-update-on-container-support-on-google-cloud-platform.html. retrieved: 14.07.2014.

[32] Container at Google App Engine. https://developers.google.com/compute/docs/. retrieved: 30.07.201.

[33] coreOS. http://linux-vserver.org/. retrieved: 14.07.2014.

[34] Coursera. https://www.coursera.org. retrieved: 15.07.2013.

[35] Docker. http://docker.com/. retrieved: 14.07.2014.

[36] EPIC Patriot Act. http://epic.org/privacy/terrorism/usapatriot. retrieved: 15.07.2013.

[37] Facebook Developers. http://developers.facebook.com. retrieved: 16.07.2013.

[38] G-Cloud. http://gcloud.civilservice.gov.uk. retrieved: 20.12.2011.

[39] Google App Engine. https://appengine.google.com. retrieved: 16.08.2013.

[40] Google App Engine. https://cloud.google.com/appengine. retrieved: 15.07.2013.

[41] Google Apps. http://www.google.com/apps. retrieved: 16.07.2013.

[42] Google Compute Engine. https://cloud.google.com/products/compute-engine. retrieved: 15.07.2013.

[43] Google Container Engine. https://cloud.google.com/container-engine. retrieved: 20.04.2015.

[44] Hadoop. http://hadoop.apache.org. retrieved: 02.07.2014.

[45] Hadoop MapReduce. http://hadoop.apache.org/mapreduce. retrieved: 14.03.2014.

[46] IBM Blue Cloud. http://www-03.ibm.com/systems/deepcomputing/cod/index.html. retrieved: 16.07.2013.

[47] Intel VT-x. www.intel.com/go/virtualization. retrieved: 15.07.2013.

[48] Iperf. http://iperf.sourceforge.net. retrieved: 03.08.2009.

[49] Kernel Based Virtual Machine. http://www.linux-kvm.org. retrieved: 15.07.2013.

[50] libcloud. http://libcloud.apache.org/. retrieved: 20.12.2011.

[51] Linux Containers. https://linuxcontainers.org. retrieved: 14.07.2014.

[52] Linux-VServer. https://developers.google.com/compute/docs/containers. retrieved: 14.07.2014.

[53] MapReduce Benchmarks. http://web.ics.purdue.edu/~fahmad/benchmarks.htm. retrieved: 02.07.2014.

[54] Microsoft HyperV. http://www.microsoft.com/en-us/server-cloud/hyper-v-server. retrieved: 15.07.2013.

[55] MRCloud. http://infosys.cs.uni-saarland.de/MRCloud.php. retrieved: 20.12.2011.

[56] MS Azure. http://www.microsoft.com/enterprise/it-trends/cloud-computing. retrieved: 15.07.2013.

[57] MS Office 365. http://office.microsoft.com. retrieved: 15.07.2013.

[58] MS Windows Azure. http://www.microsoft.com/windowsazure. retrieved: 15.07.2012.

[59] Netflix Steal Time. https://support.cloud.engineyard.com/entries/22806937-Explanation-of-Steal-Time. retrieved: 16.07.2014.

[60] NIST. http://csrc.nist.gov/groups/SNS/cloud-computing/index.html. retrieved: 20.12.2011.

[61] Opencompute Initiative. www.opencompute.org. retrieved: 12.03.14.

[62] Openstack. www.openstack.org/. retrieved: 15.07.2013.

[63] OpenVZ. http://openvz.org. retrieved: 14.07.2014.

[64] Oracle VirtualBox. https://www.virtualbox.org. retrieved: 15.07.2013.

[65] Oracle VM Server. http://www.oracle.com/us/technologies/virtualization/index.html. retrieved: 15.07.2013.

[66] Patriot Act. http://www.fincen.gov/statutes_regs/patriot. retrieved: 15.07.2013.

[67] Project Voldemort. http://project-voldemort.com. retrieved: 02.07.2014.

[68] Rackspace. http://www.rackspacecloud.com. retrieved: 16.07.2013.

[69] Safe-Harbor-Agreement. http://export.gov/safeharbor/eu. retrieved: 15.07.2013.

[70] SalesForce. http://www.salesforce.com. retrieved: 16.07.2013.

[71] SAP On-Demand. http://www.sap.com/solutions/business-suite/crm/crmondemand/index.epx. retrieved: 16.07.2013.

[72] SimpleCloud. http://www.simplecloud.org. retrieved: 15.07.2013.

[73] Solaris Zones. http://www.oracle.com/technetwork/server-storage/solaris11/technologies/virtualization-306056.html. retrieved: 14.07.2014.

[74] Spark Streaming. https://spark.apache.org/streaming. retrieved: 20.04.2015.

[75] Sun Cloud. http://sun.com/cloud. retrieved: 03.08.2009.

[76] Ubench. http://phystech.com/download/ubench.html. retrieved: 03.08.2009.

[77] Unixbench. https://code.google.com/p/byte-unixbench. retrieved: 05.08.2013.

[78] VmWare ESX. http://www.vmware.com/products/vsphere. retrieved: 15.07.2013.

[79] VmWare Workstation. www.vmware.com/Workstation. retrieved: 15.07.2013.

[80] Xen IO handling. http://www.mulix.org/lectures/xen-iommu/xen-io.pdf. retrieved: 15.07.2013.

[81] Yahoo Cloud Serving Benchmark. http://labs.yahoo.com/news/yahoo-cloud-serving-benchmark/cloudsuite.html. retrieved: 14.03.2014.

[82] TPC-H Benchmark Specification 2.14.4. http://www.tpc.org/tpch/, 2012. retrieved: 15.07.2013.

[83] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. *Proceesings of the 24rd International Conference on Very Large Data Bases*, 1998.

[84] A. Abouzeid, K. Bajda-Pawlikowski, D. Abajdi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[85] O. Acıiçmez, B. B. Brumley, and P. Grabher. New results on Instruction Cache Attacks. *Proceedings for the Cryptographic Hardware and Embedded Systems Conference*, pages 110–124, 2010.

[86] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. *ACM SIGMOD Record*, 26(2):417–427, 1997.

[87] C. Aguilar-Melchor, S. Fau, C. Fontaine, G. Gogniat, and R. Sirdey. Recent Advances in Homomorphic Encryption: A Possible Future for Signal Processing in the Encrypted Domain. *Signal Processing Magazine, IEEE*, 30(2):108–117, 2013.

[88] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar. Tarazu: Optimizing MapReduce on heterogeneous Clusters. *Proceedings of the 17th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–74, 2012.

[89] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. *EECS Department, UCB, Tech. Rep. UCB/EECS-2009-28*, 2009.

[90] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[91] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. V. Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. *Proceedings of the 2006 Ottawa Linux Symposium*, pages 71–86, 2006.

[92] S. Benedict. Performance Issues and Performance Analysis Tools for HPC Cloud Applications: A Survey. *Computing*, 95:89–108, 2013.

[93] D. J. Bernstein. Technical report: Cache-Timing Attacks on AES. `http://www.hamidreza-mz.tk/files/cachetiming-20050414.pdf`, 2005. retrieved: 23.07.2013.

[94] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, and R. Pasquin. Incoop: Mapreduce for Incremental Computations. *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 7, 2011.

[95] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. *Proceedings of the Second International Workshop on Testing Database Systems*, page 9, 2009.

[96] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. *Procceedings of the 13th International Workshop on the Web and Databases*, page 10, 2010.

[97] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, 2008.

[98] Y. Chen, V. Paxson, and R. H. Katz. What is new about Cloud Computing Security. *University of California, Berkeley Report No. UCB/EECS-2010-5 January*, 20(2010):2010–5, 2010.

[99] D. Chisnall. *The Definitive Guide to the Xen Hypervisor*. Pearson Education, 2013.

[100] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[101] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. Mad Skills: New Analysis Practices for Big Data. *Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009.

[102] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 10(4):20, 2010.

[103] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[104] J.-D. Cryans, A. April, and A. Abran. Criteria to Compare Cloud Computing with Current Database Technology. *Proceedings for the Conference on Software Process and Product Measurement*, pages 114–126, 2008.

[105] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[106] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *Communications of the ACM*, 53(1):72–77, 2010.

[107] J. Dejun, G. Pierre, and C.-H. Chi. EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications. *ServiceWave Workshop for Service-Oriented Computing*, pages 197–207, 2009.

[108] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proceedings of the VLDB Endowment*, 3(1):518–529, 2010.

[109] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only Aggressive Elephants are Fast Elephants. *Proceedings of the VLDB Endowment*, 5(11):1591–1602, 2012.

[110] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs Containerization to Support PaaS. *Proceedings of the IEEE International Conference on Cloud Engineering*, pages 610–614, 2014.

[111] I. Elghandour and A. Aboulnaga. ReStore: Reusing Results of MapReduce Jobs in Pig. *Proceedings of the VLDB Endowment*, 5(6):701–704, 2012.

[112] R. Esteves, R. Pais, and C. Rong. K-means Clustering in the Cloud: A Mahout Test. *Advanced Information Networking and Applications*, pages 514–519, 2011.

[113] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: a Study of emerging scale-out Workloads on modern Hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–47, 2012.

[114] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-Oriented Storage Techniques for MapReduce. *Proceedings of the VLDB Endowment*, 4(7):419–429, 2011.

[115] E. Frachtenberg, A. Heydari, H. Li, A. Michael, J. Na, A. Nisbet, and P. Sarti. High-Efficiency Server Design. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 27, 2011.

[116] S. Garfinkel. An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. Technical Report TR-08-07, Harvard University, July 2007.

[117] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[118] C. Gentry, A. Sahai, and B. Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. *Advances in Cryptology*, pages 75–92, 2013.

[119] J. Goodacre and A. N. Sloss. Parallelism and the ARM Instruction set Architecture. *IEEE Computer*, 38(7):42–50, 2005.

[120] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[121] B. Gregg. *Systems Performance: Enterprise and the Cloud*. Pearson Education, 2013.

[122] S. Gupta, C. Fritz, B. Price, R. Hoover, J. de Kleer, and C. Witteveen. ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters. *Proceedings 10th ACM International Conference on Autonomic Computing*.

[123] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing Database as a Service. *Proceedings of the 18th International Conference on Data Engineering*, pages 29–38, 2002.

[124] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[125] R. Ikeda, H. Park, and J. Widom. Provenance for Generalized Map and Reduce Workflows. *Proceedings of Conference on Innovative Data Systems Research*, 2011.

[126] R. Ikeda, S. Salihoglu, and J. Widom. Provenance-based refresh in data-oriented workflows. Technical Report 962, Stanford University, 2011.

[127] T. S. S. O. ITU. Focus Group on Cloud Computing Technical Report. Technical report, ITU, 2012.

[128] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011.

[129] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley- Interscience, 1991.

[130] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *Proceedings of the VLDB Endowment*, 3(1):472–483, 2010.

[131] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 21, 2011.

[132] A. Jindal, F. M. Schuhknecht, J. Dittrich, K. Khachatryan, and A. Bunte. How Achaeans Would Construct Columns in Troy. *Proceedings of Conference on Innovative Data Systems Research*, 13:6–9, 2013.

[133] M. Johnson, H. McCraw, S. Moore, P. Mucci, J. Nelson, D. Terpstra, V. Weaver, and T. Mohan. PAPI-V: Performance Monitoring for Virtual Machines. *Proceedings of the 41st International Conference on Parallel Processing Workshops*, pages 194–199.

[134] D. Kossmann, T. Kraska, and S. Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 579–590, 2010.

[135] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's Inside the Cloud? An Architectural Map of the Cloud Landscape. *Proceedings of the ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31, 2009.

[136] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai. Performance model driven QoS guarantees and optimization in clouds. *Proceedings of the ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 15–22, 2009.

[137] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Stateful Bulk Processing for Incremental Analytics. *Proceedings of the 1st ACM symposium on Cloud computing*, pages 51–62, 2010.

[138] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-Situ MapReduce for Log Processing. *Proceedings of USENIX Annual Technical Conference*, page 115, 2011.

[139] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, and E. Ozer. Scale-out Processors. In *ACM Special Interest Group on Computer Architecture Computer Architecture News*, volume 40, pages 500–511. IEEE Computer Society, 2012.

[140] M. Mani. Enabling Secure Query Processing in the Cloud using Fully Homomorphic Encryption. *Proceedings of the Second Workshop on Data Analytics in the Cloud*, pages 36–40, 2013.

[141] V. G. Martínez, L. H. Encinas, and A. M. Muñoz. A Comparative Analysis of Hybrid Encryption Schemes Based on Elliptic Curves. *Open Mathematics Journal*, 6:1–8, 2013.

[142] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the Performance Isolation Properties of Virtualization Systems. *Proceedings of the 2007 Workshop on Experimental Computer Science*, page 6, 2007.

[143] A. Menon. Big data@ facebook. *Proceedings of the 2012 Workshop on Management of Big Data Systems*, pages 31–32, 2012.

[144] K. Morton and A. Friesen. KAMD: A Progress Estimator for MapReduce Pipelines. *IEEE 26th International Conference on Data Engineering*, 2010.

[145] R. Nikolaev and G. Back. Perfctr-Xen: A Framework for Performance Counter Virtualization. *ACM SIGPLAN Notices*, 46(7):15–26, 2011.

[146] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, 2009.

[147] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. *Proceedings of the 2008 ACM SIGMOD international Conference on Management of Data*, pages 1099–1110, 2008.

[148] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in Virtual Machine Monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[149] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. *Proceedings of the International Conference on Cloud Computing*, pages 115–131, 2009.

[150] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, 2009.

[151] D. Peng and F. Dabek. Large-Scale Incremental Processing Using Distributed Transactions and Notifications. *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[152] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing Work in Large-Scale Computations. *Proceedings of the Conference on Hot topics in Cloud Computing*, 2009.

[153] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[154] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the Art of Virtualization. *Linux Symposium*, page 65, 2005.

[155] J.-A. Quiané-Ruiz, C. Pinkel, J. Schad, and J. Dittrich. Raft at work: speeding-up mapreduce applications under task and node failures. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 1225–1228, 2011.

[156] J.-A. Quiane-Ruiz, C. Pinkel, J. Schad, and J. Dittrich. RAFTing MapReduce: Fast Recovery on the Raft. *2011 IEEE 27th International Conference on Data Engineering*, 2011.

[157] H. Raj, R. Nathuji, A. Singh, and P. England. Resource Management for isolation enhanced Cloud Services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[158] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[159] R. Rose. Survey of System Virtualization Techniques. *Technical Report*, 2004.

[160] J. Schad. Flying Yellow Elephant: Predictable and Efficient MapReduce in the Cloud. *PhD Workshop VLDB 2010*, 2010.

[161] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proceedings of the VLDB Endowment*, 3(1):460–471, 2010.

[162] J. Schad, J.-A. Quiané-Ruiz, and J. Dittrich. Elephant, Do not Forget Everything! Efficient Processing of Growing Datasets. *IEEE Sixth International Conference on Cloud Computing*, 2013.

[163] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[164] D. Strauss. The Future Cloud is Container, not virtual Machines. *Linux Journal*, 2013(228):5, 2013.

[165] S. Subramanian, Y. Zhang, R. Vaidyanathan, H. S. Gunawi, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. F. Naughton. Impact of Disk Corruption on Open-Source DBMS. In *IEEE 26th International Conference on Data Engineering*, 2010.

[166] X. Tang, Z. Zhang, M. Wang, Y. Wang, Q. Feng, and J. Han. Performance evaluation of light-weighted virtualization for paas in clouds. In *Algorithms and Architectures for Parallel Processing*, pages 415–428. Springer, 2014.

[167] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a Petabyte Scale Data Warehouse using Hadoop. *IEEE 26th International Conference on Data Engineering*, 2010.

[168] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data Warehousing and Analytics Infrastructure at Facebook. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1013–1020, 2010.

[169] J. Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[170] T. White. *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[171] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving Mapreduce Performance through Data Placement in heterogeneous Hadoop Clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), IEEE International Symposium on*, pages 1–9. IEEE, 2010.

[172] C. Yang, C. Yen, C. Tan, and S. R. Madden. Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *IEEE 26th International Conference on Data Engineering*, 2010.

[173] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. *Proceedings of the 5th European Conference on Computer Systems*, pages 265–278, 2010.

[174] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in heterogeneous Environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42, 2008.

[175] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *IEEE Symposium on Security and Privacy*, pages 313–328. IEEE, 2011.