Imperial College London
Department of Computing

Bridging the Gap between Serving and Analytics in Scalable Web Applications

by

Panagiotis Garefalakis

Supervised by Peter Pietzuch

Submitted in partial fulfilment of the requirements for the MRes Degree in High Performance Embedded and Distributed Systems (HiPEDS) of Imperial College London

September 2015

Abstract

Nowadays, web applications that include personalised recommendations, targeted advertising and other analytics functions must maintain complex prediction models that are trained over large datasets. Such applications typically separate tasks into *offline* and *online* based on the latency, computation and data freshness requirements. To serve requests robustly and with low latency, applications cache data from the analytics layer. To train models and offer analytics, they use asynchronous offline computation, which leads to stale data being served to clients. While web application services are deployed in large clusters, collocation of different tasks is rarely done in production mode in order to minimize task interference and avoid unpredictable latency spikes. The service-level objectives (SLOs) of latency-sensitive, online tasks can be violated by these spikes. Although this model of decoupling tasks with different objectives in web applications works, it lacks data freshness guarantees and suffers from poor resource efficiency.

In this thesis, we introduce *In-memory Web Objects* (IWOs). In-memory Web Objects offer a unified model to developers when writing web applications that have the ability to serve data while using big data analytics. The key idea is to express *both* online and offline logic of a web application as a single *stateful distributed dataflow graph* (SDG). The state of the dataflow computation is, then, expressed as In-memory Web Objects, which are directly accessible as persistent objects by the application. The main feature of the unified IWOs model is finer-grained resource management with low latency impact. Tasks can be cooperatively scheduled, allowing to move resources between tasks of the dataflow efficiently according to the web application needs. As a result, the application can exploit data-parallel processing for compute-intensive requests and also maintain high resource utilisation, e.g. when training complex models, leading to fresher data while serving results with low latency from IWOs. The experimental results on real-world data sets, presented in this thesis, show that we can serve client requests with 99th percentile latency below 1 second while maintaining freshness.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Peter Pietzuch, for his continuous support, inspiring counsel and guidance during the course of this work.

Next I would like to thank Imperial College London and more specifically High-Performance and Embedded Distributed Systems Centre for Doctoral Training (CDT) HiPEDS programme for the support during my studies.

I would also like to thank the members of Large-Scale Distributed Systems (LSDS) group for all the fun conversations, the technical discussions, the group meetings and every other moment that we shared together.

My best thanks to my friends and colleagues Ira Ktena, John Assael, Salvatore DiPietro, Nadesh Ramanathan, Nicolas Miscourides, Emanuele Vespa, Christian Priebe, Antoine Toisoul, Eric Seckler, Konstantinos Boikos, James Targett, Nicolas Moser and Julian Sutherland for making the CDT lab a fun and interesting place.

Finally, I would like to thank my wonderful parents Georgia and Giannis and my sister Eirini for their endless love, unconditional support, and for encouraging me to pursue my dreams.

To my parents,

Contents

List of Tables

List of Figures

1. Introduction

The rise of large-scale, commodity cluster, compute frameworks such as MapReduce [20], Spark [88] and Naiad [62], has enabled the increased use of complex data analytics tasks at unprecedented scale. A large subset of these complex tasks facilitates the production of statistical models that can be used to make predictions in applications such as personalised recommendations, targeted advertising, and intelligent services. Distributed model training previously assigned to proprietary data-parallel warehouse engines using MPI [69] is now part of the so-called "*BigData*" ecosystem, described in more detail in Section 2.5. Platforms like Hadoop [5], Spark [88] and the more recently developed Stratosphere/Flink [3] are part of this ecosystem and offer scalable high-volume data analysis. These platforms have a significant impact in the systems research community and a considerable amount of time is spent designing, implementing and optimising them both at an academic and industrial level.

Several modern social networking and e-commerce websites, offering insight to the user, are enabled by the above mentioned, distributed platforms. These websites implement a variety of data mining applications, providing derived data features. A typical example of this kind of feature is collaborative filtering [76] which showcases relationships between pairs of items based on people's ratings of those items. Organisations such as Amazon [50], YouTube [92], Spotify [90] and LinkedIn [76] take advantage of these models to provide a better service to the users. Petabytes of data from millions of users on thousands of commodity machines need to be processed by these applications. Consequently, they heavily depend on the "*BigData*" ecosystem to provide horizontal scalability, fault tolerance and multi-tenancy.

Another interesting aspect of "*BigData*", besides the aforementioned volume, their ubiquitous character. Data is received from servers, sensors, mobile phones, appliances, wearable devices, even toothbrushes. If technology predictions are right [60], all that ubiquitous data will be more valuable than ever. It is the key to change how business decisions are made and how consumers interact with brands and products. However, web applications today, whether ordering a taxi, or looking for a restaurant, have to know everything about the consumer at that specific moment, to deliver the best possible service. In order for this to happen they have to know what kinds of restaurants a consumer likes, what restaurants are available at that time, and finally deliver a recommendation based on the street corner just walked by. That requires real-time data processing across millions of devices which can be extremely challenging as the volumes of data increase in an unpredictable scale.

The main reason real-time processing for modern web applications is challenging is their strict response time SLOs. It is mandatory to offer low-latency responses to the end users, as the response time is affecting revenue. A slowdown experiment at Bing [42] showed that every 100-millisecond speedup could potentially improve revenue by 0.6%. The most common technique for achieving low latency in web applications is decoupling computationally-expensive *analytics* tasks such as model training or personalised recommendations from the critical path of *serving* web requests.

The tasks that are important for serving client requests are usually named latency-critical (LC) or online while the computation intensive tasks are often called best-effort (BE) or offline. The non-critical offline tasks are computed asynchronously using back-end systems. The developers, then, load pre-computed results into scalable stores such as distributed key/value stores to make data available for serving. The pre-computation part can be performed in a data-parallel fashion by frameworks such as Hadoop [5] or Spark [88] which is rather best effort than real-time, failing to comply with the ubiquitous character of "*BigData*".

Despite achieving low latency, the aforementioned approach has a number of limitations: First, decoupling data analytics (BE) from serving (LC) means that results can be stale, which has a negative impact on time-critical data such as user behaviour analysis or personalised services. An increasing number of companies is now trying to adapt, and compute that kind of critical data in real-time [7, 60]. Second, using key/value stores for storing data, while reducing the read latency, requires the construction of complex queries, involving multiple back-end stores; more than 70% of all Hadoop jobs running at LinkedIn [76] are reported to use key-value stores as egress mechanism confirming the scale of the problem. This is also part of the data integration problem that is frequently cited as one of the most difficult issues facing data practitioners [40]. Third, this variety of stores and back-end systems that must be managed and scaled out independently has proven to be hard, error-prone and inefficient. Last but not least, decoupling and hosting BE and LC tasks on different servers, while minimising the interference between colocated workloads or shared resources, leads to low machine utilisation negatively impacting datacenter cost effectiveness [52, 24, 75].

The low utilisation issue in web applications is part of a wider proneness to low resource efficiency at large, shared, private and public datacenters. Several research papers have reported that the average server utilization in most datacenters is low, ranging between 10% and 50% [24, 13, 73]. A primary reason for the low utilization is the popularity of latency-critical (LC) web services mentioned above, ranging from social media, search engines and online maps to webmail, online shopping and advertising. These user-facing services are typically scaled across thousands of servers. While their load varies significantly due to daily patterns and unpredictable bursts in user accesses, it is difficult to unify and consolidate load on a subset of highly utilized servers. This is due to the fact that the application state does not fit in a small number of servers and moving state is expensive. The cost of such underutilization can be significant. Even in companies using state of the art scheduling mechanisms like Google [52, 81] and Microsoft [41, 37], servers often have an average idleness of 30% over a 24 hour period [51] which could easily translate to a wasted capacity of thousands of servers [52].

A promising and convenient way to improve efficiency is by launching best-effort (BE) batch tasks on the same servers and exploiting any resources underutilized by LC workloads, as described in the literature [54, 55, 23]. Batch analytics frameworks like Spark [88] and MapReduce [20] can generate numerous BE tasks and derive significant value, even if these tasks are occasionally deferred or restarted [11, 13, 17, 24]. The main challenge of this approach is interference between colocated workloads on shared resources such as CPU, memory, disk, and network. LC tasks, like serving web requests, operate with strict service level objectives (SLOs) on tail latency, and even small amounts of interference can cause significant SLO violations [54, 49, 56]. Hence, some of the past work on workload colocation focused only on workloads aiming for a higher throughput [63, 16]. More recent systems predict, or detect when an LC task suffers significant interference from the colocated tasks and avoid or terminate the colocation [24, 54, 66, 79]. These systems protect LC workloads, but reduce the opportunities for higher utilization through colocation.

In this thesis, we present *In-memory Web Objects* (IWOs), a unified model for web applications. The goal of IWOs is twofold: First, to expose a common object-based interface to the developers for handling both *offline*, best effort and *online*, latency critical tasks. Second, to schedule resources efficiently between BE and LC tasks in a real-time and coordinated fashion, while maintaining the strict SLOs of request serving in web applications. This is achieved by fine-grained resource management inside the Java Virtual Machine and by being reactive to bursty workloads. Compared to other existing systems such as CPI [91], Quasar [24], and Bubble-Up [54] that prevent colocation of interfering workloads, we enable an LC task like serving to be colocated with any BE job like user recommendations. We guarantee that the LC job will get enough shared resources to achieve its SLO, maximizing the utility from the BE task which is now producing fresher results. Using real-time latency monitoring for serving (LC) tasks we can identify when shared resources become a bottleneck and are likely to cause SLO violations. Then we are using reactive thread scheduling to shift resources from the BE to the LC tasks in order to increase the isolation and prevent that from happening.

1.1. Thesis Contributions

This thesis, makes the following main contributions:

- Introduces In-memory Web Objects (IWOs), offering a unified model to developers when writing web applications that have the ability to serve data while using big data analytics.
- The design of IWOs isolation mechanism that is based on cooperative task scheduling. Cooperative task scheduling reduces the scheduling decisions and allocates resources in a fine-grained way, leading to improved resource utilisation.
- The implementation of *In-memory Web Objects* (IWOs) based on SEEP [26], an open-source, data-parallel processing platform that supports stateful dataflow graphs.
- The evaluation of IWOs by implementing a real web application similar to Spotify [45], with both online/LC and offline/BE tasks. The web application was implemented as an extension of Play [78], a popular web application framework for Java and the evaluation was conducted using real-world data [9].

1.2. Thesis Organization

This thesis is organized as follows: Chapter 2 provides background and relates *In-memory Web Objects* to other work in the field. The overall design is described in Chapter 3 while Chapter 5 provides details of the implementation. Motivation and the results of the case study evaluation are presented in Chaprter 4 and Chapter 5 respectively. Finally, we conclude and discuss the directions of future work in Chapter 6.

2. Background

In-memory Web Objects (IWOs) model lays in the intersection of several areas including web applications dealing with "*BigData*", cluster computing, scalable data-parallel processing, stream processing, efficient resource scheduling/management and scalable systems in general. This chapter is aiming to provide a deeper background in parts of these areas but also relate IWOs with already existing solutions. While important for understanding the technologies behind IWOs, this is not prerequisite reading, and the reader can skip to Section 3.

2.1. Scalable Web Applications

The popularity of modern latency critical (LC) web applications that must offer user-centric services led to the expansion of datacenter-scale computing, on clusters with thousands of machines [41, 24]. A broad class of data analytics tasks is now routinely carried out on such large clusters over large heterogeneous datasets. This is often referred to as "Big Data" computing, and the diversity of applications sharing a single cluster is growing dramatically. The reasons vary from the consolidation of clusters to increase efficiency, to data diversity and the increasing variety of techniques ranging from query processing to machine learning being used to understand that data. Of course, the ease of use of cloud-based services, and the expanding adoption of "Big Data" technologies have also affected cluster growth among traditional organizations.

In a typical web application as depicted in Figure 2.1, clients send HTTP requests that are handled by a load-balancer. The load-balancer can be in a form of a web proxy like nginx [72]. The proxy handles the distribution of requests in a number of web servers, increasing dynamically the number of users the application can handle. The web server itself is just responsible for the user interface and page rendering. The web application business logic is implemented using a framework chosen by the developer, usually in an Object-Oriented Programming (OOP) language exposing an API. This type of separation between the logic and the representation is dictated by the popular Model-View-Controller (MVC) web application paradigm [44].

In the Model-View-Controller pattern, models represent knowledge. A *model* can be a single object or a structure of objects, and there is a one to one mapping between the model and its representation. A *view* is a visual representation of its model. It can highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter. A view is attached to a particular model and gets the data necessary for the presentation by querying the model. A *controller* is the link between a user and the system. More specifically in a web app its functionality is twofold. The first part is the web server, mapping incoming HTTP URL requests to a particular handler for that request. The second part is those handlers themselves, which are also called controllers. Thus, a web app MVC includes both the web server routing requests to handlers and the logic of those handlers, which pull the data from the database and push it into the template. The controller also receives and processes HTTP POST requests, sometimes updating the database. In a typical implementation of the design as depicted in Figure 2.1, both the web server and the

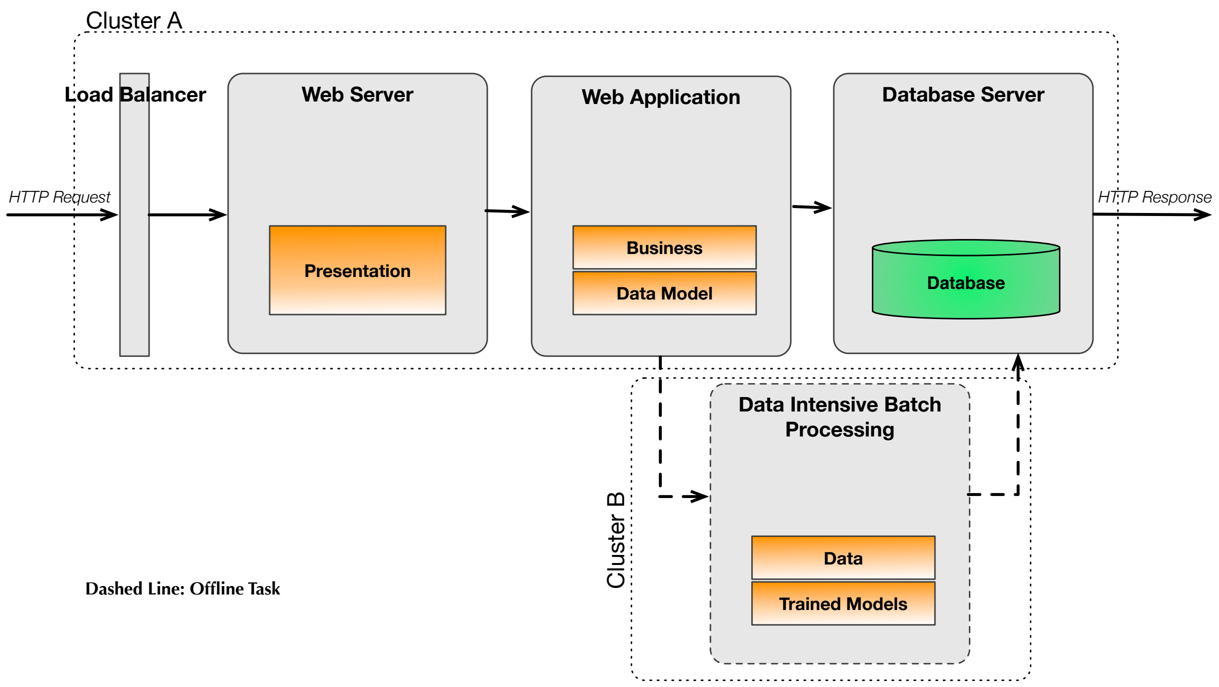


Figure 2.1.: Typical web application architecture today.

web application logic would be part of a typical framework code like Play [78]. The business logic including the data models is implemented by the application framework as well.

All the data defined by the data model, are stored in external databases, that can also be scalable using for instance distributed key/value stores [6, 47]. For the online client serving task, the application has to run complex queries on these databases to fetch the needed data. For the offline/analytics task like collaborative filtering and online regression, the web application *asynchronously* sends new user data to the distributed batch processing framework that runs on a separate cluster for isolation. The batch framework periodically processes the newly created data and pushes the changes in the database [76]. Consequently, clients and developers observe analytics data that are as fresh as the processing task completion time, which is in the best case hundreds of seconds. Moreover, strictly decoupling online from offline tasks increases drastically the machine demands for each web application with a negative impact on the cluster cost effectiveness.

2.2. Cluster Computing

Cluster Computing is tightly coupled with the web applications and "Big Data" trend, and has emerged as a significant evolution in the way that businesses and individuals consume and operate computing resources. Cloud computing paradigm that evolved from cluster computing made cheap infrastructure accessible to everyone, and this infrastructure can be used to analyse huge volumes of data. The main goal of cloud computing is to reduce the cost by *sharing resources*. Increase reliability, and flexibility by transforming computers from a device that we buy and operate ourselves to something that is managed by a third party. However, most of the challenges in distributed systems, arise from this fundamental need for cheap *shared* computing power. Moreover, since we are using commodity hardware, it is highly likely for this hardware to fail, quite often. As a result cloud providers often rely on redundancy to achieve continuous operation.

2.3. Cluster Resources

2.3.1. Interference

One of the main challenges cloud computing and *shared* cluster environments in general introduced is *resource interference*. When a number of workloads execute concurrently on a server, they compete for shared resources. The main resources that are affected are CPU, caches (LLC), memory (DRAM), and network. There are also non-obvious interactions between resources, known as *cross-resource interactions*. For instance, a BE task can cause interference in all the shared resources mentioned. Therefore, it is not sufficient to manage a single source of interference. To maintain strict service SLOs, recent resource managers [52], monitor and isolate all potential sources. For example cache contention causes both types of tasks (BE and LC) to require more DRAM bandwidth, creating a DRAM bandwidth bottleneck. Likewise, a task that notices network congestion may attempt to use compression, causing more CPU contention.

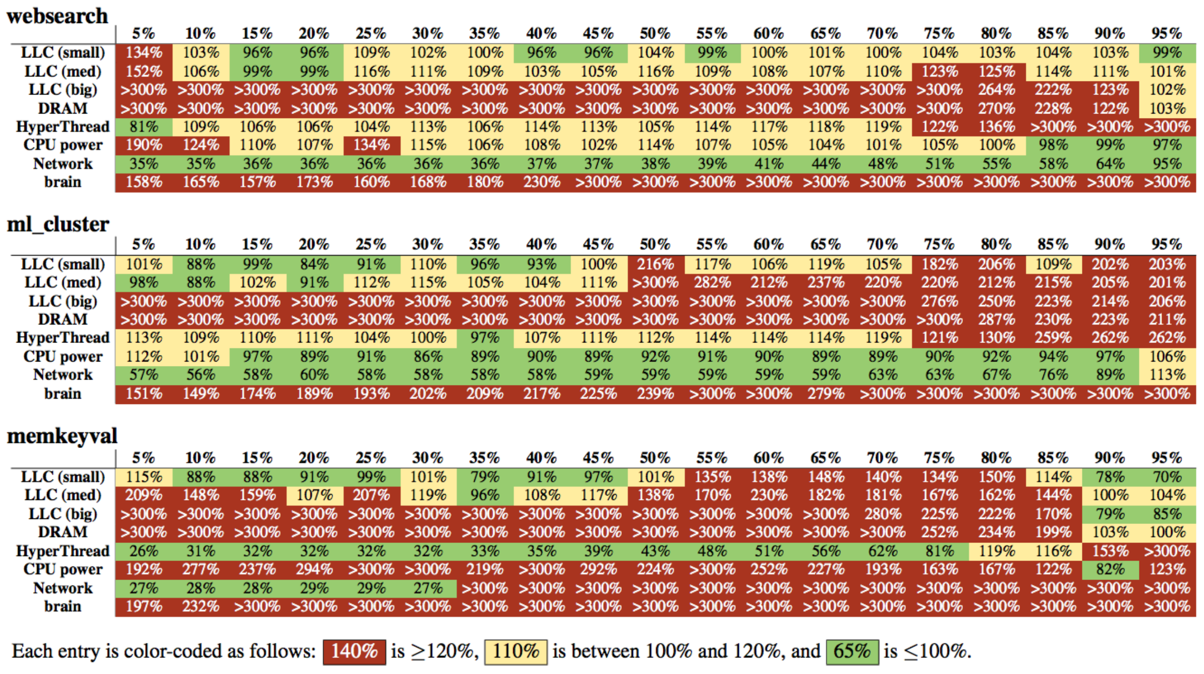
The Operating System (OS) by default enables tasks isolation through scheduling. While there are more mechanisms that can be used for isolation described in Section 2.3.2, it is interesting to evaluate the overall impact of running a BE task along with a LC task using only the mechanisms provided by the OS. The evaluation, provided by the Heracles [52] system was conducted with real services workloads from a Google cluster. In more detail, there were two workloads running (one BE and one LC), executed in two separate Linux containers with the BE priority set to low. This scheduling policy was enforced by the Completely Fair Scheduler (CFS) using the shares parameter, where the BE task receives few shares compared to the LC workload. The BE task was the Google brain workload [48] while the LC tasks were websearch, ml_cluster, and memkeyval described below.

websearch: is the query serving portion of a production Google web search service. It is a scale-out workload where each query spawns thousands of leaf processing nodes. The SLO for every single leaf node is tens of milliseconds for the 99%-ile latency. This workload has high DRAM memory footprint and moderate bandwidth requirements. It is mostly computational intensive and does not consume network bandwidth.

ml_cluster: is a standalone service performing real-time text clustering using machine learning. The SLO is tens of milliseconds for the 95%-ile latency, and it is mostly memory bandwidth intensive with medium CPU and low bandwidth requirements.

memkeyval: is an in memory key-value store similar to memcached [29], but with really strict SLOs in terms of few hundreds of microseconds for the 99%-ile latency. Unlike the other two workloads, memkeyval is network bandwidth limited and CPU intensive while the DRAM usage is limited.

For the evaluation, these 3 LC workloads were running on a single node, along with resource greedy BE tasks. Figure 2.2 presents the impact of the interference of the BE task on the tail latency of the LC workloads. Each row depicts the tail latency at a certain load. The interference is acceptable only if the tail latency is below 100% of the target SLO. By observing the rows in all three systems, we immediately notice that the current OS isolation mechanisms are inadequate for colocated tasks (LC and BE). Even at low load, BE tasks create sufficient pressure on the shared resources to lead to SLO violations. There are differences depending on the LC sensitivity on shared resources. For example, memkeyval is sensitive to network usage while the other two are not affected. Websearch is affected by cache interference but not the other two. It is also interesting that the impact of interference changes depending on the load of the workload.



websearch

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	134%	103%	96%	96%	109%	102%	100%	96%	96%	104%	99%	100%	101%	100%	104%	103%	104%	103%	99%
LLC (med)	152%	106%	99%	99%	116%	111%	109%	103%	105%	116%	109%	108%	107%	110%	123%	125%	114%	111%	101%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	264%	222%	123%	102%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	270%	228%	122%	103%
HyperThread	81%	109%	106%	106%	104%	113%	106%	114%	113%	105%	114%	117%	118%	119%	122%	136%	>300%	>300%	>300%
CPU power	190%	124%	110%	107%	134%	115%	106%	108%	102%	114%	107%	105%	104%	101%	105%	100%	98%	99%	97%
Network	35%	35%	36%	36%	36%	36%	36%	37%	37%	38%	39%	41%	44%	48%	51%	55%	58%	64%	95%
brain	158%	165%	157%	173%	160%	168%	180%	230%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%

ml_cluster

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	101%	88%	99%	84%	91%	110%	96%	93%	100%	216%	117%	106%	119%	105%	182%	206%	109%	202%	203%
LLC (med)	98%	88%	102%	91%	112%	115%	105%	104%	111%	>300%	282%	212%	237%	220%	220%	212%	215%	205%	201%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	276%	250%	223%	214%	206%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	287%	230%	223%	211%
HyperThread	113%	109%	110%	111%	104%	100%	97%	107%	111%	112%	114%	114%	114%	119%	121%	130%	259%	262%	262%
CPU power	112%	101%	97%	89%	91%	86%	89%	90%	89%	92%	91%	90%	89%	89%	90%	92%	94%	97%	106%
Network	57%	56%	58%	60%	58%	58%	58%	58%	59%	59%	59%	59%	59%	63%	63%	67%	76%	89%	113%
brain	151%	149%	174%	189%	193%	202%	209%	217%	225%	239%	>300%	>300%	279%	>300%	>300%	>300%	>300%	>300%	>300%

memkeyval

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	115%	88%	88%	91%	99%	101%	79%	91%	97%	101%	135%	138%	148%	140%	134%	150%	114%	78%	70%
LLC (med)	209%	148%	159%	107%	207%	119%	96%	108%	117%	138%	170%	230%	182%	181%	167%	162%	144%	100%	104%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	280%	225%	222%	170%	79%	85%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	252%	234%	199%	103%	100%
HyperThread	26%	31%	32%	32%	32%	32%	33%	35%	39%	43%	48%	51%	56%	62%	81%	119%	116%	153%	>300%
CPU power	192%	277%	237%	294%	>300%	>300%	219%	>300%	292%	224%	>300%	252%	227%	193%	163%	167%	122%	82%	123%
Network	27%	28%	28%	29%	29%	27%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
brain	197%	232%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%

Each entry is color-coded as follows: 🟥 140% is ≥120%, 🟨 110% is between 100% and 120%, and 🟩 65% is ≤100%.

Figure 2.2.: Impact of interference on shared resources on 3 different workloads. Each row is an antagonist and each column is a load point for the workload. The latency values are normalised to the SLO latency [52].

2.3.2. Isolation

The observations from Figure 2.2 clearly state that the default OS scheduler is inadequate, and motivate the need for better resource isolation mechanisms, especially for LC applications. This section describes the major sources of interference and summarises the available isolation mechanisms.

Resource	Interference Proneness	SW Isolation	HW Isolation
CPU	✓	cgroups, cpuset [58]	✗
Cache (LLC)	✓	Q-Clouds [63]	Cache Partitioning [71]
Memory (DRAM)	✓	Fair queuing [64], QoS contr. [38]	✗
Network	✓	DC-TCP [4]	DC frabric topologies [2]

Table 2.1.: Major resources prone to interference in a datacenter environment and available SW and HW isolation mechanisms.

CPU

The primary shared resource in a server is CPU *cores* on one or more sockets. We can statically partition cores between the LC and BE tasks using software mechanisms such as *cgroups* and *cpuset* [58]. This is though inefficient, because when user-centric web services face a load spike, they need all available cores to meet throughput demands to avoid latency SLO violations. Similarly, we can not simply assign high priority to LC tasks and rely on OS-level scheduling of cores between tasks as experimentally demonstrated in Figure 2.2. Common OS scheduling algorithms such as Linux's completely fair scheduler (CFS) reported having vulnerabilities that lead to frequent SLO violations when LC tasks are colocated with BE tasks [49]. Real-time scheduling algorithms like *SCHED_FIFO* are not work-preserving and result in low utilisation. It is clear that we need a way to assign efficiently and dynamically cores to LC tasks based on demand.

Cache

Several studies have shown that uncontrolled interference on the shared cache (LLC) can be harmful to colocated tasks [24, 31, 74]. To address this issue, Intel has recently introduced LLC cache partitioning in server chips enabling way-partitioning of highly-associativity. Cores are assigned to one cache subset and can only allocate cache lines in that subset. They are however allowed to hit any part of the LLC. Even when colocation is between throughput tasks (BE), it is best to dynamically manage cache partitioning using either hardware such as Utility-Based Cache Partitioning [71] or software techniques such as Q-Clouds [63]. In the presence of LC workloads, dynamic management is more critical as interference translates to large latency spikes [49] that could hurt the service response time and, as a result, the revenue [42].

Memory

Most of the latency critical (LC) services operate on huge datasets that do fit in caches (LLC). Therefore, they affect DRAM as they consume more bandwidth at high loads, and they are sensitive to DRAM bandwidth interference. Despite numerous studies being published in memory bandwidth isolation [64, 38], there are no hardware isolation mechanisms in commercially available chips. In multi-socket servers on can isolate workloads across NUMA channels but this approach constrains DRAM capacity allocation and could cause interleaving. The lack as mentioned earlier of hardware support for memory bandwidth isolation complicates and constrains the efficiency of any system that dynamically manages workload colocation.

Network

Private or public datacenters contain applications that scale-out and generate network traffic. Many datacenters use rich topologies with sufficient bisection bandwidth to avoid routing congestion in the fabric [2]. There is also a number of networking protocols that prioritize short messages for LC tasks over large messages for BE tasks [4]. Within a server, interference can occur both in the incoming and the outgoing direction of the network link. If a BE task causes incoming traffic interference, we can throttle its core allocation until networking flow control mechanisms trigger. In the outgoing direction, we can use traffic control mechanisms in operating systems like Linux to provide bandwidth guarantees to LC tasks and to prioritize their messages ahead of those from BE tasks.

Containers

Instead of isolating a single resource at a time using one of the mechanisms above, Linux containers provide a unified solution. Linux Containers (LXC) is an operating-system-level virtualization environment for running multiple isolated systems containers on a single control host [1]. The Linux kernel provides the cgroups [58] functionality allowing limitation and prioritization of resources such as CPU, memory and network, without the performance and resource penalty of a virtual machine. Namespace isolation functionality allows complete isolation of an applications' view of the operating environment, including process trees, networking, user IDs and mounted file systems. LXC combines kernel's cgroups and support for isolated namespaces to provide an isolated environment for applications. Other alternatives like Docker containers [59] which started as a wrapper of LXC, now provide similar isolation mechanisms using different tools. However, as we experimentally demonstrate in Chapter 4, these solutions lack distributed design, and they suffer from slow startup times making them inadequate for dynamic resource management.

Application Level

Other approaches towards efficient resource management and isolation include application-level techniques such as IBM's multitenant virtual machine (VM) [19] and Oracle's Barcelona project [18]. The goals of these projects are identical, consolidate applications in the same Java Virtual Machine (JVM) and benefit from sharing common resources such as caches, and heap memory. Sharing the JVM saves both memory and processor time. When applications share the same VM, it is easier to apply SLOs and resource isolation mechanisms defining how many resources each tenant is going to use. Figures 2.3 and 2.4 depict the differences in the deployment between a standard dedicated VM and a multitenant VM. The multitenant feature is promising to enable deployments gain the advantages of sharing the JVM while maintaining better isolation. For unknown reasons, though, the research and development of both these projects [18, 19] is paused.

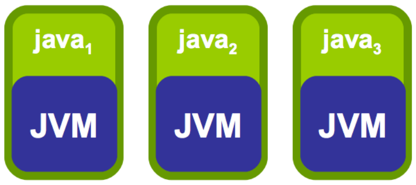


Figure 2.3.: A standard java invocation creates a dedicated (non-shared) JVM in each process [19].

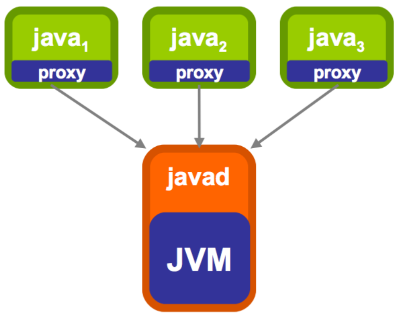


Figure 2.4.: Multitenant JVM uses a lightweight 'proxy' JVM in each java invocation. The 'proxy' knows how to communicate with a shared JVM daemon called javad [19].

In more detail, the main benefit of using a multitenant JVM is that deployments avoid the memory consumption that is typically associated with *multiple identical* mechanisms of standard JVMs. The main overhead is introduced by the Java heap, consuming hundreds of megabytes of memory. Heap objects cannot be shared between JVMs, even when the objects are identical. Furthermore, JVMs tend to use all of the heap that's allocated to them even if they need the peak amount for a short period. Another cause is the just-in-time (JIT) compiler that consumes tens of megabytes of memory. The JIT generated code is private, consumes memory and also significant processor cycles, stealing time from other running applications. Internal artefacts for classes, such as String and Hashtable, which exist for all applications, also consume memory. One instance of each of these artefacts exists for each JVM. Each JVM has a GC helper thread per core by default and also multiple compilation threads. Compilation or GC activity can occur simultaneously in one or more of the JVMs, which can hurt the performance as the JVMs will compete for limited processor time.

In-memory Web Objects (IWOs) share some concepts with multitenant VMs as both target fine-grained application resource management written in the same programming language, in both cases Java. They are different though in the sense that IWOs are targeting a specific class of web applications unifying serving and analytics. We believe that the main reason projects like multitenant JVM [19] did not proceed is that they tried to support a broad range of applications with different needs and challenges. IWOs also have a distributed character that was not intended in any of the multitenant implementations. Furthermore, IWOs do not make use of any static policy described by the isolation mechanisms above that would be either too conservative or overly

Workload	% Long Jobs	% Task-Seconds
Google 2011	10.00%	83.65%
Cloudera-b 2011	7.67%	99.65%
Cloudera-c 2011	5.02%	92.79%
Cloudera-d 2011	4.12%	89.72%
Facebook 2010	2.01%	99.79%
Yahoo 2011	9.41%	98.31%

Table 2.2.: Long jobs in heterogeneous workloads form a small fraction of the total number of jobs, but use a large amount of resources. [22]

optimistic ending up either wasting resources or leading to SLO violations.

2.3.3. Management

The popularity of modern web applications and the strict decoupling of latency critical (LC) tasks from the best effort (BE) analytics tasks is also affecting the workload heterogeneity in current datacenters [73]. Short BE analytics tasks usually dominate typical workloads. Long running LC jobs are considerably fewer but dominate in terms of resource usage. Previous work [22], analysed the degree of the aforementioned heterogeneity in real workloads, from publicly available Google traces [83, 73]. Table 2.2 summarises the jobs ordered by average task duration. The top 10% jobs account for 83.65% of the task-seconds. Moreover, they are responsible for 28% of the total number of tasks and their average task duration is 7.34 times larger than the average task duration of the remaining 90% of jobs.

Running highly utilized datacenters introduces several orthogonal challenges beyond a single server and can be considered a sophisticated *cluster management* task. The problem consists of scheduling jobs to servers in a scalable fashion such that all resources in the cluster are efficiently used and also taking into account strict SLOs of long running tasks and interference issues. Some of these issues (interference and SLOs) along with the workload diversity is addressed by a variety of modern frameworks such as YARN [80], Mesos [35], and Omega [75] described in the next Section.

2.4. Cluster Resource Scheduling

Resource scheduling frameworks like Mesos [35] and YARN [80], expose cluster resources via a well-defined set of APIs. This enables concurrent sharing between applications with a variety of differing characteristics, ranging from BE batch jobs to LC long running web services. These frameworks, while differing on the exact solution monolithic, two level (explained in more detail in Section 2.4.5) or shared-state, are built around the same notion of centralized coordination to schedule cluster resources and they are considered centralized scheduler solutions.

2.4.1. Centralised

In these centralised solutions, individual per-job or per-application framework managers request resources from the centralized scheduler via the resource management APIs. Then they coordinate application execution by launching tasks within such resources. Apparently, these centralized designs simplify cluster management in that it is a single place where scheduling invariants such as capacity and fairness) are specified and enforced. Furthermore, the central scheduler has cluster-wide visibility and can optimize task placement along multiple dimensions like locality [86], packing [32], and more. However, the centralized scheduler is, by design, in the *critical path* of all allocation decisions. This poses scalability and latency concerns. Centralized designs rely on heartbeats that are used for both liveness and for triggering allocation decisions. As the cluster size scales, to minimize heartbeat processing overheads, operators are forced to lower the heartbeat rate. In turn, this increases the schedulers allocation latency. This compromise becomes problematic if typical tasks are short as described in the Sparrow [68] paper.

2.4.2. Distributed

Fully distributed scheduling is the leading alternative to obtaining high scheduling throughput. A practical system leveraging this design is Apollo [11]. Apollo allows each running job to perform independent scheduling choices and to queue its tasks directly at worker nodes. Unfortunately, this approach relies on a uniform application type workload, as all job managers need to run the same scheduling algorithm. In this context, allowing arbitrary applications while preventing SLO abuses and strictly enforcing capacity and fairness guarantees is non-trivial. Furthermore, due to lack of global view of the cluster state, distributed schedulers make local scheduling decisions that are often *not globally optimal*. Other approaches try to compensate these non-optimal decisions using global reservations and work stealing [22].

2.4.3. Executor Model

To amortize the high scheduling cost of centralized approaches, the *executor* model has been proposed [57, 43, 15]. This hierarchical approach focuses in reusing containers assigned by the central scheduler to an application framework that multiplexes them across different tasks. Reusing containers, though, assumes that submitted tasks have similar characteristics and can fit in existing containers. Moreover, since the same system-level process is shared across tasks, the executor model has limited applicability within a single application type.

2.4.4. Hybrid

More recently some hybrid schedulers were proposed such as Mercury [41], and Hawk [22]. In these approaches long LC jobs are scheduled using a centralized scheduler while short BE ones are scheduled in a fully distributed way. For example in Hawk [22], a small portion of the cluster is reserved for the use of short jobs. To compensate for the occasional poor decisions made by the distributed scheduler, the authors propose a randomized work-stealing algorithm.

2.4.5. Two Level

In this section, we further explain Apache Mesos [35], a distributed scheduler, also used for our case-study of Section 5.2. Mesos is a popular open source project mainly due to the out-of-the-box compatibility with a Spark [88], a data parallel processing framework. A key design point that allows Mesos to scale is its use of a two-level scheduler architecture. By delegating the actual scheduling of tasks to frameworks, the master can be very light-weight and easier to scale as the size of the cluster grows. This is because the master does not need to know the scheduling intricacies of every type of application that it supports. Also, since the master does not handle the scheduling of every task, it does not become a performance bottleneck which is the case when using a monolithic scheduler to schedule every task or VM.

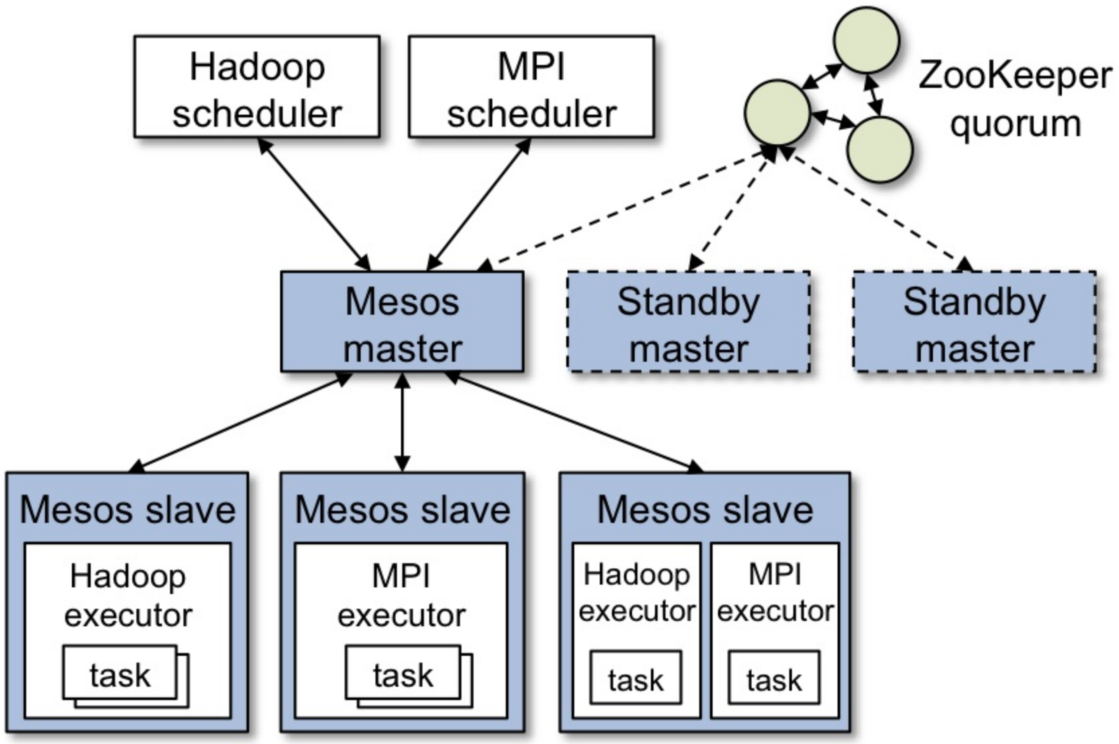


Figure 2.5.: Mesos architecture including the main components [35]

Figure 2.5 depicts the main components of Mesos. Mesos consists of a master daemon that manages slave daemons running on each cluster node, and Mesos applications also termed frameworks, that run tasks on these slaves. The master enables fine-grained resource sharing, such as CPU and memory, across applications by sending them resource offers. Each resource offer contains a list of demands. The master decides how many resources to offer to each application framework according to a given organizational policy, such as fair sharing, or strict priority. To support a diverse set of policies, the master employs a modular architecture that makes it easy to add new allocation modules via a plugin mechanism.

An application framework running on top of Mesos consists of two components. The first one is a scheduler that registers with the master to be offered resources, and the other one is an executor process that is launched on slave nodes to run the frameworks tasks [35]. While the master determines how many resources are offered to each framework, the frameworks' schedulers select which of the offered resources to accept. When a framework accepts offered resources, it then sends to Mesos master a description of the tasks to be executed. In turn, Mesos launches the tasks on the corresponding slaves and enforces resource isolation using existing techniques like cgroups [58] and containters [1].

2.5. The Big Data ecosystem

Efficient resource management gained increasing popularity with the explosion of different data parallel processing frameworks. Processing frameworks greedily consume cluster resources, with the goal of simplifying cluster programming and providing data analytics. The ability to perform large-scale data analytics over huge data sets has proved to be a competitive advantage in a wide range of industries (retail, telecom, defence, etc.) in the past decade. In response to this trend, the research community and the IT industry have proposed a number of platforms to facilitate large-scale data analytics. Such platforms include a new class of databases, often referred to as NoSQL data stores [21, 6, 30], as well as a new set of the aforementioned frameworks that can achieve parallel data processing [20, 36, 88].

Storing huge amount of data can be challenging, but it is not the biggest issue anymore. Companies already store a significant amount of information on a daily basis. For example, Facebook was able to store data from its user on-line activity directly to its backend in 2005 [82]. LinkedIn more recently developed Kafka [46], a scalable publish/subscribe system based on an optimised distributed write ahead log (WAL), to store the user feed of millions of users efficiently. In the data analytics part, MapReduce [20] that was first developed at Google for indexing web pages back in 2004, now is a popular programming model that triggered scientists to rethink how large-scale data operations should be handled. Hadoop [5], which is the open source implementation of MapReduce, Spark [88], Dryad [36] and Flink [3], are other examples of powerful processing tools that contributed to the "Big Data" era. These platforms spread data across a number of commodity servers, and use the processing power of those machines to produce useful results. This scheme is highly attractive because commodity servers are cheap, and as data grow, one can increase the performance by adding more servers, a process known as "scaling out".

Every "Big Data" system, in general, is build to allow storing, processing, analysing and visualising data. In such environment, there is a typical hierarchy starting with the *infrastructure*, and selecting the appropriate tools for storing, processing and analysing data. Then there are specialised *analytics* tools with the ability to expose trends and give insight within these data. Finally, there are applications running on top of the processed, analysed data. There is a variety of different components implemented with similar goals across the hierarchy, some of which depicted in Figure 2.6, forming the so-called "Big Data ecosystem".

Infrastructure

Infrastructural technologies are the core of the ecosystem. They process, store and also have the ability to analyse data. The rise of unstructured data created the need for storing data beyond rows and tables. As a result new infrastructural technologies emerged, capable of capturing a plethora of data, and making it possible to run applications on systems with thousands of nodes, potentially involving petabytes of data. Some of the key infrastructural systems include Hadoop [5], Spark [88] and NoSQL databases like HBase [6] and Cassandra [47].

Analytics

Although infrastructural technologies sometimes involve data analysis, there are specific technologies that are designed specifically with analytical capabilities in mind. Some of the key data processing systems include scalable batch processing systems like MapReduce [20], Spark [88] and Naiad [62] while more recently the need for more real-time or near real-time analytics created a shift for scalable stream processing systems like Apache Storm [77], Flink [3] SEEP [26], Yahoo's S4 [65] and others. These systems are key to achieve scalable, *low latency* processing and they are further explained in Section 2.7.

Applications

Applications on top of these systems lie in the broad area of security, e-commerce, trading, health and more. Popular examples include modern social networking platforms like Facebook [82], and digital music services like Spotify [90], offering personalised services and insight to the user behaviour. These web services attract millions of users across the world creating the need for increasing scalability. In order to improve their service, these platforms implement a variety of machine learning algorithms used to train different models over large datasets. They also need efficient, fast and timely data processing, as close to the user activity as possible. The scalability and large, real-time data processing demands well define the profile of data intensive web applications *IWOs* model is targeting.

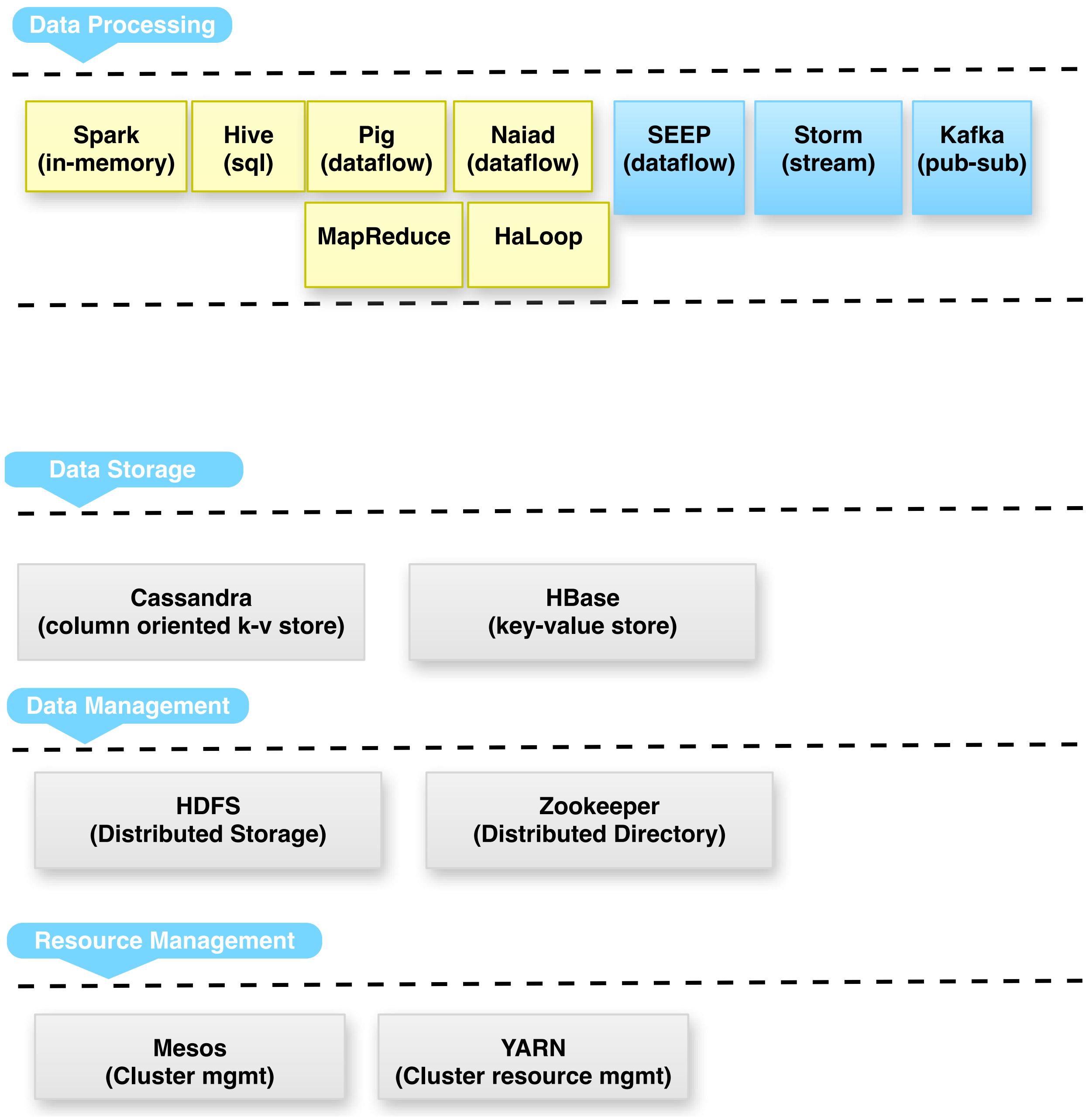


Figure 2.6.: Big Data ecosystem containing systems for data storing, processing and management.

2.6. Data-Parallel processing Frameworks

The “Big Data” ecosystem is tightly coupled with the analytics part which is responsible for producing valuable insight from huge unstructured and ubiquitous datasets. Data parallel frameworks can scale computation to a large number of machines but require developers to adopt a specific functional, declarative or dataflow programming model. While the first generation of data parallel frameworks like MapReduce [20] followed a more restricted functional programming model, latest implementations like Spark [88] and Flink [3] expose a more expressive and complex programming model in order to implement a richer set of high level functions.

2.7. Dataflow Model

Depending on the analytics algorithm, the latency requirements and the input data, some of these parallel frameworks might be more suitable than others. More precisely, for machine learning algorithms like k-means, logistic regression and recommendation systems it is mandatory to maintain large state when need to compute with low latency [27]. A suitable, high-level abstraction for these algorithms could be an executable distributed dataflow representation. Depending on the nature of the algorithm, the dataflow representation can be either stateful or stateless. In every dataflow, though, as in every data stream system, there is a stream of data elements flowing between the source and the operators, ending up to a sink as described below.

Data Stream. A Data Stream is a continuous partitioned and partially ordered stream of data elements, which can be potentially infinite. Each data element of the stream follows a predefined schema [8].

Data Element. A Data Element is the smallest unit of data which can be processed by a data streaming application, as well as the smallest unit that can be sent over the network. A Data Element can be any object of a given expected schema.

Data Stream Source. A source that produces a possibly infinite amount of data elements, which can not be placed into the memory, thus making batch processing infeasible.

Data Stream Sink. A Data Stream Sink is an operator representing the end of a stream. Data Stream Sinks have only an input data stream and no output data stream. In MapReduce programming model, an operator writing to a file or database can be considered as a Sink operator.

Data Stream Operator. A Data Stream Operator is a function that applies a user-defined transformation in the incoming stream of data elements and emits the transformed data to the outgoing streams.

2.7.1. Stateless

Stateless dataflows, fist made popular by MapReduce [20]. They define a functional dataflow graph in which vertices are stateless data-parallel tasks. There is no distinction between state and data. In Spark [88], dataflows are represented as RDDs, which are immutable. The immutability simplifies the failure recovery but requires a new RDD for every new state update [87]. This is extremely inefficient for online algorithms like collaborative filtering (CF) where only a part of a matrix is updated every time.

In the stateless model, data is also unified with the state, so a system cannot treat them differently. For example, they cannot use custom index data structures for state access or cache only state in memory. Shark [84], which supports caching needs hints for which explicit dataflows to cache. Part of this problem was solved with incremental dataflows which is a stateful scenario as explained below.

2.7.2. Stateful

Stateful dataflow models that represent state explicitly include systems like SEEP [26] and Naiad [62]. These systems permit tasks to gain access to in-memory data structures but have to deal with challenges like large state management. One assumption usually made is that state is often small compared to the data. In the case of large state though requiring distributed processing through partitioning or replication they do not provide abstractions for that.

Incremental dataflows are also fundamentally stateful since they maintain results from previous computations. They avoid rerunning entire jobs after updates occur to the input data, but they cannot avoid caching all data since they cannot infer which data will be reused in the future. Systems like Incoop [10] and Nectar [33] use this caching technique and systems like CBP automatically transform batch jobs for incremental computation [53]. Piccolo's [70] runtime supports distributed state as it provides a key/value abstraction, but it is not compatible with the dataflow model.

Table 2.3 classifies existing data-parallel processing models according to their programming model. The table also includes features that a model should support to enable the translation of a modern machine learning algorithm from an imperative programming language like Java, as described by Fernandez et al [27]. The required features of a dataflow model to support the algorithms with large size of mutable state like online recommendations or machine learning algorithms include:

- *large state* size (in the order of GBs)
- state *fine-grained updates*
- process data in *low latency*
- *iteration support*
- *fast failure recovery*

			State Handling			Dataflow		
Computational Model	Systems	Program. Model	Large state size	Fine-grained updates	Execution	Low Latency	Iteration	Failure Recovery
Stateless dataflow	Pig [67]	functional	n/a	✗	scheduled	✗	✗	recompute
	MapReduce [20]	map/reduce	n/a	✗	scheduled	✗	✗	recompute
	DryadLINQ [85]	functional	n/a	✗	scheduled	✗	✓	recompute
	Spark[88]	functional	n/a	✗	hybrid	✗	✓	recompute
	CIEL[61]	imperative	n/a	✗	scheduled	✗	✓	recompute
	HaLoop[12]	map/reduce	✓	✗	scheduled	✗	✓	recompute
Incremental dataflow	Incoop [10]	map/reduce	✓	✗	scheduled	✗	✗	recompute
	Nectar [33]	functional	✓	✗	scheduled	✗	✗	recompute
	CBP [53]	dataflow	✓	✓	scheduled	✗	✗	recompute
Batched dataflow	Comet [34]	functional	n/a	✗	scheduled	✓	✗	recompute
	D-Streams [89]	functional	n/a	✗	hybrid	✓	✓	recompute
	Naiad [62]	dataflow	✗	✓	hybrid	✓	✓	sync. global c/p
Continuous dataflow	Storm [77]	dataflow	n/a	✗	pipelined	✓	✗	recompute
	SEEP [26]	dataflow	✗	✓	pipelined	✓	✗	sync. local c/p
Parallel in-memory	Piccolo [70]	imperative	✓	✓	n/a	✓	✓	async. global c/p
Stateful dataflow	SDG [27]	imperative	✓	✓	pipelined	✓	✓	async. local c/p

Table 2.3.: Design space of data-parallel processing frameworks [27]

While most of the current data-parallel frameworks do not handle large state efficiently it is clear that in order to enable IWOs we need stateful tasks. These tasks should be able to expose

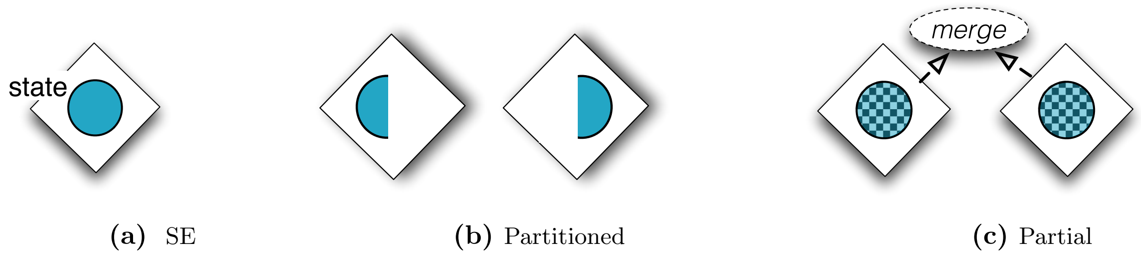


Figure 2.7.: Types of distributed state in SDGs [27]

and manipulate their state in an efficient way. In stateless frameworks, computation is defined through functional tasks. Any modification to state must be implemented as the creation of a new immutable data which is prone and inefficient. While most recent frameworks like Naiad [62] and SEEP [26] report the need for per-task state, they lack abstractions for distributed state. IWOs are building on top of SEEP [26], a data-parallel stream processing system to provide extra abstractions.

IWOs state abstractions are identical to then ones exposed by SGDs [27], but they are implemented as part of a web application framework. In more detail as depicted in Figure 2.7, a state element can be distributed in different ways. A *partitioned* state element splits its internal data structure into disjoint partitions. Access to that kind of state instances occurs in parallel. If this is not possible, a *partial* state element duplicates its data structure, creating multiple copies that are updated independently. When a task accesses a partial state, there are two possible types of accesses based on the semantics of the algorithm: a task instance may access the local state instance on the same node or the global state by accessing all of the partial state instances, which introduces a synchronisation point.

2.7.3. Dataflow scheduling

Tasks in a dataflow graph can be either *scheduled* for execution or materialised in a *pipeline*. Each of these methods has different performance implications. Thus, some frameworks follow a more hybrid approach choosing one of the methods mentioned depending on the task placement (local or remote). For example tasks on the same node could be pipelined and tasks between nodes could be scheduled for execution.

Since tasks in stateless dataflows are scheduled to process coarse-grained batched data, such systems can exploit the full parallelism of a cluster but they cannot achieve low processing latency. For lower latency, batched dataflows divide data into small batches for processing, also known as microbatches, and use efficient, yet complex task schedulers to resolve data dependencies. There is a fundamental trade-off between the lower latency of small batches and the higher throughput of larger ones. Typically developers , that are bounded by that trade-off [89], pick the framework that matches each application needs.

To improve performance, *continuous dataflows* do not materialise intermediate data between nodes and adopt a streaming model with a pipeline of tasks. Thus achieving lower latency without the extra overhead of scheduling. In *iterative computation*, a way to improve performance is caching the results of the former iteration as input to the latter, as implemented by early systems such as HaLoop [12]. More recent frameworks [25, 36, 61, 87] generalise this concept by permitting iteration over arbitrary parts of loops. Similarly, IWOs inherit from *stateful dataflow graphs* (SDGs) the support for iteration explicitly by permitting cycles in the dataflow graph but use a scheduler instead of a pipeline. A key point here is because the scheduler is running in a different thread within the same JVM as the dataflow it introduces minimum impact while managing to achieve low latency and ensuring the strict SLOs of the LC dataflow tasks.

To sum up, the problem of high cluster utilisation and management and resource isolation is not new. Though, the enormous growth of cluster and cloud computing and the increased popularity of the modern latency critical web applications that combine big data analytics made it more important than ever. We propose *In-memory Web Objects* (IWOs), a unified model to developers for writing web applications that have the ability to serve data while using big data analytics. IWOs expose a common interface without affecting the programming model and developers could benefit from writing domain logic using already existing popular web frameworks. IWOs are extending stateful dataflow graphs that are expressive enough to model scalable web applications today. The state is exposed through a low latency, generic interface, in order to support both serving requests and data-analytics. IWOs further improve cluster resource utilisation and isolation by fine-grained dataflow scheduling. Scheduling resources efficiently between BE and LC tasks in a real-time and coordinated fashion is the key to maintaining the strict SLOs of serving requests in web applications while keeping the cluster resource utilisation high.

3. System Design

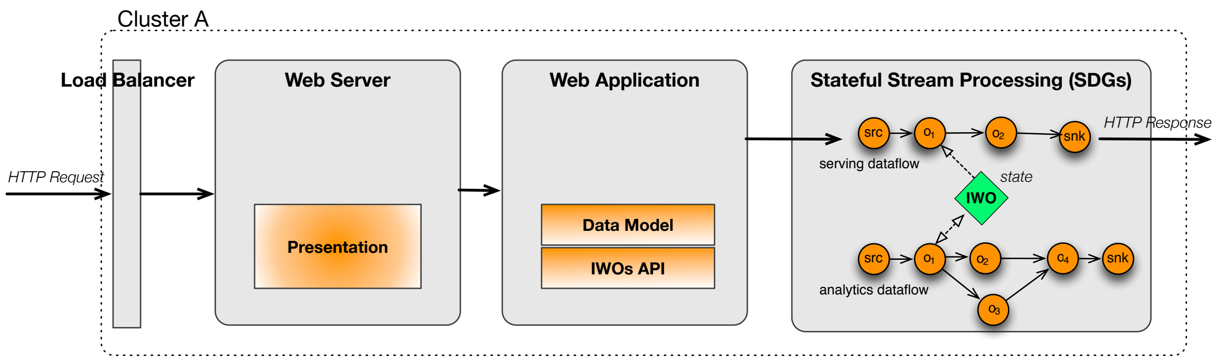


Figure 3.1.: Web application architecture using In-memory Web Objects (IWOs).

In-memory Web Objects model is targeting scalable web applications that provide low latency services to the clients and also support data-intensive analytics tasks. IWOs propose a unified model to developers for writing web applications that have the ability to combine both, in a finer and more resource efficient way, protecting the strict SLOs of request serving. In Section 3.1 we describe the IWOs model in more detail extending stateful dataflow graphs, while in Section 3.2 we show how we integrate the analytics and serving components as part of a dataflow. Finally, in Section 3.3 we describe the IWOs scheduling approach for efficient resource management and isolation.

The IWOs model is depicted in figure 3.1 as part of a web application architecture. Similar to a typical web application, client requests follow the load balancer and then the web server path until they reach the application specific logic. In the IWOs design, the web application is using the IWOs API to send the requests directly to the stateful stream processing engine in a pipelined fashion achieving low latency. The stateful stream processing framework implements the business logic of the application using the dataflow abstraction. In more detail, the online and offline logic of the web application is implemented as separate dataflows. The online request serving part is implemented by the *serving dataflow* graph which is responsible for processing the client requests and sending HTTP responses back with the minimum latency. The offline analytics task is implemented by the *analytics dataflow* graph which is implementing data-intensive algorithms such as collaborative filtering aiming high throughput. The analytics dataflow can be more complex than the serving, and its goal is to produce fresh results without affecting the SLO of the serving dataflow as we will explain later in Section 3.3. To achieve that, the analytics dataflow is computing new results continuously and updates the IWO, which is a form of state. The key observation here is that the serving dataflow can access the shared IWO and return responses to the clients without being affected by the analytics dataflow. Obviously the IWOs state will be as stale as the last analytics job which is in the order of seconds.

Comparing IWOs with the typical web application architecture, as depicted in Figure 2.1, besides the clean-slate design, IWOs benefit from the fact that both serving and analytics tasks are handled by a scalable stateful stream processing engine [26] and state can represent anything. Instead of running offline batch processing jobs in an isolated cluster, and store data asynchronously back to a data store we exploit the flexibility of IWOs model to handle the client requests in a pipelined fashion while generating fresh analytics data with a lower priority. The IWOs design is simpler, more flexible, and can handle the cluster resources in a more efficient and fine-grained way.

3.1. In-memory Web Objects Model

In-memory Web Objects use *stateful dataflow graphs* [28] (SDGs), which is a fault-tolerant data-parallel processing model, explicitly distinguishing data from state, built on top of SEEP [26]. It is a cyclic graph of pipelined data-parallel tasks, which execute on different nodes or threads and access local in-memory state. SDGs include abstractions for maintaining large state efficiently in a distributed fashion. When tasks can process state entirely in parallel, the state is partitioned across nodes. When this is not possible, tasks are given local instances of partial state for independent computation as explained in Section 2.7.2. Computation can include synchronisation points to access all partial state instances, and instances can be reconciled according to application semantics. Data flows between tasks in an SDG and cycles specify iterative computation. When tasks are pipelined, we can achieve lower latency, less intermediate data during failure recovery and simplified scheduling by not having to compute data dependencies. Tasks can be also replicated at runtime to overcome processing bottlenecks and stragglers, mostly for data-intensive workloads.

3.2. Unifying Serving and Analytics using IWOs

IWOs are also extending *stateful dataflow graphs* to provide distributed state abstractions that can now be accessible from the web application through IWO API. The goal of IWOs API is to simplify the translation of web application with analytics tasks to a number of dataflow representations (serving dataflow and analytics dataflow) that perform in parallel, with low latency. A web application can directly manipulate persistent dataflow state, which we extend to become In-memory Web Objects (IWOs). IWOs are computed in data-parallel fashion, either on-demand when a web request is handled, or asynchronously when representing previously computed, stored IWO data. They are implemented as in-memory state in a stateful distributed dataflow model [26]. Since IWOs are maintained in-memory but manipulated in a data-parallel fashion, they can satisfy the requirements of both offline and online data processing in web applications.

3.3. Towards efficient Dataflow Scheduling with IWOs

By default, the operators of an SDG are not scheduled for execution but the entire SDG is materialised, i.e. each task is assigned to a number of physical nodes. For IWOs though, it is important to prioritise the serving dataflow and protect its strict SLOs, while maximizing overall utilization of the cluster. High utilisation can be achieved by task colocation and as explained in Section 2.3.1, it can lead to serious interference. To avoid that, dataflow tasks in IWOs are using a custom scheduler.

In more detail, IWOs rely on a centralised scheduler, depicted in Figure 3.2, to allocate CPU

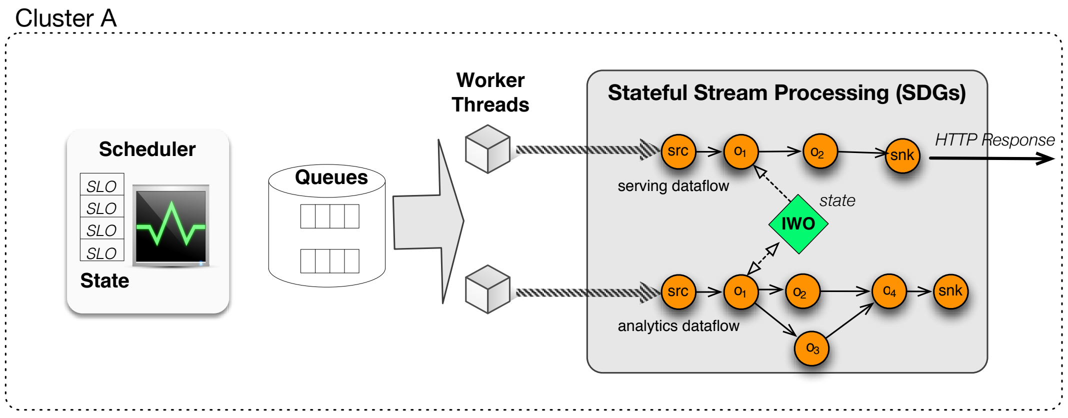


Figure 3.2.: In-memory Web Objects (IWOs) scheduler for serving and analytics dataflows.

time for both the LC and BE tasks. The allocation is achieved by 'feeding' a number of queues with different individual priorities. For example, if we have two cores we could set up two queues, and each queue would then have 50% of the total capacity for processing tasks. Then we define one queue for the serving dataflow and the other one for the analytics dataflow. A number of worker threads handle reading and running the next task from these queues. Usually, the number of threads responsible for the dataflow tasks is proportional to the number of physical cores for better isolation, but it can vary depending on the number of multiplexing we want to achieve.

IWOs scheduler, which is running in a separate thread, is also supporting the notion of LC applications like serving that can ask for multiple-slots in bursty periods. This is a very useful feature for latency critical applications in order to avoid SLO violations. The scheduler is aware of the application SLO goals and by monitoring the runtime performance of the dataflows sets future goals and takes appropriate action with *low latency impact* since it resides in the same JVM. When SLOs are close to being violated the scheduler is reactively scheduling more LC tasks in all queues slowing down the analytics tasks effectively increasing the resources the LC dataflow can use. The resources consumed by the LC tasks are naturally accounted for against the queue capacity. When the serving part is not highly utilised, the scheduler has mechanisms to rebalance BE and LC tasks in the queues, increasing the resource efficiency.

4. Resource Isolation

Maximising cluster resource efficiency is one of the primary goals of IWOs and task colocation is usually the only way to achieve that in a cluster. In this Chapter, we justify our decision to implement a custom task scheduler described above instead of using an already existing solution. LC tasks with their strict SLOs often underutilise resources in the machines they reside, and launching multiple BE tasks in the same machines is a promising way to improve utilisation. The main challenge in this approach is resource interference on shared resources. Mechanisms like cgroups [58] can be used to isolate a particular resource but due to cross-resource interference solutions like Linux containers [1] that can isolate multiple resources, are more adequate. While other studies [52] mostly focused on the poor efficiency of the isolation mechanisms in this section we focus on the flexibility. In a dynamic environment, isolating resources is important but the ability to shift them from one service to another in a robust way at runtime is more challenging. For example, in a web application deployment, when the serving part is idle, the resources could be shifted to the analytics part to compute fresher results.

4.1. Linux Containers

The first mechanism we evaluate is Linux Containers (LXC), providing a variety of isolation mechanisms for the application running in them. LXC implement kernel abstractions and also provide an API to restrict the container specific resources. To make the experiment more realistic, we created a container running a real web application similar to Spotify, further explained in Chapter 5, using up to 2 cores and 2 Gigabytes of RAM. Every time the container starts, an initialization script starts the web application which is listening to a preconfigured HTTP port to serve clients. In order

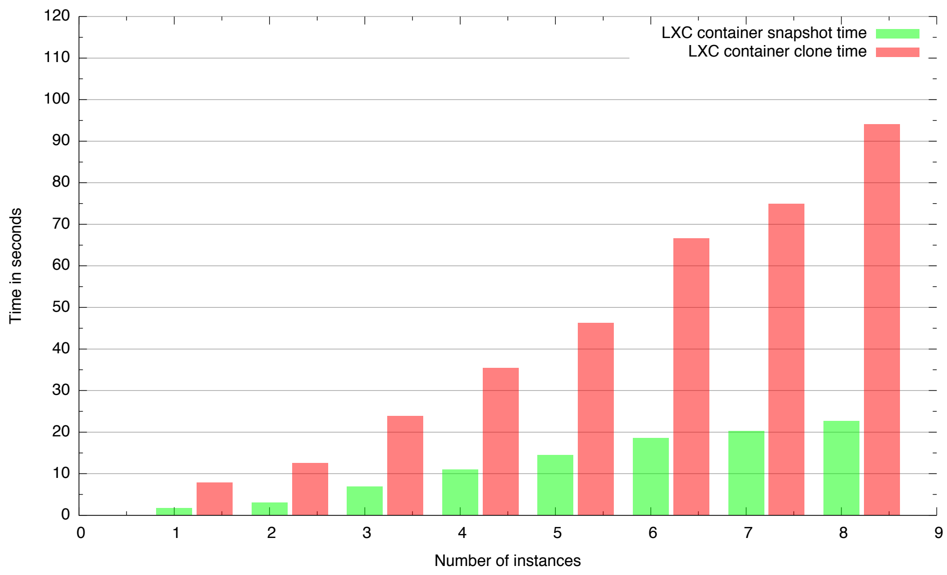


Figure 4.1.: Web application deployment duration using Linux Containers.

to shift resources using containers, we have to make a copy of the running instance to maintain the state and start the new one with updated resource scheduling policies. We measure the duration of the copy process, evaluating both container clone and snapshot techniques. A clone copies the root filesytem from the original container to the new one. A snapshot uses the backing store's snapshot functionality to create a subtle copy-on-write snapshot of the original container. Snapshot clones require the new container backing store to support snapshotting which in our case was ext4.

In Figure 4.1 we observe that starting a new web container using cloning takes 7.8 seconds while using snapshotting takes 1.7 seconds. If we want to multiplex instances, the amount of time needed to make multiple copies using cloning increases linearly, with *four* copies taking 36 seconds and *eight* copies taking 95 seconds. While snapshotting is faster, requiring 11 seconds for *four* instances and 22 seconds for *eight* instances, we are still orders of magnitude away from the SLOs we want to provide. Usually, web applications provide SLOs in the order or *tens* of milliseconds [42].

4.2. Mesos Framework

The second mechanism we evaluate is Apache Mesos [35]. Mesos is a two level scheduling framework with a plethora of extensions like *Marathon*. Marathon runs alongside Mesos and provides a REST API for starting, stopping, and scaling applications. The developer provides the application archive path, resource demands, and the startup command. Then Marathon invokes an executor to launch the task in a machine in the cluster. The executor in a Linux platform is using cgroups to provide CPU and memory isolation.

For the evaluation, we implemented a Java REST client. The client makes Marathons REST API calls for different instance numbers and measures the time until the application becomes available by sending an HTTP request to the new application instance. We deploy the same web application as the experiment above requesting 2 CPU cores and 2 Gigabytes of RAM. Figure 4.2 depicts how long deployment takes for our web application in red, and how long application launch takes in green. Since Mesos is not using by default containers, we can not use an initialization script, and this is the reason why there are two separate bars that could be summed instead. The time spent

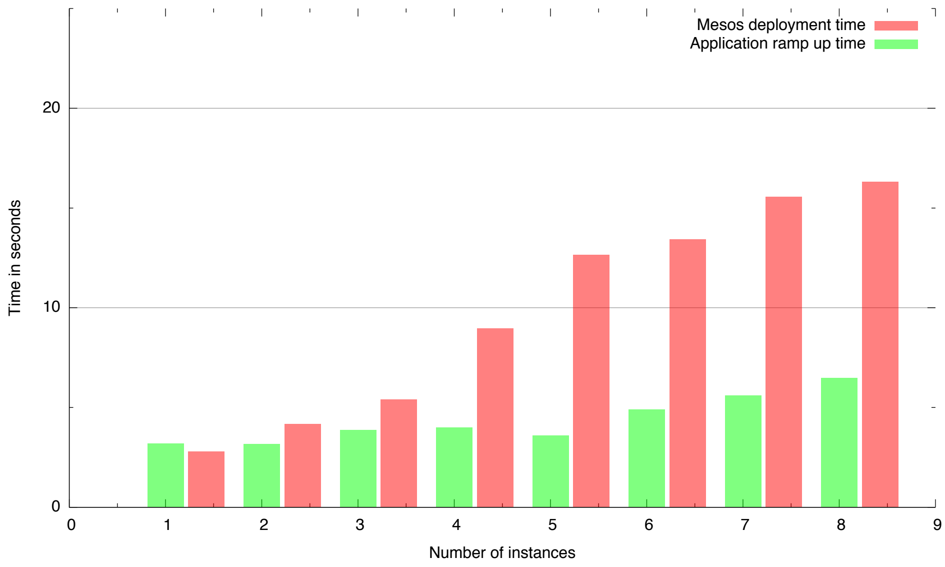


Figure 4.2.: Web application deployment duration using Apache Mesos.

deploying *one* instance is 2.8 seconds and 3 seconds for the application initialisation need to be added to this number. This time increases to 8.9 plus 3.5 seconds for *four* instances and to 16 seconds plus 6 for *eight* instances. We make two observations here: First, while the deployment time increases linearly, the ramp up period is almost stable, and that has to do with the deployment type of Mesos. Mesos is using a cluster of machines and most of the time is spent transferring the application archive to these machines. The service instantiation itself is usually stable and, in fact, is the one in the machine with the slowest deployment time. The second observation is that the deployment time is much faster than container cloning. This is due to the fact that Mesos is distributing these tasks in multiple machines compared to one machine in Linux Containers and is by default using executors and cgroups to achieve application-level isolation that introduces less overhead. Even with the Mesos executor model, though, we achieve deployment latency in the order of seconds which is not suitable for dynamic, fine-grained resource management.

5. Case Study: Exploiting IWOs in a Music Web Application

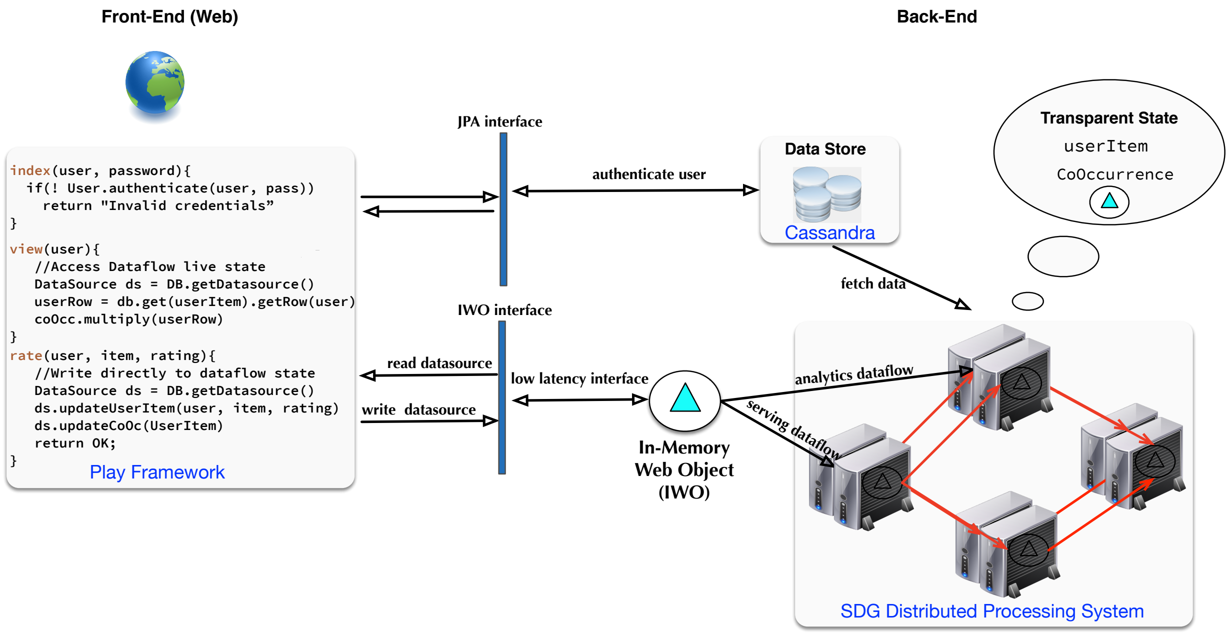
The IWOs implementation uses Stateful Dataflow Graphs [27] and SEEP [26] as a baseline. It preserves the SDG model abstractions for distributed state and mechanisms for efficient failure recovery, state management and scaling. IWOs provide an API that can be used by web frameworks like Play for Java [78] to manage dataflows using imperative code. Online LC tasks like serving and offline BE tasks like machine learning algorithms can be implemented as dataflows in Java code and attached to web application logic exploiting IWOs API.

As a case study, we used Play Framework to implement *Play2SDG*. Play2SDG is a web application implementing business logic similar to Spotify [90], where users login to their accounts, listen to songs, rate songs and group them into playlists. Then, the service itself handles new songs recommendations for the users according to other user preferences using algorithms like collaborative filtering [76]. Play2SDG was implemented in approximately 5k lines of code, mainly written in Java, Scala and Javascript while the collaborative filtering implementations and the IWOs abstraction were implemented as a separate pure Java module in approximately 8k lines of code. Section 5.1 describes the internals of Play2SDG implementation using Play MVC model for the web application and Apache Cassandra [47] as a storage back-end. Sections 5.1.1, 5.1.2 and 5.1.4 describe the application implementation in more detail and the modifications we made to the application code in order to support IWOs. Finally, in Section 5.2 we compare the two different approaches using Apache JMeter as load generator [39] and demonstrate the efficiency of our scheduling approach.

A general overview of *Play2SDG* web application components and interactions is shown in Figure 5.1, while an overview of a typical web application's components *without* IWOs is depicted in Figure 5.2.

5.1. Play2SDG Web Application

Play is considered a fully stateless and request-oriented framework. In other words, all HTTP requests sent by the clients follow the same path. First, an HTTP request is typically received by a load balancer that forwards the request to the framework. The router component is responsible for finding the most appropriate route able to accept this request. Router component implements the mapping between the request path and the application controller actions. After the mapping succeeds, the corresponding action method is invoked. The action method contains most of the application code with the business logic. When a complex view needs to be generated, a template file is rendered. The result of the action method including HTTP response code and the content is finally written as an HTTP response.

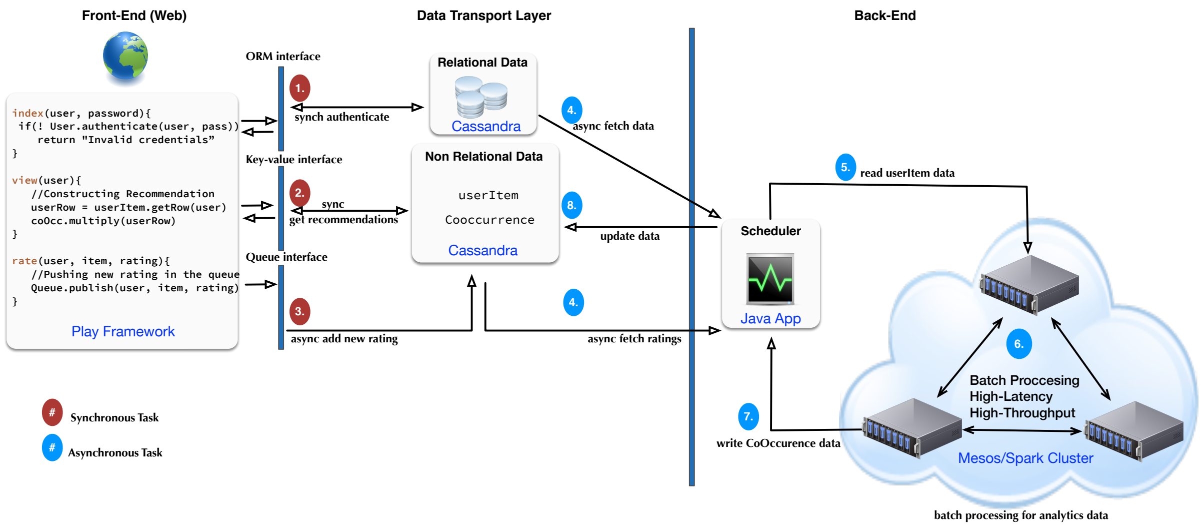


```
index(user, password){
  if(! User.authenticate(user, pass))
    return "Invalid credentials"
}
view(user){
   //Access Dataflow live state
   DataSource ds = DB.getDatasource()
   userRow = db.get(userItem).getRow(user)
   coOcc.multiply(userRow)
}
rate(user, item, rating){
   //Write directly to dataflow state
   DataSource ds = DB.getDatasource()
   ds.updateUserItem(user, item, rating)
   ds.updateCoOc(UserItem)
   return OK;
}
```

Figure 5.1.: Dataflow-based web application implementation using In-memory Web Objects.

5.1.1. Application Controller

Play2SDG router component is summarised in table 5.1 below. The main functionality we want to support is similar to a typical web music service. A user should be able to login, logout and authenticate. Login action for instance is an HTTP GET method, which returns the rendered page to the user, while authenticate is a POST method pushing the user credentials to the server for validation. Each user can manage their own playlists meaning they can create, delete and rename playlists. As is usually the case, they can also add songs in their playlists as well as rate them based on their experience. The rate action is a POST method, since there is no need to render a new page for every new rating. Javascript functions are responsible for updating the user page after the new rating without extra server-side overhead.



```
index(user, password){
 if(! User.authenticate(user, pass))
    return "Invalid credentials"
}

view(user){
   //Constructing Recommendation
   userRow = userItem.getRow(user)
   coOcc.multiply(userRow)
}

rate(user, item, rating){
   //Pushing new rating in the queue
   Queue.publish(user, item, rating)
}
```

Figure 5.2.: Typical web application implementation using Play framework and Cassandra.

Page	**Controller Action**
GET /	controllers.App.index()
GET /login	controllers.Login.index()
POST /login	controllers.Login.authenticate()
GET /logout	controllers.App.logout()
POST /rate/:playlistid/:sid	controllers.App.rate(playlistid: UUID, sid: String)
DELETE /playlist/delete/:pid/:sid	controllers.App.deletePLSong(pid: UUID, sid :String)
GET /recommend	controllers.App.getUserRecommendations()
POST /playlist/:playlistid	controllers.PLController.add(playlistid: String)
DELETE /playlist/:playlistid	controllers.PLController.delete(playlistid: java.util.UUID)
PUT /playlist/:playlistid	controllers.PLController.rename(playlistid: java.util.UUID)

Table 5.1.: Play2SDG route table.

For every successful page request the router component is invoking a controller action, listed in the right column of table 5.1. Algorithm 1 describes part of the Play2SDG application controller implementing these actions. The first action that is typically called is the index. The index method is using the Play authentication API to make sure a user is logged in, subsequently queries the database back-end to get user playlists and songs and, finally, calls the appropriate view renderer to return the home page to the user. The view rendering is explained in more detail in Section 5.1.2. From the rendered home page a user can rate new songs. The rate action method receives playlist and track IDs as arguments and asynchronously stores them in the database. All the rating data will be periodically fetched by a batch processing analytics job that is generating useful user data like song recommendations. The analytics jobs are usually part of a separate cluster to achieve isolation as we discussed in Section 2.3.2. We implemented our recommendation task using the Spark [88] data parallel framework. The analytics data are stored in a scalable key/value store and a user can access them trough the recommendation page. The recommendation page invokes the getUserRecommendations method, which is fetching the latest results and some job statistics.

Algorithm 2 describes the modifications we made to our application controller in order to support IWOs. In more detail, the rate method is now pushing the new ratings to an analytics dataflow that is computing user recommendations, also depicted in Figure 5.6. The dataflow source is reading the new rate requests and a custom serializer is implemented to ensure that data are encoded and decoded correctly. The GET recommendations method is invoking the serving dataflow requesting the latest recommendations for the user who triggered the request. The rest of the controllers functionality remains the same. The IWOs API was mainly used in parts that are coupled with the computationally intensive recommendations task. Moreover, we maintain the Cassandra backend for data storage durability.

5.1.2. View

From the application controller the responses containing the data are handled by the renderers. The view renderers are responsible for transforming the model into a form suitable for interactions, typically a user interface. Multiple views can exist for a single model, each serving a different purpose. In a web application, the view is usually rendered in a web format like HTML. For simplicity we implemented the view renderers using Play Scala template engine which is more expressive and compact than pure HTML code. Algorithm 3 shows a Scala implementation of Play2SDG user home page. The template parameters are the user object, the number of total

Algorithm 1: Play2SDG Application Controller implementation without IWOs.

```

public class App extends Controller {

   private static EchoNestAPI en;

   @Security.Authenticated(Secured.class)
   public static Result index() {
   User.find.byId(request().username()));
      return ok(views.html.index.render(PlayListController.findExisting(request().username()),
          PlayListController.getTracksPage(0), Login.findUser(request().username()),
          CassandraController.getCounterValue("tracks") ) );
   }

   @Security.Authenticated(Secured.class)
   public static Result getNextPage(String lastcurrentPageTrack){
      return ok(views.html.index.render(PlayListController.findExisting(request().username()),
          PlayListController.getnextTracksPage(lastcurrentPageTrack), Login.findUser(request().
          username()), CassandraController.getCounterValue("tracks") ) );
   }


   @Security.Authenticated(Secured.class)
   public static Result getUserRecommendations(){
      Recommendation userRec = CassandraController.getUserRecc(request().username());
      Stats jobStats = CassandraController.getSparkJobStats();
      return ok(views.html.ratings.cf.render(userRec, jobStats,controllers.CassandraController.
          findbyEmail(request().username()) ));
   }

   @Security.Authenticated(Secured.class)
   public static Result rate(UUID playlistid, String track_id){
      Track found = PlayListController.findByTrackID(track_id);
      PlayListController.addSong(playlistid, found);
      return ok(views.html.index.render(PlayListController.findExisting(request().username()),
          PlayListController.findAllSongs(), Login.findUser(request().username()),
          CassandraController.getCounterValue("tracks") ) );
   }

   public static Result logout() {
      session().clear();
      flash("success", "You've been logged out");
      return redirect(routes.Login.index());
   }
}
```

Algorithm 2: Play2SDG Application Controller implementation with IWOs.

```

public class App extends Controller {

   @Security.Authenticated(Secured.class)
   public static Result getUserRecommendations(){
      Stats jobStats = CassandraController.getSdgJobStats();
   Recommendation userRec = Dataflow.getDataServingInstance().getRec(request().username());
      return ok(views.html.ratings.cf.render(userRec, jobStats,controllers.CassandraController.
          findbyEmail(request().username()) ));
   }

   @Security.Authenticated(Secured.class)
   public static Result rate(UUID playlistid, String track_id){
      DataSource ds = Dataflow.getDataSourceInstance();
      ds.sendData(request());
      return ok(views.html.index.render(PlayListController.findExisting(request().username()),
          PlayListController.findAllSongs(), Login.findUser(request().username()),
          CassandraController.getCounterValue("tracks") ) );
   }

}
```

tracks and a list of user-specific playlists and tracks. Template is like a function and the function parameters are declared at the top of the file. HTML code inside the template defines the object classes and layout and the '@' character represents a dynamic statement that can invoke application methods, that are accessible through the router table.

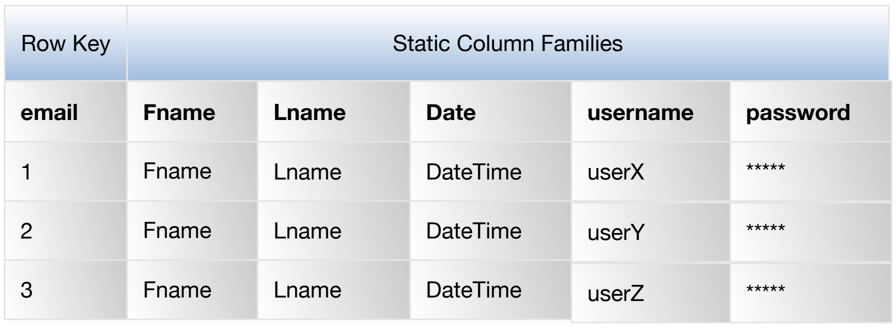
Algorithm 3: Web Application home page Front-End using Play framework and Scala.

```
@(ratings: List[PlayList], songs: List[Track], user: User, totalTracks : Integer)

@main(ratings, user) {

    <header>
        <hgroup>
            <h1>Dashboard</h1>
            <h2>Songs</h2>
        </hgroup>
    </header>

    <article class=”tasks”>

                <div class=”folder” data-folder-id=”Songs List”>
                    <header>
                        <h3>Most Recent Tracks List - Total Tracks (@totalTracks) </h3>
                    </header>
                    <ul class=”list”>
                        @songs.map { song =>
                            @views.html.elements.song(song)
                        }
                    </ul>
                </div>

                <div>
                    <a href=”@routes.Application.getNextPage(songs.last.getTrack_id())”>Next Page</a>
                </div>

    </article>

}
```

5.1.3. Data Model

Play2SDG defines some main entities for its data models such as Users, Playlists, Tracks, Ratings, Recommendations and Statistics. We are exploiting Java application programming (JPA) interface to persist relational data in the database. Play2SDG is using Apache Cassandra as a data store. Cassandra is exposing a column-oriented, schema-optional data model explained in Appendix A in more detail.

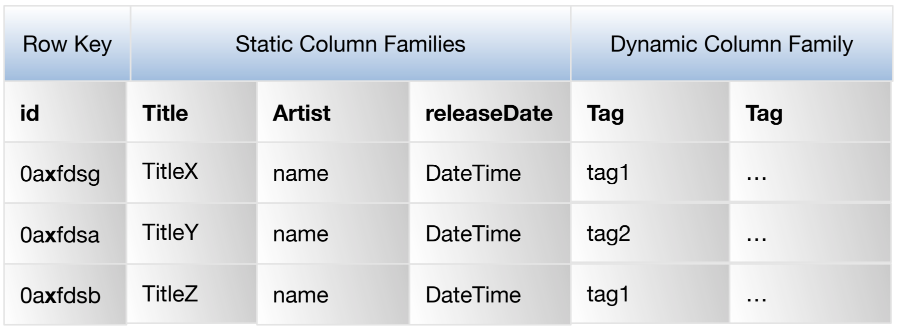
Users: In the User column family, each row represents a user as depicted in Figure 5.3. Each user has a unique email that is the row key and the rest of the columns store information such as first name, last name, registration date, username and password.



Row Key	Static Column Families				
email	**Fname**	**Lname**	**Date**	**username**	**password**
1	Fname	Lname	DateTime	userX	*****
2	Fname	Lname	DateTime	userY	*****
3	Fname	Lname	DateTime	userZ	*****

Figure 5.3.: Play2SDG User data model.

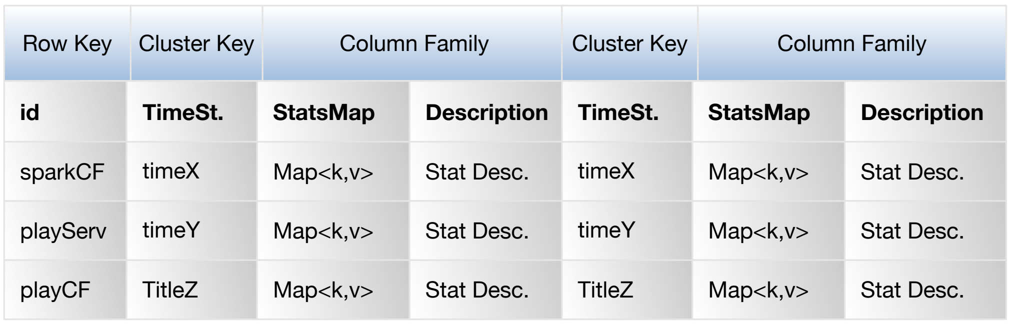
Tracks: In the Tracks column family each row represents a track, and there is a distinction between static columns and dynamic columns. Static track fields are the ones that we know their size while dynamic are the ones that we don't. We want our data to be as close to reality as possible, so we used the million songs dataset [9] which is a collection of tracks from real music providers and associated with Spotify IDs. The dataset contains 943,347 unique tracks all of which have title artist and releaseDate. 505,216 of them have at least one tag, and all of them create 8,598,630 track-tag pairs. Cassandra has the notion of dynamic wide rows as depicted in Figure 5.4, which is a perfect match for non-static track tags.



Row Key	Static Column Families			Dynamic Column Family	
id	**Title**	**Artist**	**releaseDate**	**Tag**	**Tag**
0axfdsg	TitleX	name	DateTime	tag1	...
0axfdsa	TitleY	name	DateTime	tag2	...
0axfdsb	TitleZ	name	DateTime	tag1	...

Figure 5.4.: Play2SDG Track data model.

Playlists: Playlists column family is practically a list of Tracks with the only difference that each row has two identifiers: user-id and playlist-id.
Recommendations: Recommendations column family is similar to the Playlist described above. The main difference is each recommendation has one row-key which is the user-id.
Statistics: The statistics column family handles storing analytic jobs statistics. Each row represents a specific job, and the row key is the job name. These jobs are usually reoccurring, so the statistics are in a form of time series as depicted in Figures 5.5. The dynamic column family feature is used to cluster the wide, growing rows by their timestamp.



Row Key	Cluster Key	Column Family		Cluster Key	Column Family	
id	**TimeSt.**	**StatsMap**	**Description**	**TimeSt.**	**StatsMap**	**Description**
sparkCF	timeX	Map<k,v>	Stat Desc.	timeX	Map<k,v>	Stat Desc.
playServ	timeY	Map<k,v>	Stat Desc.	timeY	Map<k,v>	Stat Desc.
playCF	TitleZ	Map<k,v>	Stat Desc.	TitleZ	Map<k,v>	Stat Desc.

Figure 5.5.: Play2SDG Statistics data model.

5.1.4. Analytics Back-End

To provide user song recommendations for *Play2SDG* web application we implemented two different version of collaborative filtering (CF) algorithm. The first implementation is using Spark as a typical parallel processing framework and the second one is a dataflow implementation using SEEP that is part of the IWOs integration with Play framework. Collaborative filtering is a machine learning algorithm that outputs up-to-date recommendations of items to users based on previous item ratings.

Spark Collaborative Filtering

For the Spark CF implementation, we used MLlib library that implements a number of machine learning algorithms. It supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries. MLlib uses the alternating least squares (ALS) algorithm to learn these latent factors, and it is configured to run 10 iterations. The full algorithm implementation can be found in Appendix 5 and 6.

SDG Collaborative Filtering

For the dataflow implementation of CF we used an algorithm similar to the one described by Fernandez et al. [28] (more details can be found at Appendix B). Figure 5.6 describes the interactions between the dataflow tasks, state elements and the data streams. The algorithm maintains state in two data structures. The matrix userItem stores the ratings of items made by users, and the co-occurrence matrix coOcc records correlations between items that were rated together by multiple users. The *new rating* function first adds a new rating to userItem and then incrementally updates coOcc by increasing the co-occurrence counts for the newly-rated item and existing items with non-zero ratings. The function *rec request* takes the rating vector of a user, userRow, and multiplies it by the cooccurrence matrix to obtain and return a recommendation vector result.

We have to note here that *new rating* and *rec request* have different performance goals. The former is part of the analytics dataflow in our web application and must achieve high throughput while the later is part of the serving dataflow and must serve requests in low latency. In Play2SDG implementation, the analytics dataflow is continuously fetching ratings data from the Cassandra backend to produce fresh recommendations while the serving dataflow handles client requests directly from the Play web server.

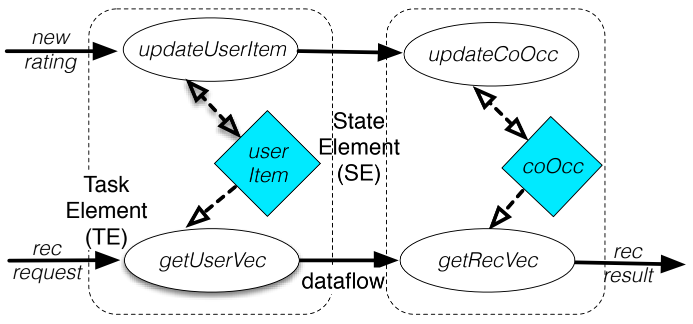


Figure 5.6.: Stateful dataflow graph for CF algorithm.

5.2. Experimental Results

As a case study, we evaluate Play2SDG in a private cluster consisting of 5 machines running Ubuntu 12.04.5 LTS 64 bit. The machines have 8 core CPUs, 8 GB memory, and a 1TB locally mounted disk. The Apache Spark version used is 1.1.0, Apache Mesos is 0.22.1, Nginx is 1.1.19 and the Cassandra database is 2.0.1.

Our load generator is Apache JMeter 2.13, which can be configured to produce a specific user access pattern in web applications and measure performance [39]. For our experiments the specific functional behaviour pattern is: (i) user login, (ii) navigate through the home page displaying the

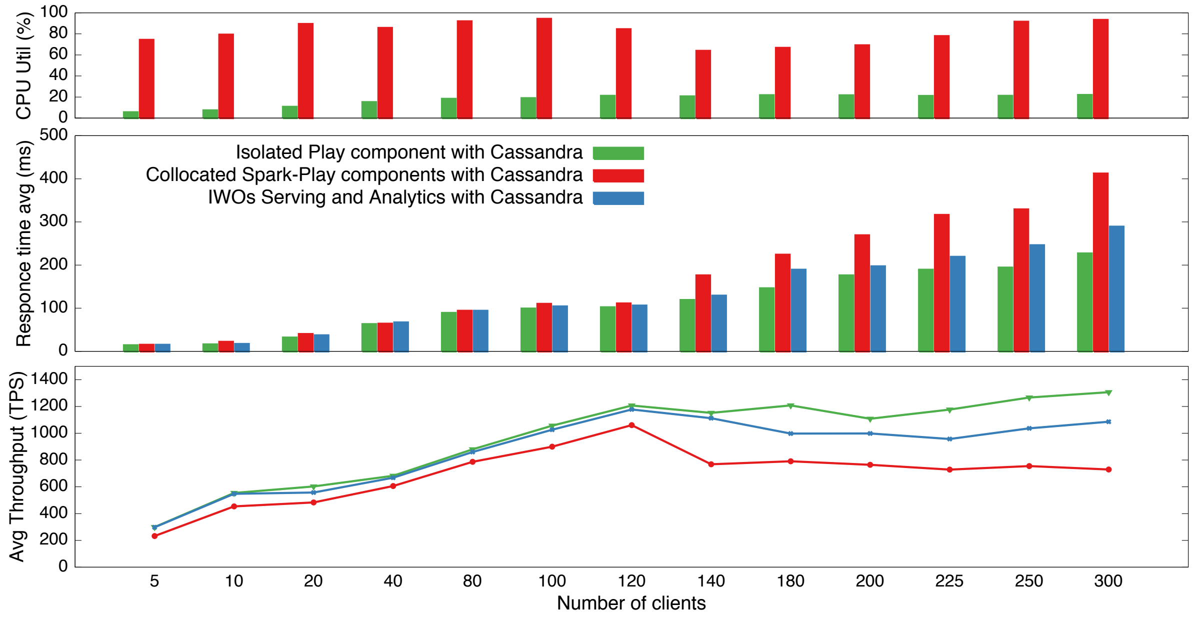


Figure 5.7.: Average response time and throughput using Play-Spark Isolated, Play-Spark Colocated and In-Memory-Web-Objects.

top 100 tracks, (iii) visit the page with the latest recommendations and (iv) user logout. In all our experiments, JMeter is running in a dedicated machine inside the local network.

We deploy Play2SDG with Cassandra and Spark using Mesos cluster manager. Mesos is running 1 master instance and 3 slave instances, managing 4 nodes in total. Nginx proxy is configured as proxy to redirect requests to Play framework efficiently. Play framework, Cassandra and Spark are configured to use up to 2 cores and 2GB of memory each through the Mesos API. Spark is set up in cluster mode. In cluster mode, the application is coordinated by the SparkContext object (also called driver program) depicted in Figure 5.11. SparkContext is configured to use Mesos cluster manager that handles allocating resources for the executors on nodes in the cluster. Executors receive the recommendation application packaged as a JAR file and coordinated by the driver and run tasks in a best effort (BE) fashion. The best effort tasks consume as many resources as they can on the available hosts and can usually cause interference when colocated with latency critical tasks.

We first quantify the performance impact of colocating Spark executors on a single machine running Play2SDG web application and its proxy with and without IWOs. Figure 5.7 depicts the average throughput and the average response time in milliseconds of the typical JMeter workload described above, over an increasing number of clients. We compare the Play application when isolated, meaning that no Spark tasks are allowed running on the same machine, with Play application when colocated with Spark tasks. In both cases, the resources given by Mesos that were dedicated to Play application remained the same. We observe that for a number of clients smaller than 120 the performance of both isolated and colocated applications is identical. When the number of client increases above that point, resources are closer to saturation, and the colocated application performance drops dramatically. The main reason is resource interference, since the isolation mechanisms provided by the OS and Mesos are inadequate. As a result, for 300 clients the isolated application can achieve 79% better throughput (1300 TPS over 730 TPS) and 82% lower response time (227ms over 412ms) compared to the colocated one.

We next evaluate the performance of Play2SDG web application using IWOs with pipelined

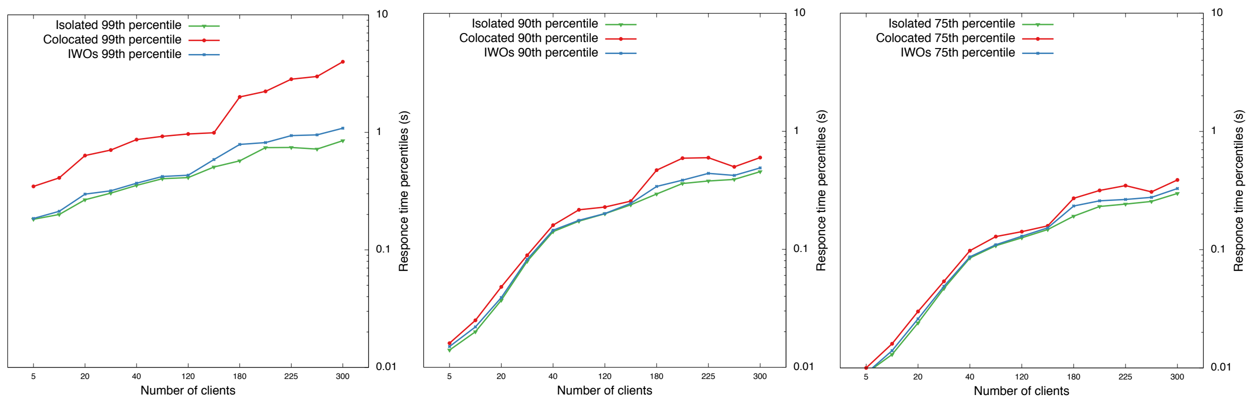


Figure 5.8.: Response time percentiles using Play-Spark Isolated, Play-Spark Colocated and In-Memory-Web-Objects.

tasks. To make the results comparable we configure the application JVM to use the same resources as above and the dataflow to reside in a different JVM in the same machine. We observe that increasing the workload above 140 clients can hurt performance again due to increased resource demands and interference, but the drop is much smaller. For instance, with 300 clients the average throughput decreased by 19% and the average latency increased by 25%. The results clearly show that we can achieve low serving latency using stateful dataflow graphs, which effectively means that when the serving part is under-utilised we could use the resources to run more analytics. Moreover, even though the pipelined dataflows do not introduce too much resource interference since they are isolated in their JVM, and they are not as greedy as batch BE tasks, we firmly believe that we can achieve even better performance using a scheduled version of the dataflow tasks inside the same JVM.

It is worth mentioning here that in order to achieve the maximum throughput we applied several optimisations both in the operating system and the application level. Nginx was further optimised by disabling logging to reduce I/O contention and by increasing the number of workers to 1k per core we made sure it is not the bottleneck. Play framework by default is using small thread pool to handle clients that fast became a bottleneck. It was configured to use 50 threads which seemed to be sufficient for our experiments. JMeter logging was disabled and was run by command line to reduce the extra user interface overhead. At operating system level, we had to increase the open files limit which was a limit for the database. Another interesting issue we faced is running out of HTTP addresses in the JMeter side. By default, the operating system was creating a new one for every client page visit which soon reached a limit. We addressed this issue by enabling fast HTTP address recycling.

To get a deeper understanding of the real response times users observe in one of these application deployments, we also calculate the 99th, 90th and 75th response time percentiles depicted in Figure 5.8. Usually, modern web applications target 99th percentile below 1-second meaning that out of 100 clients 99 should experience latency lower than 1 second. As users tend to visit more than one page in a web application and as web applications tend to add more features increasing

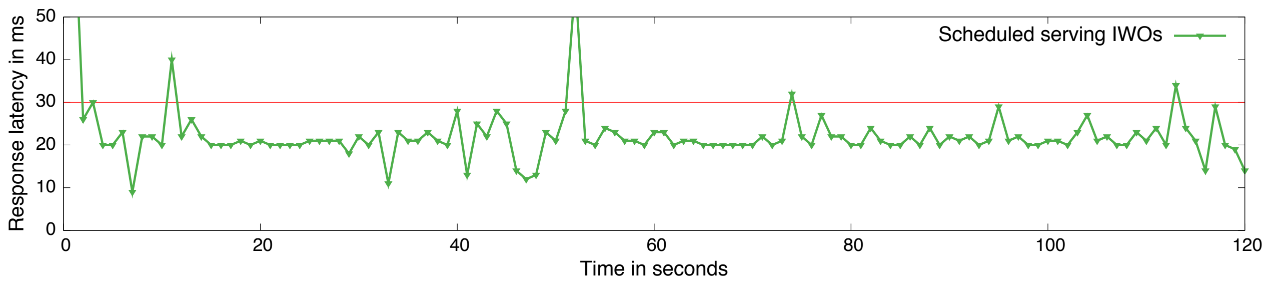


Figure 5.9.: Serving Response latency using Scheduled IWOs.

the number of requests on each page, one user experiencing latency higher than the 99th percentile is not uncommon. Figure 5.8 shows that collocating Play with Spark seriously affects the 99th percentile which is more than double for any client size. 90th and 75th percentiles are mostly affected when the client size is above 120, which is close to the resource saturation point. IWOs implementation remains close to the isolated case with latency close to 1 second even for a 300 client load. The results mostly suggest that we can indeed achieve low serving latency using dataflows while running analytics tasks in web applications and that resource interference inside the JVM is less aggressive. Scheduling these tasks and prioritising the LC serving dataflow, as described below, would further optimise these results that could be (in some cases) even faster than the baseline.

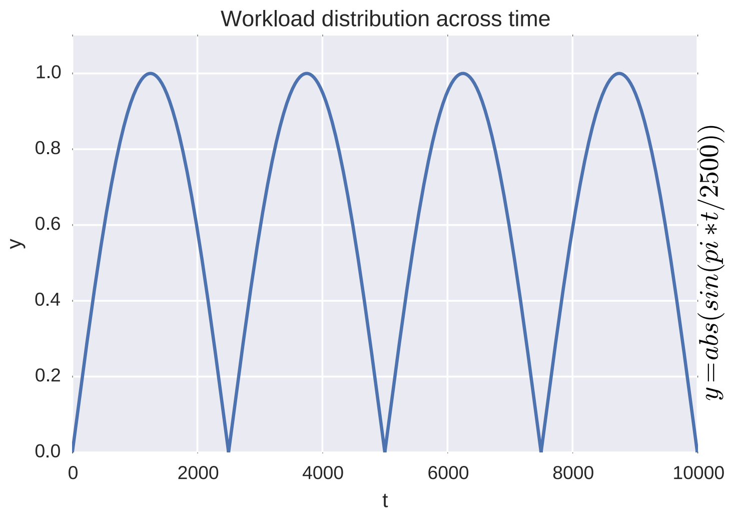


Figure 5.10.: Workload distribution produced by the equation:

$$y = |\sin(pi * t/2500)|$$

To evaluate our scheduler approach we implemented a simplified version according to the design in Figure 3.2. The scheduler was running in a single machine in the cluster with two separate queues, one for analytics tasks and the other one for serving tasks. One consumer thread was devoted to the analytics dataflow, which consisted of single operator calculating 20 pi digits. Another thread was devoted to the serving dataflow, which was again a single operator deserialising the request and returning a response. The scheduler was running in its own thread and was continuously monitoring the latency of the serving response latency. As a load generator we implemented a synthetic workload according to the sin equation depicted in Figure 5.10. The scheduler was continuously sampling from the equation to decide how many tasks are going to be produced. Since the sampling rate depends on the processing power there is no one-to-one mapping in the time axis between Figures 5.10 and 5.8. The result of the equation was multiplied with 100k tasks to produce enough stress load and these tasks were queued for processing. The response time SLO was set to 30ms in the scheduler state. When the serving response time went above 30ms the scheduler slowed down the analytics dataflow by pre-emptying. On the other hand, when the latency was below 21 ms the scheduler added more analytics tasks to the dataflow to produce more results. Figure 5.8 shows the latency in the LC serving part over a period of 120 seconds. We observe that during the whole period of the experiment the scheduler manages to maintain the strict SLO and can easily re-adapt to the dynamicly changing request number. When a single request is facing high tail latency the fine-grained scheduler reactively slows down the serving part, shifting more resources to the LC part, and reactivates the analytics when the response latency is low enough. IWOs can, therefore, efficiently protect strict SLOs and manage resource isolation in a finer and more robust way than existing solutions, while achieving high resource utilisation which was not a requirement in already existing solutions.

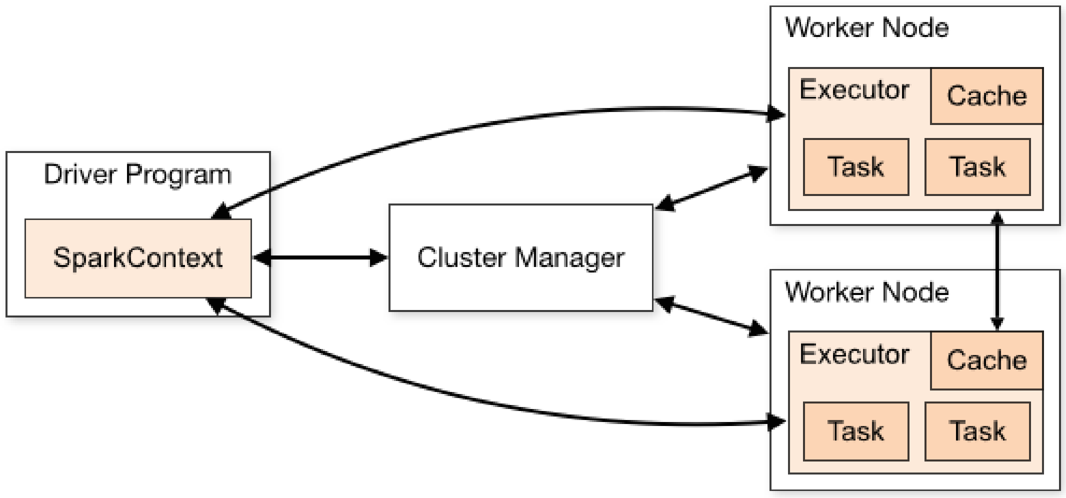


Figure 5.11.: Spark Distributed/Clustered mode overview.

6. Conclusions

In this thesis we described *In-memory Web Objects* (IWOs), a unified model for developers to write web applications that have the ability to serve data while using big data analytics. IWOs expose a common interface without affecting the programming model and developers could benefit from writing domain logic using already existing popular web frameworks. We are extending stateful dataflow graphs (SDGs), which are expressive enough to model scalable web applications today. The state is exposed through a low latency, generic interface, in order to support both serving requests and data-analytics. IWOs further aim to improve cluster resource utilisation and isolation by fine-grained scheduling. Scheduling resources between BE and LC tasks efficiently, in a real-time and coordinated fashion, is the key to maintaining the strict SLOs of serving in web applications. Experimental results using our case study application, implemented using Play framework [78], indicate that using IWOs we can achieve serving latency lower than 1sec at the 99th percentile while also running data analytics. Furthermore, our scheduling approach evaluation suggests that by fine-grained task management we could safeguard serving from SLOs violations with extremely low latency impact.

6.1. Future Work

Efficient *distributed* scheduling of BE analytics and LC serving tasks is an important research area that we plan to focus on next. While the approach we presented performs well on a single machine, scheduling dataflow tasks in a distributed dataflow is more demanding introducing orthogonal challenges such as optimal placement, low latency decision making and reactive scheduling in SLO violations.

Additionally, we plan on further investigating the automatic conversion of a web application in an SDG. We are considering extracting information about the data model and the processing associated with it in an extra offline step. An approach towards this direction would be extending our framework to extract all the code connected with back-end data from the *Model* component and use it to model state. The application's *Controller* code will then be converted into in an statefull dataflow graph using a special purpose compiler and will be deployed in a distributed data-parallel framework like Seep [26]. The SDG could then be deployed and automatically scale, providing fault tolerance without requiring further input from the user.

Finally, since IWOs provide just an abstract programming model they should not be restricted to a single dataflow processing framework. IWOs abstractions can be implemented for other stateful stream processing frameworks, like Flink [3], as well and investigate how they integrate and potentially solve other types of challenges in areas such as intrusion detection, graph processing etc.

Appendices

A. Column Data Model

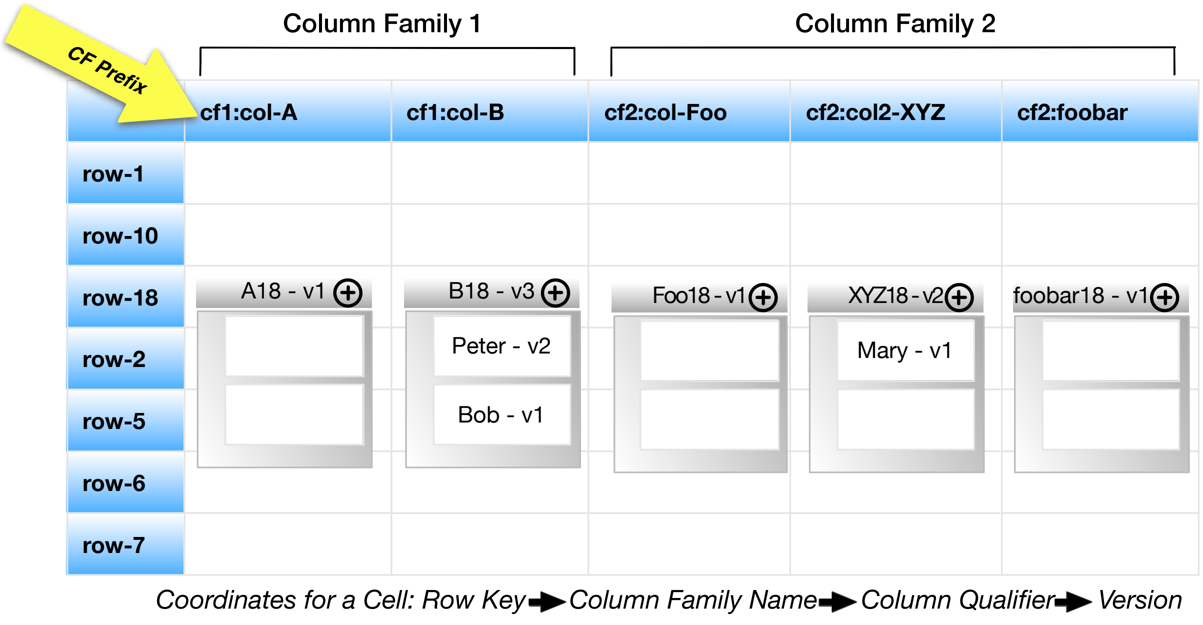


Figure A.1.: Column data model used in systems such as BigtTable, and Cassandra.

The case study web application described in in Chapter 5, uses Apache Cassandra as a data store. Inspired by BigTable [14] Cassandra's data model is a schema-optional, column-oriented data model. This means that, unlike a relational database, you do not need to model all of the columns required by an application up front, as each row is not required to have the same set of columns. Columns can be added by an application at runtime. In Cassandra, the keyspace is the container for application's data, similar to a database or schema in a relational database. Inside the keyspace there are one or more column family objects, which are analogous to tables. *Column families* contain *columns* and a set of related columns is identified by a *row key*. In each *row* data are stored in the basic storage unit which is a *cell*. Cassandra similar to Bigtable allows multiple timestamp versions of data within a cell. A cell can be addressed by its' *row-key*, *column familyname*, *Column qualifier* and the version as shown in Figure A.1. At the physical level, data are stored per column family contiguously on disk sorted by row-key, column name and version (column oriented data model).

B. Algorithms

Algorithm 4 shows a Java implementation of collaborative filtering, an online machine learning algorithm. It outputs up-to-date recommendations of items to users (function getRec) based on previous item ratings (function addRating). This imperative code contains annotations, further explained in the literature [28], that make possible to translate it directly to a stateful dataflow graph using a custom compiler.

Algorithm 4: SDGs Online collaborative filtering implementation.

```java
@Partitioned Matrix userItem = new Matrix();
@Partial Matrix coOcc = new Matrix();

void addRating(int user, int item, int rating) {
  userItem.setElement(user, item, rating);
  Vector userRow = userItem.getRow(user);
  for (int i = 0; i < userRow.size(); i++)
    if (userRow.get(i) > 0) {
      int count = coOcc.getElement(item, i);
      coOcc.setElement(item, i, count + 1);
      coOcc.setElement(i, item, count + 1);
    }
}
Vector getRec(int user) {
  Vector userRow = userItem.getRow(user);
  @Partial Vector userRec = @Global coOcc.multiply(userRow);
  Vector rec = merge(@Global userRec);
  return rec;
}
Vector merge(@Collection Vector[] allUserRec) {
  Vector rec = new Vector(allUserRec[0].size());
  for (Vector cur : allUserRec)
    for (int i = 0; i < allUserRec[0].size(); i++)
      rec.set(i, cur.get(i) + rec.get(i));
  return rec;
}
```

Algorithm 5 below, shows another implementation of collaborative filtering using Apache Spark [88]. This approach is using MLlib to deal with Implicit Feedback Datasets, meaning that instead of trying to model the matrix of ratings directly, it treats the data as a combination of binary preferences and confidence values. The algorithm is reading the rating from a Cassandra database and writing the intermediate data to an HDFS filesystem.

Algorithm 5: Spark collaborative filtering implementation (1/2).

```java

public class SparkCollaborativeFiltering {
  private static final String dataset_path = "hdfs://wombat30.doc.res.ic.ac.uk:8020/user/pg1712/
      lastfm_train";

  private static List<PlayList> allplaylists;
  private static Map<String, Integer> usersMap;
  private static List<User> allusers;
  private static Map<String, Integer> tracksMap;
  private static List<Track> tracksList;

  public static void main(String[] args) {

    long jobStarted = System.currentTimeMillis();
    SparkConf conf = new SparkConf()
        .set("spark.executor.memory","1g")
        .set("spark.driver.maxResultSize","1g")
        .setMaster("mesos://wombat30.doc.res.ic.ac.uk:5050")
        .setAppName("play2sdg Collaborative Filtering Job");
    JavaSparkContext sc = new JavaSparkContext(conf);
    /*
     *  Fetch the Track List
    */
    tracksList = CassandraQueryController.listAllTracksWithPagination();
    tracksMap = generateTrackMap( tracksList );
    logger.info("## Fetched # "+ tracksMap.size() +" Tracks ##");
    /*
     * Fetch PlayLists From Cassandra - aka Ratings
     */

    allplaylists = CassandraQueryController.listAllPlaylists();
    allusers = CassandraQueryController.listAllUsers();
    usersMap = generateUserMap( allusers );
    List<String> ratingList = new ArrayList<String>();
    /*
     * Convert IDS and save to HDFS File
     */
    for(PlayList playList : allplaylists){
      for(String track : playList.getTracks()){
        StringBuilder sb = new StringBuilder();
        sb.append(usersMap.get(playList.getUsermail()) + ",");
        sb.append(tracksMap.get(track) + ",");
        sb.append("5.0");
        ratingList.add(sb.toString());
      }
    }
    logger.info("## Converted ratings from: "+allplaylists.size() + " playlists##");
    /*
     * Persist To FS
     */
    RatingsFileWriter rw = new RatingsFileWriter("hdfs://wombat30.doc.res.ic.ac.uk:8020/user/pg1712/
        lastfm_subset");
    rw.persistRatingsFile(ratingList);
    // Load and parse the data
    String path = "hdfs://wombat30.doc.res.ic.ac.uk:8020/user/pg1712/lastfm_subset/ratings.data";
    logger.info("## Persisting to HDFS -> Done ##");
    JavaRDD<String> data = sc.textFile(path);
    JavaRDD<Rating> ratings = data.map(new Function<String, Rating>() {
      public Rating call(String s) {
        String[] sarray = s.split(",");
        return new Rating(Integer.parseInt(sarray[0]), Integer
            .parseInt(sarray[1]), Double.parseDouble(sarray[2]));
      }
    });
```

Algorithm 6: Spark collaborative filtering implementation (2/2).

```
    // Build the recommendation model using ALS
    int rank = 10;
    int numIterations = 20;
    MatrixFactorizationModel model = ALS.train(JavaRDD.toRDD(ratings),
        rank, numIterations, 0.01);
    // Evaluate the model on rating data
    JavaRDD<Tuple2<Object, Object>> userProducts = ratings
        .map(new Function<Rating, Tuple2<Object, Object>>() {
          public Tuple2<Object, Object> call(Rating r) {
            return new Tuple2<Object, Object>(r.user(), r.product());
          }
        });
    JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions = JavaPairRDD
        .fromJavaRDD(model
            .predict(JavaRDD.toRDD(userProducts))
            .toJavaRDD()
            .map(new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
              public Tuple2<Tuple2<Integer, Integer>, Double> call(
                  Rating r) {
                return new Tuple2<Tuple2<Integer, Integer>, Double>(
                    new Tuple2<Integer, Integer>(r.user(),
                        r.product()), r.rating());}
            }));
    JavaRDD<Tuple2<Double, Double>> ratesAndPreds = JavaPairRDD
        .fromJavaRDD(
            ratings.map(new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
              public Tuple2<Tuple2<Integer, Integer>, Double> call(
                  Rating r) {
                return new Tuple2<Tuple2<Integer, Integer>, Double>(
                    new Tuple2<Integer, Integer>(r.user(),
                        r.product()), r.rating());
              }
            })).join(predictions).values();

    double MSE = JavaDoubleRDD.fromRDD(
        ratesAndPreds.map(
            new Function<Tuple2<Double, Double>, Object>() {
              public Object call(Tuple2<Double, Double> pair) {
                Double err = pair.\_1() - pair.\_2();
                return err * err;
              }
            }).rdd()).mean();
    /**
     * Create recommendations based on stored Track and User id
     */
    List<Recommendation> newUserSongRec = new ArrayList<Recommendation>();
    for( Tuple2 <Tuple2<Integer, Integer>,Double> pred: predictions.toArray() ){
      logger.debug(”Creating Recommendation−> user: ”+pred.\_1().\_1 + ”\t track: ” + pred.\_1().\_2 +
           ”\t score: ”+pred.\_2() );
      Recommendation newRec = new Recommendation(allusers.get(pred.\_1().\_1).getEmail());
      newRec.getRecList().put(tracksList.get(pred.\_1().\_2).getTitle(), pred.\_2());
      newUserSongRec.add(newRec);
    }
    /**
     * Create an RDD from recommendations and Save it in parallel fashion
     */
    JavaRDD<Recommendation> rdd = sc.parallelize(newUserSongRec);
    rdd.persist(StorageLevel.MEMORY_AND_DISK_SER());
    rdd.foreach(new VoidFunction<Recommendation>() {
      @Override
      public void call(Recommendation t) throws Exception {
        CassandraQueryController.persist(t);
      }
    });
  }
  /**
   * Method Mapping generated Recommendations to Tracks and Users
   */
  public static void MapPredictions2Tracks(JavaPairRDD<Tuple2<Integer, Integer>, Double>
       predictions){
    for( Tuple2 <Tuple2<Integer, Integer>,Double> pred: predictions.toArray() ){
      logger.debug(”Creating Recommendation−> user: ”+pred.\_1().\_1 + ”\t track: ” + pred.\_1().\_2 +
           ”\t score: ”+pred.\_2() );
      Recommendation newRec = new Recommendation(allusers.get(pred.\_1()._1).getEmail());
      newRec.getRecList().put(tracksList.get(pred.\_1().\_2).getTitle(), pred.\_2());
      CassandraQueryController.persist(newRec);
    }
  }
}
```

C. Web Interface

This Chapter contains a number *Play2SDG* user interface screen captures. Figure C.1 depicts the Login page, and Figure C.2 the main user home page with tracks, preview and rating. Figure C.3 and Figure C.4 depict the user recommendation page and the coloborative filtering job statistics respectively.

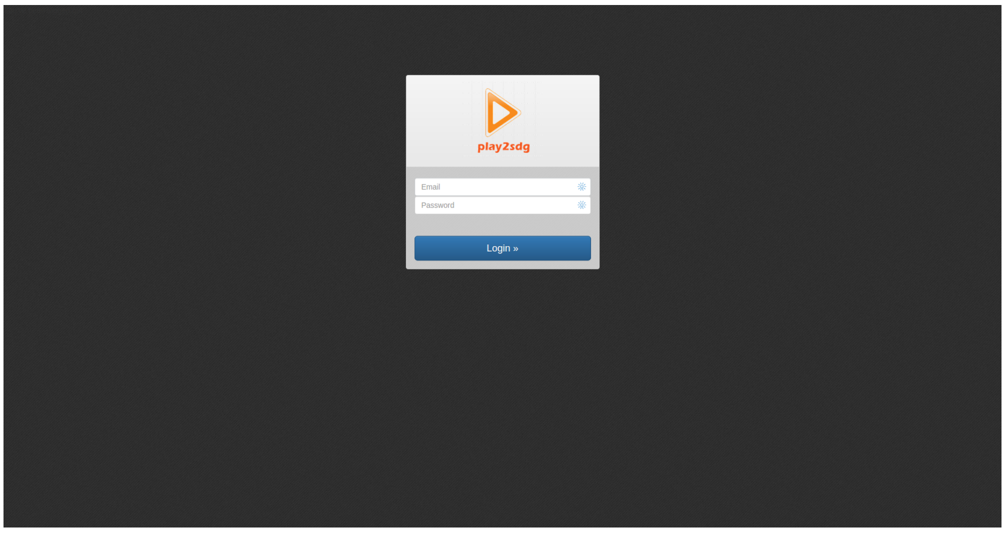
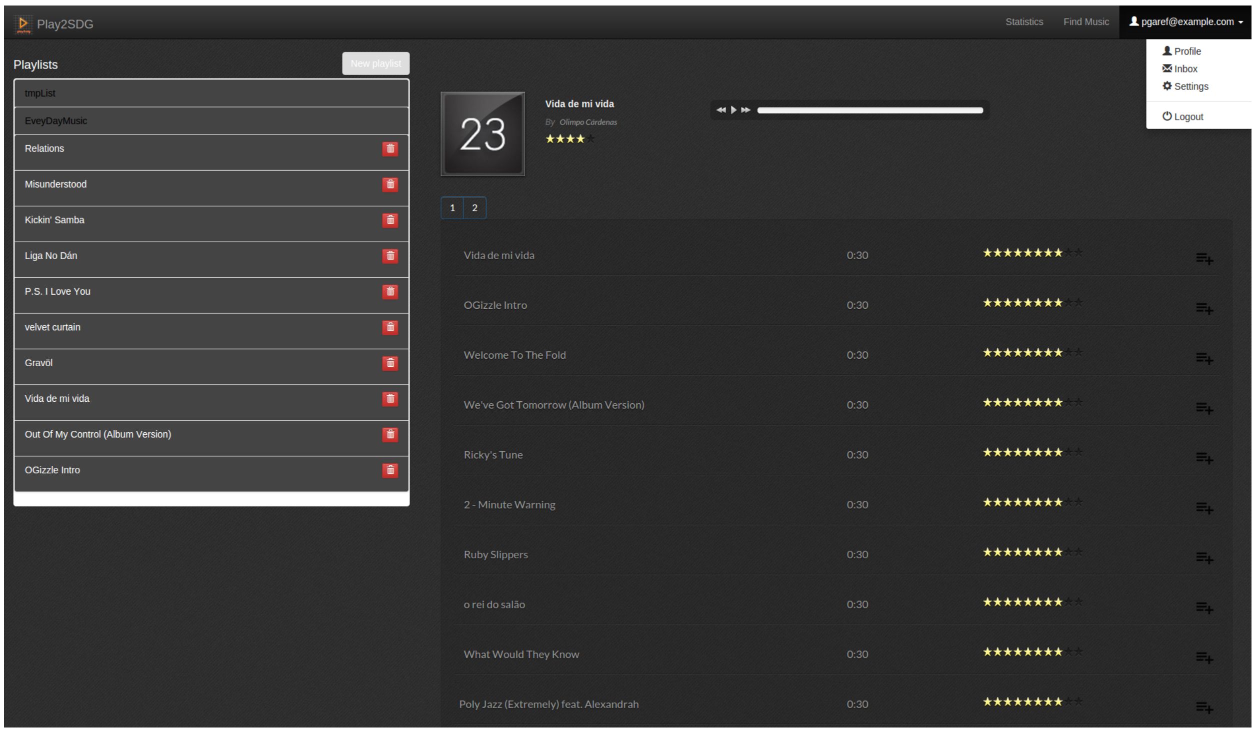


Figure C.1.: Play2SDG Login page View.

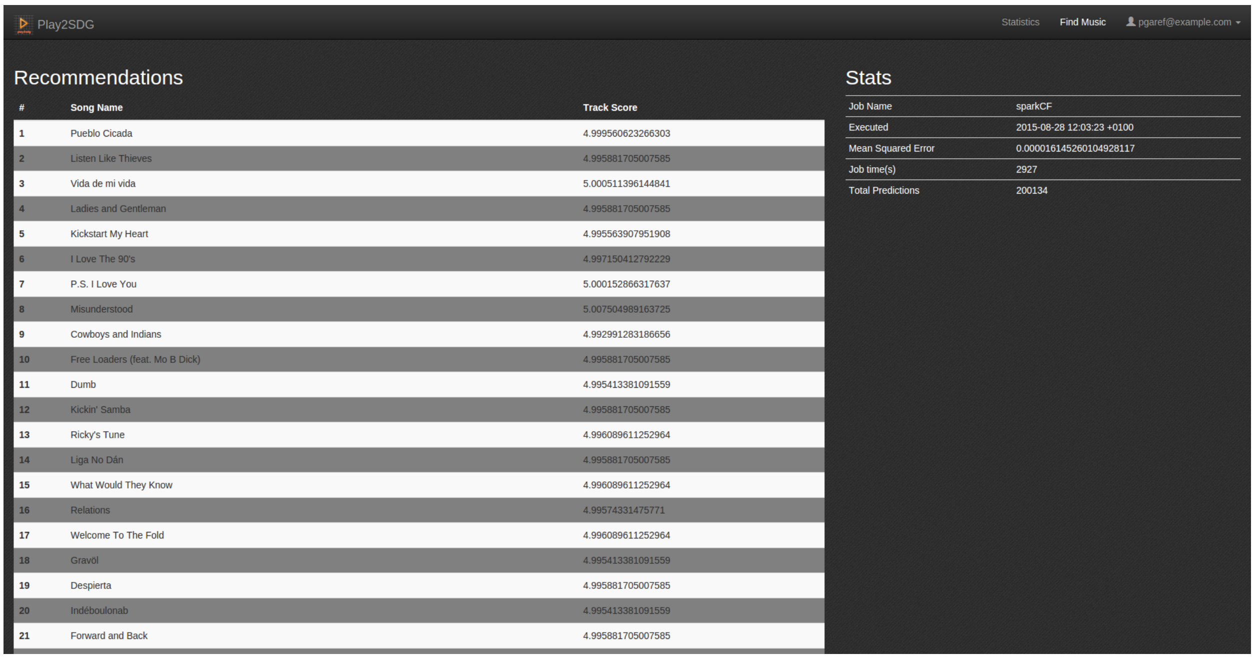


Playlists:
- tmpList
- EveyDayMusic
- Relations
- Misunderstood
- Kickin' Samba
- Liga No Dán
- P.S. I Love You
- velvet curtain
- Gravöl
- Vida de mi vida
- Out Of My Control (Album Version)
- OGizzle Intro

Vida de mi vida
By Olimpo Cárdenas
★★★★☆

Vida de mi vida	0:30	★★★★★★★★☆☆
OGizzle Intro	0:30	★★★★★★★★☆☆
Welcome To The Fold	0:30	★★★★★★★★☆☆
We've Got Tomorrow (Album Version)	0:30	★★★★★★★★☆☆
Ricky's Tune	0:30	★★★★★★★★☆☆
2 - Minute Warning	0:30	★★★★★★★★☆☆
Ruby Slippers	0:30	★★★★★★★★☆☆
o rei do salão	0:30	★★★★★★★★☆☆
What Would They Know	0:30	★★★★★★★★☆☆
Poly Jazz (Extremely) feat. Alexandrah	0:30	★★★★★★★★☆☆

Figure C.2.: Play2SDG Home page View.



Recommendations

#	Song Name	Track Score
1	Pueblo Cicada	4.999560623266303
2	Listen Like Thieves	4.995881705007585
3	Vida de mi vida	5.000511396144841
4	Ladies and Gentleman	4.995881705007585
5	Kickstart My Heart	4.995563907951908
6	I Love The 90's	4.997150412792229
7	P.S. I Love You	5.000152866317637
8	Misunderstood	5.007504989163725
9	Cowboys and Indians	4.992991283186656
10	Free Loaders (feat. Mo B Dick)	4.995881705007585
11	Dumb	4.995413381091559
12	Kickin' Samba	4.995881705007585
13	Ricky's Tune	4.996089611252964
14	Liga No Dán	4.995881705007585
15	What Would They Know	4.996089611252964
16	Relations	4.99574331475771
17	Welcome To The Fold	4.996089611252964
18	Gravöl	4.995413381091559
19	Despierta	4.995881705007585
20	Indéboulonab	4.995413381091559
21	Forward and Back	4.995881705007585

Stats

Job Name	sparkCF
Executed	2015-08-28 12:03:23 +0100
Mean Squared Error	0.000016145260104928117
Job time(s)	2927
Total Predictions	200134

Figure C.3.: Play2SDG Recommendations page View.

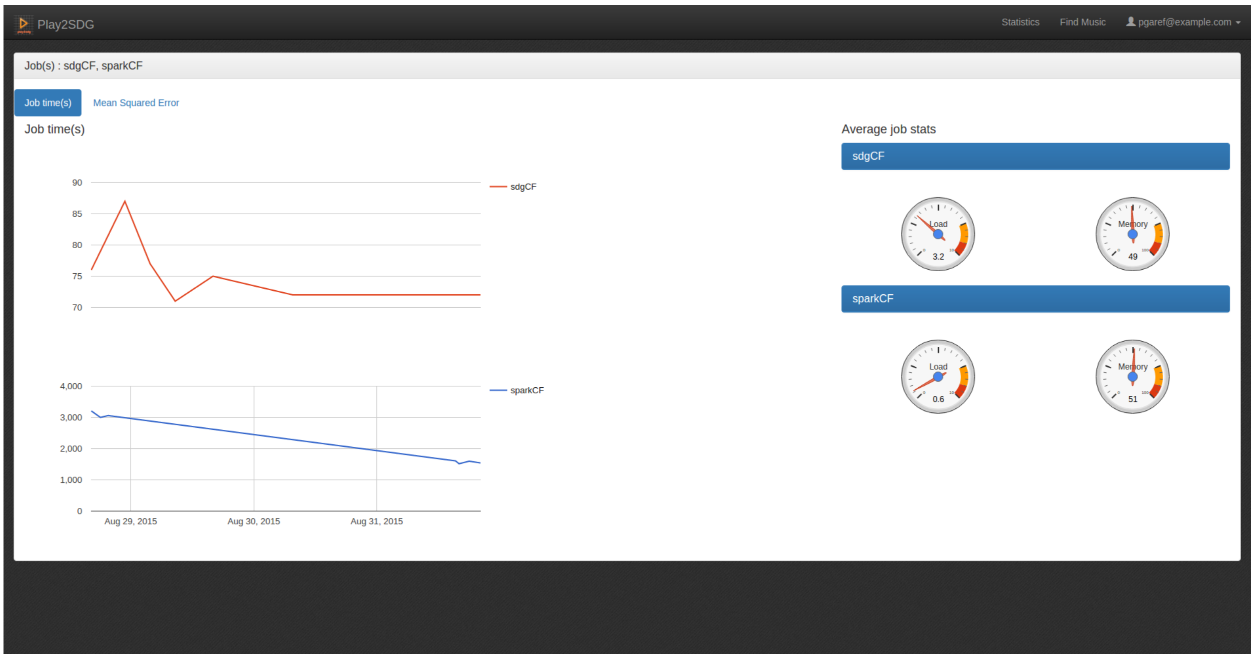


Figure C.4.: Play2SDG Statistics page View.

Bibliography

[1] Linux containers [online]. http://lxc.sourceforge.net, 2012.

[2] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. *SIGCOMM 38*, 4 (2008), 63–74.

[3] Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., et al. The stratosphere platform for big data analytics. In *VLDB* (2014).

[4] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center tcp (dctcp). *SIGCOMM 41*, 4 (2011), 63–74.

[5] Apache. Hadoop. http://hadoop.apache.org, 2013.

[6] Apache. Hbase. http://hbase.apache.org/, 2013.

[7] Arpteg, A. Big data at spotify. http://ictlabs-summer-school.sics.se/slides/company_presentation_spotify.pdf, 2015.

[8] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (2002), ACM, pp. 1–16.

[9] Bertin-Mahieux, T., Ellis, D. P., Whitman, B., and Lamere, P. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)* (2011).

[10] Bhatotia, P., Wieder, A., et al. Incoop: MapReduce for Incremental Computations. In *SOCC* (2011).

[11] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *USENIX OSDI* (2014).

[12] Bu, Y., Howe, B., et al. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB* (2010).

[13] Carvalho, M., Cirne, W., Brasileiro, F., and Wilkes, J. Long-term slos for reclaimed cloud computing resources. In *SOCC* (2014), ACM, pp. 1–13.

[14] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[15] Chun, B.-G., Condie, T., Curino, C., Douglas, C., Matusevych, S., Myers, B., Narayanamurthy, S., Ramakrishnan, R., Rao, S., Rosen, J., et al. Reef: Retainable evaluator execution framework. In *VLDB* (2013).

[16] Cook, H., Moreto, M., Bird, S., Dao, K., Patterson, D. A., and Asanovic, K. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *SIGARCH* (2013), vol. 41, ACM, pp. 308–319.

[17] Curino, C., Difallah, D. E., Douglas, C., Krishnan, S., Ramakrishnan, R., and Rao, S. Reservation-based scheduling: If you're late don't blame us! In *SOCC* (2014), ACM, pp. 1–14.

[18] Czajkowski, G. The multi-tasking virtual machine: Building a highly scalable jvm. `http://www.oracle.com/technetwork/articles/java/mvm-141094.html`, 2005.

[19] Dawson, M. Introduction to java multitenancy. `http://www.ibm.com/developerworks/java/library/j-multitenant-java/index.html`, 2013.

[20] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *OSDI* (2004).

[21] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *SIGOPS* (2007), vol. 41, ACM, pp. 205–220.

[22] Delgado, P., Dinu, F., Kermarrec, A.-M., and Zwaenepoel, W. Hawk: hybrid datacenter scheduling. In *USENIX ATC* (2015), USENIX Association, pp. 499–510.

[23] Delimitrou, C., and Kozyrakis, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGARCH 41*, 1 (2013), 77–88.

[24] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS* (2014).

[25] Ewen, S., Tzoumas, K., Kaufmann, M., and Markl, V. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment 5*, 11 (2012), 1268–1279.

[26] Fernandez, R. C., Migliavacca, M., et al. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *SIGMOD* (2013).

[27] Fernandez, R. C., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. Making state explicit for imperative big data processing. In *USENIX ATC* (2014).

[28] Fernandez, R. C., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. Making state explicit for imperative big data processing. In *USENIX ATC* (2014).

[29] Fitzpatrick, B., and Vorobey, A. Memcached: a distributed memory object caching system, 2011.

[30] Garefalakis, P., Papadopoulos, P., and Magoutis, K. Acazoo: A distributed key-value store based on replicated lsm-trees. In *SRDS* (2014), IEEE, pp. 211–220.

[31] Govindan, S., Liu, J., Kansal, A., and Sivasubramaniam, A. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *SOCC* (2011), ACM, p. 22.

[32] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource packing for cluster schedulers. In *SIGCOMM* (2014), ACM.

[33] Gunda, P. K., Ravindranath, L., et al. Nectar: Automatic Management of Data and Comp. in Datacenters. In *OSDI* (2010).

[34] He, B., Yang, M., et al. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *SOCC* (2010).

[35] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011).

[36] Isard, M., Budiu, M., et al. Dryad: Dist. Data-Parallel Programs from Sequential Building Blocks. In *EuroSys* (2007).

[37] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 261–276.

[38] Jeong, M. K., Erez, M., Sudanthi, C., and Paver, N. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *Proceedings of the 49th Annual Design Automation Conference* (2012), ACM, pp. 850–855.

[39] JMeter, A. Apache software foundation, 2010.

[40] Kandel, S., Paepcke, A., Hellerstein, J. M., and Heer, J. Enterprise data analysis and visualization: An interview study. *IEEE Transactions* (2012), 2917–2926.

[41] Karanasos, K., Rao, S., Curino, C., Douglas, C., Chaliparambil, K., Fumarola, G., Heddaya, S., Ramakrishnan, R., and Sakalanaga, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters.

[42] Kohavi, R., Deng, A., Longbotham, R., and Xu, Y. Seven rules of thumb for web site experimenters. In *SIGKDD* (2014), ACM.

[43] Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR* (2015).

[44] Krasner, G. E., Pope, S. T., et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming 1*, 3 (1988), 26–49.

[45] Kreitz, G., and Niemelä, F. Spotify–large scale, low latency, p2p music-on-demand streaming. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on* (2010), IEEE, pp. 1–10.

[46] Kreps, J., Narkhede, N., Rao, J., et al. Kafka: A distributed messaging system for log processing. In *NetDB* (2011), pp. 1–7.

[47] Lakshman, A., and Malik, P. Cassandra: a decentralized structured storage system. *SIGOPS 44*, 2 (2010), 35–40.

[48] Le, Q. V. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (2013), IEEE, pp. 8595–8598.

[49] Leverich, J., and Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 4.

[50] Linden, G., Smith, B., and York, J. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE* (2003).

[51] Lo, D., Cheng, L., Govindaraju, R., Barroso, L. A., and Kozyrakis, C. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st annual international symposium on Computer architecuture* (2014), IEEE Press, pp. 301–312.

[52] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving resource efficiency at scale. In *ISCA* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 450–462.

[53] Logothethis, D., Olson, C., et al. Stateful Bulk Processing for Incremental Analytics. In *SOCC* (2010).

[54] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO* (2011), ACM.

[55] Marshall, P., Keahey, K., and Freeman, T. Improving utilization of infrastructure clouds. In *CCGrid* (2011), IEEE, pp. 205–214.

[56] Meisner, D., Sadler, C. M., Barroso, L. A., Weber, W.-D., and Wenisch, T. F. Power management of online data-intensive services. In *ISCA* (2011), IEEE, pp. 319–330.

[57] Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., and Vassilakis, T. Dremel: interactive analysis of web-scale datasets. In *VLDB* (2010).

[58] Menage, P., Jackson, P., and Lameter, C. Cgroups. *Available on-line at: http://www. mjmwired. net/kernel/Documentation/cgroups. txt* (2008).

[59] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (2014), 2.

[60] Mesosphere. Mesosphere infinity: Youre 4 words away from a complete big data system. `https://mesosphere.com/blog/2015/08/20/mesosphere-infinity-youre-4-words-away-from-a-complete-big-data-system/`.

[61] Murray, D., Schwarzkopf, M., et al. CIEL: A Universal Exec. Engine for Distributed Data-Flow Comp. In *NSDI* (2011).

[62] Murray, D. G., McSherry, F., et al. Naiad: A Timely Dataflow System. In *SOSP* (2013).

[63] Nathuji, R., Kansal, A., and Ghaffarkhah, A. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys* (2010), ACM, pp. 237–250.

[64] Nesbit, K. J., Aggarwal, N., Laudon, J., and Smith, J. E. Fair queuing memory systems. In *MICRO* (2006), IEEE, pp. 208–222.

[65] Neumeyer, L., Robbing, B., et al. S4: Distributed Stream Computing Platform. In *ICDMW* (2010).

[66] Novakovic, D., Vasic, N., Novakovic, S., Kostic, D., and Bianchini, R. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *USENIX ATC* (2013).

[67] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: a not-so-foreign language for data processing. In *SIGMOD* (2008), ACM, pp. 1099–1110.

[68] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: distributed, low latency scheduling. In *SOSP* (2013), ACM, pp. 69–84.

[69] Pacheco, P. S. *Parallel programming with MPI.* Morgan Kaufmann, 1997.

[70] Power, R., and Li, J. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI* (2010).

[71] Qureshi, M. K., and Patt, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006), IEEE Computer Society, pp. 423–432.

[72] Reese, W. Nginx: the high-performance web server and reverse proxy. *Linux Journal 2008*, 173 (2008), 2.

[73] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC* (2012), ACM, p. 7.

[74] Sanchez, D., and Kozyrakis, C. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 57–68.

[75] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys* (2013), ACM.

[76] Sumbaly, R., Kreps, J., and Shah, S. The big data ecosystem at linkedin. In *SIGMOD* (2013).

[77] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al. Storm@ twitter. In *SIGMOD* (2014), ACM.

[78] Typesafe. Play framework. `http://www.playframework.com/`.

[79] Vasić, N., Novaković, D., Miučin, S., Kostić, D., and Bianchini, R. Dejavu: accelerating resource allocation in virtualized environments. In *SIGARCH* (2012), vol. 40, ACM, pp. 423–436.

[80] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. Apache hadoop yarn: Yet another resource negotiator. In *SOCC* (2013), ACM.

[81] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *EuroSys* (2015), ACM.

[82] W, R. Facebook doesnt have big data. it has ginormous data. `http://www.xconomy.com/san-francisco/2013/02/14/how-facebook-uses-ginormous-data-to-grow-its-business/2/`, 2013.

[83] Wilkes, J. More google cluster data. `http://googleresearch.blogspot.ch/2011/11/more-google-clusterdata.html`.

[84] Xin, R. S., Rosen, J., et al. Shark: SQL and Rich Analytics at Scale. In *SIGMOD* (2013).

[85] Yu, Y., Isard, M., et al. DryadLINQ: a System for General-Purpose Distributed Data-Parallel Computing using a High-Level Language. In *OSDI* (2008).

[86] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys* (2010), ACM.

[87] Zaharia, M., Chowdhury, M., et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI* (2012).

[88] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster Computing with Working Sets. In *HotCloud* (2010), pp. 10–10.

[89] Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing* (2012), USENIX Association, pp. 10–10.

[90] Zhang, B., Kreitz, G., Isaksson, M., Ubillos, J., Urdaneta, G., Pouwelse, J., Epema, D., et al. Understanding user behavior in spotify. In *INFOCOM* (2013), IEEE.

[91] Zhang, X., Tune, E., Hagmann, R., Jnagal, R., Gokhale, V., and Wilkes, J. Cpi 2: Cpu performance isolation for shared compute clusters. In *EuroSys* (2013), ACM.

[92] Zhou, R., Khemmarat, S., and Gao, L. The impact of youtube recommendation system on video views. In *SIGCOMM* (2010), ACM.

List of Acronyms and Abbreviations

ALS	Alternating Least Squares
API	Application Program Interface
BE	Best Effort
CF	Collaborative Filtering
CFS	Completely Fair Scheduler
DRAM	Dynamic Random-Access Memory
GC	Garbage Collector
HTTP	Hypertext Transfer Protocol
HW	Hardware
I/O	Input-Output
IT	Information Technology
IWO	In-memory Web Object
IWOs	In-memory Web Objects
JIT	Just In Time
JPA	Java Persistence API
JVM	Java Virtual Machine
LC	Latency Critical
LLC	Last Level Cache
LXC	Linux Containers
MVC	Model View Controller
NUMA	Non-Uniform Memory Access
OOP	Object Oriented Programming
OS	Operating System
RDD	Resilient Distributed Dataset
RDDs	Resilient Distributed Datasets

REST	Representational State Transfer
SDG	Stateful Dataflow Graph
SLA	Service Level Agreement
SLO	Service Level Objective
SW	Software
TPS	Transaction per Second
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VM	Virtual Machine
WAL	Write Ahead Log