Numéro National de Thèse : 2019LYSEN077

THESE de DOCTORAT DE L'UNIVERSITE DE LYON

opérée par

l'Ecole Normale Supérieure de Lyon

Ecole Doctorale N° 512

École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 17/12/2019, par :

Jad DARROUS

Scalable and Efficient Data Management in Distributed Clouds: Service Provisioning and Data Processing

Gestion de données efficace et à grande échelle dans les clouds distribués : Déploiement de services et traitement de données

Devant le jury composé de :

SENS, Pierre	Professeur	Sorbonne Université	Rapporteur
PEREZ, Maria S.	Professeure	Universidad Politécnica de Madrid	Rapporteure
PIERRE, Guillaume	Professeur	Université de Rennes 1	Examinateur
PASCAL-STOLF, Patricia	Maître de Conférences	IUT de Blagnac	Examinatrice
PEREZ, Christian	Directeur de Recherche	Inria Grenoble Rhône-Alpes	Directeur de thèse
IBRAHIM, Shadi	Chargé de Recherche	Inria Rennes Bretagne-Atlantique	Co-encadrant de thèse

Acknowledgements

Reaching the point of writing this acknowledgment is the result of the help, encouragement, and support of many people around me that I would like to thank.

I would like first to express my gratitude for the members of the jury: Guillaume Pierre and Patricia Pascal-Stolf as well as my evaluators Pierre Sens and Maria S. Perez for dedicating part of their valuable time to assess my work.

I would like to express my sincere gratitude to Shadi Ibrahim. Shadi was more than an advisor, his efforts were invaluable for me during these three years scientifically and personally. I am more than happy to had the chance to work closely with him which gave me the prospect to grow as a researcher. I would like also to dedicate a big thanks to Christian Perez. Christian happily accepted to be my thesis director, provided me with ceaseless support, and taught me how to ask "metaphysical" questions without forgetting the smallest detail. Also, I would like to thank Gilles Fedak who advised me during the first year and was the first contact with my Ph.D. journey.

During my Ph.D., I had the chance to work and interact with many amazing and brilliant people that I would like to thank. Special thanks go to my two co-authors of my two first papers, Amelie and Thomas, for the nice collaboration. Thanks to the members of DISCOVERY initiative for the interesting discussions and presentations. Many thanks go to the members of Stack and Avalon teams for the nice moments we spent together.

Likewise, I would like to thank my friends that I have shared happy and pleasant moments with (in my remaining time!), especially Maha, Lisa, and Bachar.

Last but not least, I would like to thank my beloved family for their continuous encouragement and support during this very special chapter of my academic life. In particular, I would like to thank my parents who have always believed in me.

Abstract

We are living in the era of Big Data where data is generated at an unprecedented pace from various sources all over the world. These data can be transformed into meaningful information that has a direct impact on our daily life. To cope with the tremendous volumes of Big Data, large-scale infrastructures and scalable data management techniques are needed. Cloud computing has been evolving as the de-facto platform for running data-intensive applications. Meanwhile, large-scale storage systems have been emerging to store and access data in cloud data centers. They run on top of thousands of machines to offer their aggregated storage capacities, in addition to providing reliable and fast data access. The distributed nature of the infrastructures and storage systems introduces an ideal platform to support Big Data analytics frameworks such as Hadoop or Spark to efficiently run data-intensive applications.

In this thesis, we focus on scalable and efficient data management for building and running data-intensive applications. We first study the management of virtual machine images and containers images as the main entry point for efficient service provisioning. Accordingly, we design, implement and evaluate Nitro, a novel VMI management system that helps to minimize the transfer time of VMIs over a heterogeneous wide area network (WAN). In addition, we present two container image placement algorithms which aim to reduce the maximum retrieval time of container images in Edge infrastructures. Second, towards efficient Big Data processing in the cloud, we investigate erasure coding (EC) as a scalable yet cost-efficient alternative for replication in data-intensive clusters. In particular, we conduct experiments to thoroughly understand the performance of data-intensive applications under replication and EC. We identify that data reads under EC are skewed and can result in noticeable performance degradation of data-intensive applications. We then introduce an EC-aware data placement algorithm that targets balancing data accesses across nodes and therefore improving the performance of data-intensive applications under EC.

Résumé

Si nous stockions tous les mots jamais prononcés par l'être humain, leur volume serait égale à celui des données générées en seulement deux jours[1]. Le dernier rapport de l'International Data Corporation (IDC) prévoit même que les données générées dépassent les 175 zettaoctets[2] d'ici à 2025 [130]. Ces données sont produites par diverses sources telles que des capteurs, des médias sociaux ou des simulations scientifiques. Ce flot de données permet aux entreprises et aux universités d'améliorer leurs services et leurs connaissances qui ont un impact direct sur nos vies. Par exemple, les grandes sociétés Internet telles que Google et Facebook analysent les données recueillies quotidiennement pour améliorer l'expérience des utilisateurs, tandis que des instituts de recherche comme le Laboratoire National d'Argonne effectuent des simulations complexes pour repousser les limites du savoir humain [255]. Il est donc indispensable de permettre une gestion des données efficace à grande échelle pour transformer ces gigantesques volumes de données en informations utilisables.

Pour faire face à l'énorme volume de données (ce qu'on appelle le "Big Data"), des infrastructures et des techniques de gestion de données à grande échelle sont nécessaires. De fait, les centres de calculs sont devenus les plate-formes privilégiées pour l'exécution d'applications de type « Big Data ». les centres de calculs donnent l'illusion de ressources infinies auxquelles on peut accéder à bas prix. Récemment, des fournisseurs d'informatique en cloud tels qu'Amazon, Microsoft et Google ont doté leurs infrastructures de millions de serveurs répartis dans le monde entier pour faciliter la gestion des données massives. Par exemple, Amazon Web Services (AWS) compte au total près de 5 millions de serveurs [144], hébergés dans des centaines de centres de données répartis sur 5 continents [26], et des millions de services y sont lancés tous les jours [35]. D'autre part, des systèmes de stockage à grande échelle sont apparus pour stocker et accéder aux données dans ces centres de données. Ils fonctionnent sur des milliers de machines offrant leurs capacités de stockage agrégées, en plus de fournir un accès fiable et rapide aux données. Par exemple, le système cloud Windows Azure Storage (WAS) héberge plus d'un exaoctet de données [123] et traite plus de deux milliards de transactions par jour [45]. La nature distribuée des infrastructures et des systèmes de stockage en fait une plate-forme idéale pour prendre en charge les infrastructures logicielles d'analyse Big Data telles que Hadoop [19] ou Spark [20] pour exécuter efficacement des applications de type « Big Data ». Par exemple, plus d'un million de tâche Hadoop, traitant des dizaines de pétaoctets de données, sont lancés chaque mois dans les centres de données de Facebook [49].

En général, les services cloud, y compris les services d'analyse de données (p. ex.

[1]On estime que 2,5 exaoctets sont produits chaque jour [117] et que tous les mots jamais prononcés par l'être humain ont une taille de 5 exaoctets [116].

[2]1 zettaoctet équivaut à 10^{21} octets, soit 1 million de pétaoctets.

Amazon Elastic MapReduce [14] ou Microsoft Azure HDInsight [112]), sont déployés en tant que machines virtuelles (VM, virtual machine) ou conteneurs. Le déploiement d'un service pour exécuter des applications cloud nécessite que l'image du service (une image qui encapsule le système d'exploitation et le logiciel du service ainsi que ses dépendances) soit stockée localement sur le serveur hôte. Sinon, l'image correspondante est transférée via le réseau vers le serveur de destination, ce qui introduit un délai dans le temps de déploiement. La demande croissante de fourniture rapide de services, en plus de la taille et du nombre croissants d'images de services (p. ex. AWS fournit plus de 20 000 images différentes[3], et leurs tailles atteignent des dizaines de gigaoctets [35, 258]) rendent la gestion des images de service essentielle pour la déploiement de services dans le cloud. De plus, la tendance actuelle au déploiement sur plusieurs sites - facilitée par les centres de calcul géographiquement distribués [296, 26, 101] - et l'adoption progressive de l'Edge pour permettre le traitement sur place (c.-à-d. en déplaçant le calcul vers les sources de données) posent de nouveaux défis pour le déploiement de machines virtuelles et de conteneurs. Cela est dû à la bande passante limitée et à l'hétérogénéité des connexions de réseau étendu (WAN, wide area network), ainsi qu'aux capacités de stockage limitées des serveurs Edge. Par conséquent, il est crucial de fournir une gestion des images de conteneur et de machine virtuelle efficace à grande échelle pour faciliter et permettre un déploiement rapide des services dans les clouds distribués et les systèmes Edge.

Traditionnellement, les systèmes de stockage distribués utilisent la réplication pour assurer la disponibilité des données. En outre, les infrastructures d'analyse des données, notamment Spark [20], Flink [18] et Hadoop [19], bénéficient largement de la réplication pour gérer les pannes (c.-à-d. les tâches des machines en panne peuvent être simplement ré-exécutées ailleurs à l'aide d'autres répliques des données [66, 305]) et améliorer les performances des applications de type « Big Data » dans les centres de calculs en augmentant la probabilité d'ordonnancement des tâches de calcul sur la machine hébergeant les données d'entrée [36, 128, 309]). Cependant, avec la croissance incessante de la quantité de données et l'adoption progressive de périphériques de stockage rapides mais cher (c.-à-d. SSD et DRAM), les coûts matériels et de stockage de la réplication deviennent de plus en plus importants [234, 252, 316]. Les codes d'effacement (EC, erasure coding) apparaissent comme une alternative offrant une grande disponibilité avec un surcoût de stockage moindre. Ainsi, ils sont actuellement déployés dans de nombreux systèmes de stockage distribués [263, 111, 86, 123, 252, 195]. Par exemple, en appliquant EC, Microsoft réduit de plus de 50% le surcoût de stockage par rapport à la réplication [123]. L'exécution d'applications de type « Big Data » sous EC réduira aussi le surcoût de stockage, en revanche, ça peut entraîner un important transfert de données car les données d'entrée de chaque tâche de calcul sont dispersées sur plusieurs machines. Par conséquent, il est important de comprendre les performances des applications de type « Big Data » sous EC afin de combler le fossé entre les infrastructures logicielles d'analyse et les données codées par effacement, afin de permettre un traitement efficace de ces données à grande échelle.

En résumé, le sujet de cette thèse est la gestion de données efficace à grand échelle pour le déploiement et l'exécution d'applications de type « Big Data » dans les centres de caluls.

[3]Nous avons obtenu le nombre d'images depuis le tableau de bord d'Amazon EC2 le 26 juil. ., 2019.

Contributions

Dans cette thèse, nous nous concentrons sur la gestion de données efficace à grande échelle pour la conception et l'exécution d'applications de type « Big Data ». Nous étudions la gestion des images de machines virtuelles et des images de conteneurs en tant que point de départ principal pour le déploiement efficace de services. De plus, nous étudions les codes d'effacement comme alternative à la réplication permettant à la fois un meilleur passage à l'échelle et une diminution du surcoût de stockage dans les clusters très consommateurs de données. Les principales contributions de cette thèse peuvent être résumées ainsi :

Permettre le déploiement efficace de services dans les centres de calcul géo-distribués. La plupart des grands fournisseurs de services cloud, tels qu'Amazon et Microsoft, répliquent leurs images de machines virtuelles (VMIs, virtual machine images) sur plusieurs centres de données éloignés géographiquement pour offrir un déploiement rapide de leurs services. La mise en place d'un service peut nécessiter le transfert d'une VMI sur le réseau WAN. Sa durée dépend donc de la distribution des VMIs et de la bande passante du réseau entre les sites différents. Néanmoins, les méthodes existantes pour faciliter la gestion de VMIs (c.-à-d. récupération des VMIs) négligent l'hétérogénéité du réseau dans les clouds géo-distribués. Pour y répondre nous proposons Nitro, un nouveau système de gestion de VMI qui permet de minimiser le temps de transfert des VMIs sur un réseau WAN hétérogène. Pour atteindre cet objectif, Nitro intègre deux techniques complémentaires. Premièrement, il utilise la déduplication pour réduire la quantité de données qui sera transférée en raison des similitudes élevées entre les images. Deuxièmement, Nitro est doté d'une stratégie de transfert de données prenant les caractéristiques du réseau en compte afin d'exploiter efficacement les liaisons à bande passante élevée lors de la récupération de données et ainsi accélérer le temps de déploiement. Les résultats expérimentaux montrent que cette stratégie de transfert de données constitue la solution optimale lors de l'acquisition de VMI tout en minimisant les coûts engendrés. De plus, Nitro surpasse de 77% les systèmes de stockage VMI (c.-à-d. OpenStack Swift), à la pointe de la technologie. Ces travaux ont conduit à une publication lors de la conférence internationale CCGrid '18 [62].

Prise en compte des charactéristiques du réseau lors du placement d'images conteneur sur un réseau de type Edge. L'Edge computing vise à d'étendre le cloud computing en rapprochant les calculs des sources de données afin d'améliorer les performances les applications et les services à court temps d'exécution et à faible latence. Fournir un temps de déploiement de ces services rapide et prédictible constitue un défi nouveau à l'importance croissante, à mesure que le nombre de serveurs Edge augmente et que l'hétérogénéité des réseaux augmente. Ce travail est motivé par une question simple : pouvons-nous placer des images de conteneur sur des serveurs Edge de manière à ce qu'une image puisse être récupérée sur n'importe quel serveur Edge rapidement et dans un délai prévisible ? pour ce faire, nous proposons KCBP et KCBP-WC, deux algorithmes de placement d'images de conteneur qui visent à réduire le temps de récupération maximal de ces images. KCBP et KCBP-WC sont basés sur la résolution du problème du k-Center. Cependant, KCBP-WC essaie en plus d'éviter de placer de grandes couches d'une image de conteneur sur le même serveur Edge. Les évaluations utilisant des simulations basées sur des traces montrent que KCBP et KCBP-WC peuvent être appliquées à diverses configurations de réseau et permettent de réduire le temps de récupération maximal des images de conteneur de 1,1 à 4x par rapport aux techniques de placements de l'état de l'art (c.-à-d. Best-Fit et Random). Ces travaux ont conduit à une publication à la conférence internationale ICCCN '19 [63].

Caractérisation des performances des code d'effacement dans des clusters avec un volume important de données. Les clusters à forte intensité de données s'appuient sur les systèmes de stockage distribués pour s'adapter à la croissance sans précédent des quantités de données. Le système de fichiers distribué Hadoop (HDFS) [263] est le système de stockage principal des infrastructures logicielles d'analyse de données telles que Spark [20] et Hadoop [19]. Traditionnellement, HDFS fonctionne en utilisant la réplication pour garantir la disponibilité des données et permettre l'exécution locale de tâches d'applications de type « Big Data ». Récemment, les codes d'effacements (EC) apparaît comme une méthode alternative à la réplication dans les systèmes de stockage en raison de son surcoût réduit en temps de calcul. Malgré un grand nombre d'études visant à améliorer le temps de récupération sous EC en termes de sur-sollicitation de réseau et de disque, les caractéristiques de performance des tâches d'analyse exécutées sous EC ne sont pas claires. En réponse, nous avons mené des expériences pour bien comprendre les performances d'applications de type « Big Data » utilisant la réplication et EC. Nous avons utilisé des tests représentatifs sur le banc d'essai Grid'5000 [105] pour évaluer l'incidence des paramètres d'accès aux données, des accès simultanés aux données, des charges de travail analytiques, de la persistance des données, des défaillances, des périphériques de stockage principaux et de la configuration réseau. Si certains des résultats sont conformes à l'intuition, d'autres sont plus inattendus. Par exemple, la contention au niveau du disque et du réseau causés par la distribution aléatoire des blocs et la méconnaissance de leurs fonctionnalités sont les principaux facteurs affectant les performances des applications de type « Big Data » sous EC, et non la non-localité des données. Une partie de ce travail a conduit à une publication lors de la conférence internationale MASCOTS '19 [61].

Pris en compte des codes d'effacement par le système de fichiers distribués Hadoop (HDFS). Nous observons que l'ordonnanceur de tâches Hadoop ne prend pas en compte la structure des fichiers sous EC et peut entraîner un déséquilibre notable des accès aux données sur les serveurs lors de l'exécution d'applications de type « Big Data ». Cela entraîne des stragglers (c.-à-d. certaines tâches présenteront un grand écart dans leur exécution et prendront plus de temps que leur exécution moyenne) qui prolongeront à leur tour le temps d'exécution des applications. En conséquence, dans le but d'améliorer les performances de l'analyse de données avec les codes d'effacement, nous proposons un algorithme de placement sensible à EC qui équilibre les accès aux données d'un serveur à l'autre en tenant compte de la sémantique des blocs de données lors de leur distribution. Nos expériences avec la série de test Grid'5000 [105] ont montré que le placement EC peut réduire jusqu'à 25% le temps d'exécution des applications Sort et Wordcount. Nos résultats motivent l'intégration de technique prenant en compte les spécificité d'EC au niveau de l'ordonnancement afin de faire face à la dynamicité de l'environnement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Contents

1.1 Context

If we store all words ever spoken by human beings, its size will be equal to the size of data generated in just two days[1]. Furthermore, the latest report from the International Data Corporation (IDC) expects the generated data to bypass the 175 zettabytes[2] by 2025 [130]. These data are produced by various sources such as sensors, social media, or scientific simulations. This deluge of data allows empowering businesses as well as academia and has a direct impact on our lives. For example, large Internet companies such as Google and Facebook analyze their daily collected data to improve users' experiences, while research institutes such as Argonne National Laboratory run complex universe simulations to push further the boundaries of human knowledge [255]. Hence, enabling scalable and efficient data management to transform these gigantic data volumes into useful information is indispensable.

To cope with the tremendous volumes of Big Data, large-scale infrastructures and scalable data management techniques are needed. Cloud computing has been evolving as the de-facto platform for running data-intensive applications. Clouds provide the illusion of infinite resources which can be accessed in a cost-effective manner. Recently, cloud providers such as Amazon, Microsoft, and Google have equipped their infrastructures with millions of servers distributed world-wide to ease the management of Big Data. For example, Amazon Web Services (AWS) has almost 5 million servers in total [144] hosted

[1]It is estimated that 2.5 exabytes are produced every day [117] and all words ever spoken by human beings have a size of 5 exabytes [116].

[2]1 zettabyte equals to 10^{21} bytes, or 1 million petabytes.

in hundreds of data centers on 5 continents [26], with millions of services launched every day [35]. Meanwhile, large-scale storage systems have been emerging to store and access data in cloud data centers. They run on top of thousands of machines to offer their aggregated storage capacities, in addition to providing reliable and fast data access. For example, the cloud-based Windows Azure Storage (WAS) system hosts more than an exabyte of data [123] and handles more than two billion transactions per day [45]. The distributed nature of the infrastructures and storage systems introduces an ideal platform to support Big Data analytics frameworks such as Hadoop [19] or Spark [20] to efficiently run data-intensive applications. For instance, over one million Hadoop jobs, processing tens of petabytes of data, are launched every month in Facebook data centers [49].

In general, cloud services, including data analytics services (e.g., Amazon Elastic MapReduce [14], Microsoft Azure HDInsight [112], etc.), are deployed as virtual machines (VMs) or containers. Provisioning a service to run cloud applications requires the service image (an image which encapsulates the operating system and the service software along with its dependencies) to be stored locally on the host server. Otherwise, the corresponding image is transferred through the network to the destination server introducing a delay in the provisioning time. The increasing demand for fast service provisioning in addition to the increasing size and number of service images (e.g., AWS provides more than 20,000 different public images[3], and their sizes could attain dozens of gigabytes [35, 258]) make service image management essential for service provisioning in the cloud. Moreover, the current trend towards multi-site deployment – which is facilitated by geographically distributed clouds [296, 26, 101] – and the wide adoption of Edge computing to enable in-place processing (i.e., by moving computation near data sources) bring new challenges in provisioning VMs and containers. This is due to the limited bandwidth and the heterogeneity of the wide area network (WAN) connections, as well as the limited storage capacities in Edge-servers. Consequently, it is crucial to provide scalable and efficient VM and container images management to ease and enable fast service provisioning in distributed clouds and Edge systems.

Traditionally, distributed storage systems operate under *replication* to ensure data availability. In addition, data analytics frameworks including Spark [20], Flink [18], and Hadoop [19] have extensively leveraged replication to handle machine failures (i.e., tasks of the failed machines can be simply re-executed using other replicas of the data [66, 305]) and improve the performance of data-intensive applications in clouds (i.e., improve data locality by increasing the probability of scheduling computation tasks on a machine which hosts the input data [36, 128, 309]). However, with the relentless growth of Big Data, and the wide adoption of high-speed yet expensive storage devices (i.e., SSDs and DRAMs) in storage systems, replication has become expensive in terms of storage cost and hardware cost [234, 252, 316]. Alternatively, erasure codes (EC) provide high data availability with low storage overhead. Thus, they are currently deployed in many distributed storage systems [263, 111, 86, 123, 252, 195]. For example, by applying EC, Microsoft reduces the storage overhead in its cloud-based object store by more than 50% compared to replication [123]. While executing data-intensive applications under EC can also result in low storage overhead, this may incur large data transfer because the input data of each computation task (e.g., map tasks in Hadoop) is scattered on several machines.

[3]We obtained the number of images from Amazon EC2 Dashboard on Jul. 26th, 2019.

Therefore, it is important to understand the performance of data-intensive applications under EC and bridge the gap between analytics frameworks and erasure-coded data, to enable efficient large-scale data processing.

Briefly, the subject of this thesis is scalable and efficient data management for provisioning and running data-intensive applications in clouds.

1.2 Contributions

In this thesis, we focus on scalable and efficient data management for building and running data-intensive applications. We study the management of virtual machine images and containers images as the main entry point for efficient service provisioning. Moreover, we investigate erasure codes as a scalable yet cost-efficient alternative for replication in data-intensive clusters. The main contributions of this thesis can be summarized as follows:

Enabling Efficient Service Provisioning in Geo-distributed Clouds

Most large cloud providers, such as Amazon and Microsoft, replicate their Virtual Machine Images (VMIs) on multiple geographically distributed data centers to offer fast service provisioning. Provisioning a service may require to transfer a VMI over the wide area network (WAN) and therefore is dictated by the distribution of VMIs and the network bandwidth in-between sites. Nevertheless, existing methods to facilitate VMI management (i.e., retrieving VMIs) overlook network heterogeneity in geo-distributed clouds. In response, we design, implement and evaluate Nitro, a novel VMI management system that helps to minimize the transfer time of VMIs over a heterogeneous WAN. To achieve this goal, Nitro incorporates two complementary features. First, it makes use of deduplication to reduce the amount of data which is transferred due to the high similarities within an image and in-between images. Second, Nitro is equipped with a network-aware data transfer strategy to effectively exploit links with high bandwidth when acquiring data and thus expedites the provisioning time. Experimental results show that our network-aware data transfer strategy offers the optimal solution when acquiring VMIs while introducing minimal overhead. Moreover, Nitro outperforms state-of-the-art VMI storage systems (i.e., OpenStack Swift) by up to 77%. This work led to a publication at the CCGrid ′18 international conference [62].

Network-Aware Container Image Placement in the Edge

Edge computing promises to extend clouds by moving computation close to data sources to facilitate short-running and low-latency applications and services. Providing fast and predictable service provisioning time presents a new and mounting challenge, as the scale of Edge-servers grows and the heterogeneity of networks between them increases. This work is driven by a simple question: can we place container images across Edge-servers in such a way that an image can be retrieved to any Edge-server fast and in a predictable time. To this end, we present KCBP and KCBP-WC, two container image placement algorithms which aim to reduce the maximum retrieval time of container images. KCBP and KCBP-WC are based on k-Center optimization. However, KCBP-WC tries to avoid placing large layers of a container image on the same Edge-server. Evaluations using trace-driven simulations show that KCBP and KCBP-WC can be applied to various network configurations and reduce the maximum retrieval time of container images from 1.1x to 4x compared to state-of-the-art placements (i.e., Best-Fit and Random). This work led to a publication at the ICCCN ′19 international conference [63].

Characterizing the Performance of Erasure Coding in Data-Intensive Clusters

Data-intensive clusters are heavily relying on distributed storage systems to accommodate the unprecedented growth of data. Hadoop distributed file system (HDFS) [263] is the primary storage for data analytics frameworks such as Spark [20] and Hadoop [19]. Traditionally, HDFS operates under replication to ensure data availability and to allow locality-aware task execution of data-intensive applications. Recently, erasure coding (EC) is emerging as an alternative method to replication in storage systems due to the continuous reduction in its computation overhead. Despite a large body of studies targeting improving the recovery overhead in EC in terms of network and disk overhead, it is unclear what is the performance characteristics of analytics jobs running under EC. In response, we conduct experiments to thoroughly understand the performance of data-intensive applications under replication and EC. We use representative benchmarks on the Grid'5000 [105] testbed to evaluate how data access pattern, concurrent data access, analytics workloads, data persistency, failures, backend storage devices, and network configuration impact their performances. While some of our results follow our intuition, others were unexpected. For example, disk and network contentions caused by chunks distribution and the unawareness of their functionalities are the main factors affecting the performance of Big Data applications under EC, not data locality. Part of this work led to a publication at the MASCOTS ′19 international conference [61].

Bringing EC-awareness to Hadoop Distributed File System (HDFS)

We observe that Hadoop task scheduler is not aware of the data layout under EC and can result in a noticeable skew in data accesses across servers when running data-intensive applications. This causes stragglers (i.e., some tasks exhibit a large deviation in their executions and take longer time to complete compared to the average task runtime) and, in turn, prolongs the execution time of data-intensive applications. Accordingly, in an attempt to improve the performance of data-analytics jobs under erasure coding, we propose an EC-aware data placement algorithm that balances data accesses across servers by taking into account the semantics of the chunks (i.e., data or parity) when distributing them. Our experiments on top of Grid'5000 [105] testbed show that EC-aware placement can reduce the execution time of Sort and Wordcount applications by up to 25%. Our results pave the way and motivate the integration of EC-awareness on the scheduling level to cope with the dynamicity of the environment.

1.3 Publications

Papers in International conferences

- **Jad Darrous**, Shadi Ibrahim, Amelie Chi Zhou, and Christian Perez. "Nitro: Network-Aware Virtual Machine Image Management in Geo-Distributed Clouds". In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (**CCGrid '18**).* Washington DC, USA, May 2018, pp. 553–562. (CORE Rank A)

- **Jad Darrous**, Thomas Lambert, and Shadi Ibrahim. "On the Importance of container images placement for service provisioning in the Edge". In: *Proceedings of the 28th International Conference on Computer Communications and Networks (**ICCCN '19**).* Valencia, Spain, July 2019, pp. 1–9. (CORE Rank A)

- **Jad Darrous**, Shadi Ibrahim, and Christian Perez. "Is it time to revisit Erasure Coding in Data-intensive clusters?" In: *Proceedings of the 27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (**MASCOTS '19**).* Rennes, France, Oct. 2019, pp. 165–178. (CORE Rank A)

Posters in International conferences

- **Jad Darrous** and Shadi Ibrahim. *Enabling Data Processing under Erasure Coding in the Fog.* In the 48th International Conference on Parallel Processing (**ICPP '19**). Kyoto, Japan, Aug. 2019

Papers in National conferences

- **Jad Darrous**. "On the (In)Efficiency of IPFS for Geo-distributed Virtual Machine Images Management". In: *Conférence d'informatique en Parallélisme, Architecture et Système (**ComPAS '17**).* Sophia-Antipolis, France, July 2017. [Conference without proceedings]

1.4 Software and Data

Nitro: Network-aware VMI management in geo-distributed clouds

Nitro is a Virtual Machine Image (VMI) management system for geo-distributed clouds. Nitro reduces the network cost and optimizes the retrieval time when provisioning a virtual machine on a site where its image is not available locally. Nitro leverages deduplication to reduce the size of the image dataset, and thus, reduce network cost when transferring images. Importantly, Nitro incorporates a network-aware chunk scheduling algorithm that produces the optimal solution to retrieve an image.

Programming languages	Python and Yaml
Codebase size	1500 LoC
License	GPL-3.0
Repository	https://gitlab.inria.fr/jdarrous/nitro

Container image placement simulator in Edge environment

This is an extendable simulator to test the performance of data placement and retrieval algorithms for layer-based container images (e.g., Docker images) in Edge like environment. It simulates the network bandwidths between the nodes in addition to their storage capacities. It already contains four placement algorithms and two retrieval algorithms. Examples of synthetic and real-world networks in addition to container image dataset are included.

Programming languages	Python and Yaml
Codebase size	1000 LoC (+ 1500 LoC for analysis scripts)
License	GPL-3.0
Repository	https://gitlab.inria.fr/jdarrous/image-placement-edge

Hadoop EC traces

This repository contains traces of Hadoop MapReduce jobs under replication and erasure coding. Sort, Wordcount, and K-means applications are included. These traces contain runs of different software (overlapping and non-overlapping shuffle, disk persistency, failure) and hardware (HDD, SSD, DRAM, 1 Gbps and 10 Gbps network) configurations.

Version	Apache Hadoop 3.0.0
Traces size	200 MB
Format	CSV and JSON
License	GPL-3.0
Repository	https://gitlab.inria.fr/jdarrous/hadoop-ec-traces

1.5 Structure of the Manuscript

The rest of this manuscript is organized into four parts as follows.

The first part presents the context of our research. Chapter 2 provides an overview of Big Data and the challenges related to Big Data management. Then, we list some examples of business and scientific data-intensive applications. Chapter 3 introduces cloud computing with its different models. Virtualization technologies are presented in more details followed by some examples of cloud offerings and cloud management platforms. Chapter 4 presents distributed storage systems, emphasizing on availability techniques, along with Big Data processing frameworks.

The second part consists of three chapters and presents our contributions regarding efficient service provisioning in distributed clouds. In Chapter 5, we start by defining the problem and discussing in detail related work on service provisioning in clouds. In Chapter 6, we address efficient provisioning of services in geo-distributed clouds. We introduce *Nitro*, a novel data management system that minimizes the retrieval time of VMIs over the WAN. In Chapter 7, we describe our two container image placement algorithms that reduce the maximum retrieval time for an image to any Edge-server.

The third part consists of four chapters and presents our contributions with regards to enabling efficient data processing under erasure codes (EC). Chapter 8 gives an overview of the application domains of EC and how it is implemented in HDFS. In Chapter 9, we study the performance of HDFS under EC. Chapter 10 evaluates the performance of data-intensive applications under both replication and EC with different system configurations. In Chapter 11, we propose an EC-aware data placement algorithm in response to our findings in previous chapters.

The fourth part consists of Chapter 12. In this chapter, we summarize our contributions and present our conclusions about provisioning and running data-intensive applications in the cloud. Finally, we discuss the limitations of the proposed solutions and describe the perspectives.

Part I

Background

Chapter 2

Data and Data-Intensive Applications

Contents

Big Data can be described as the oil of the twenty-first century [275]. Like oil, it has a great economic value and it empowers a wide range of applications that we rely on in everyday life. Therefore, acquiring data and analyzing it represent the new challenges in our era. Today's and tomorrow's data are produced from human activities on Internet (e.g., social media, e-commerce, and video streaming), scientific research (e.g., scientific simulation and computational models), as well as a growing number of sensors devices including mobiles and handheld appliances/gadgets. In this chapter, we introduce the Big Data challenges and list examples of data-intensive applications that are managing gigantic datasets.

2.1 The Data Deluge

The human race, throughout history, has always been willing to better understand the world. Understanding is the fruit of knowledge which is in turn acquired from the information we have. And finally, the information is a result of processing the data that we collect [4]. This highlights the importance of data in a better understanding of the world that can manifest in a wide range of business and scientific applications. Hereafter, we first present the data dimensions in Section 2.1.1. Then we categorize the data according to its temperature in Section 2.1.2. And finally, we describe the different data access patterns in Section 2.1.3.

2.1.1 Data dimensions

Data can be characterized by several dimensions, however, here we review the basic three ones that are first proposed by Doug Laney [167] back in 2001, and described as the 3'Vs model. These dimensions present the main *challenges* facing data management tools. These challenges include *Volume*, *Velocity*, and *Variety* [250, 295, 267, 313]. Later on, more V challenges are added as described in [217].

Volume. Perhaps the most obvious challenge that is facing Big Data applications. Volume represents the size of the data that we are dealing with. The size of our digital universe is growing exponentially, where data volumes double every two years [80]. While the estimated size of digital data which is generated up to the year 1999 was around 12 exabytes [118], the digital universe surpassed the 1.8 zettabytes by 2011 [91] with an increase of 150x and it is expected to bypass the 175 zettabytes by 2025 [130]. The current and future scale of data stress current management tools and require scalable infrastructures and solutions for efficient data management.

Velocity. The volume of the data does not capture its rate of generation or analysis. Many types of data, especially sensor data, are volatile and should be processed on the fly. For instance, 60% of valuable sensory data loses value in milliseconds [292]. The speed of which data is generated, captured, and processed characterize its velocity. For example, millions of cameras are deployed in large cities around the world generating video streams that should be processed in real-time [311]. Moreover, a self-driving car sends around one gigabyte of data per second to cloud servers to be processed and send back the control signals [1, 145]. This opens the door for a new type of data processing, identified as *stream processing*, where achieving low-latency is crucial.

Variety. Beside its volume and velocity, data have heterogeneous types and formats. This heterogeneity is the result of the diverse sources that generate the data. From Internet-based services and activities such as social media, e-commerce, maps service, and video-on-demand to scientific sensors and simulations. Unstructured free-form text, multimedia files, and geospatial data are some examples of different types of data. This variety of data formats imposes challenges for analytics frameworks as data from different sources should be analyzed collectively.

2.1.2 Data temperature

Data access frequency, usually associated with data temperature [168, 195, 32, 230], influences how data is managed. Data can be classified into three main categories depending on its temperature.

Cold data. Data that are rarely accessed are described as cold data. This category includes archived data, backups, and data stored in peer-to-peer networks among others. High latency and low throughout can be tolerated when accessing cold data; for example, these data may take days or hours to be retrieved, as it usually ends up stored on tapes or Blu-ray discs [81, 15].

Hot data. The extreme opposite of cold data. These data are frequently accessed. For instance, a breaking news post on social media. In general, those data are stored in high-speed storage devices (e.g., SSDs, DRAM, etc.) to ensure fast data access [230].

Warm and lukewarm data. This category falls in-between the previous two ones. These data have a request rate lower than that of hot data, however, it should be retrieved in acceptable time compared to cold one [168, 195]. For instance, photos and videos that are not popular but still requested by users can fall in this category.

2.1.3 Data access patterns

Data are created for various reasons and consumed in different ways. This will impact when and how much data is accessed. Hereafter, we list the four possible patterns for data accesses.

Write-once read-many. As the name implies, once these data are stored, they will not be modified but they will be read frequently. For instance, this category includes input data for analytics frameworks and service images (i.e., Virtual Machines Images and Container Images).

Write-once read-once. This category usually represents temporary and intermediate data that are read once after being created and then discarded. For instance, partial results of multi-stage computations (e.g., Hadoop MapReduce) are stored as inputs for later stages and deleted once read and processed.

Write-many read-once. This category usually includes data that is used to ensure fault tolerance during the execution of jobs. For instance, checkpointing data in HPC is written many times and read once in case of failure.

Write-many read-many. This category includes the remaining access patterns. Essentially, it describes data that could be written, read, updated, and deleted frequently. For example, the state of online multi-player games.

2.2 Data-Intensive Applications

Nowadays, data plays a central role in many businesses as well as scientific applications. Hereafter, we list some examples of commercial and scientific applications that are characterized as data-intensive applications.

2.2.1 Internet-based applications

Google. Google has 90% of the search engine market share worldwide, while serving billions of queries every day [21, 2]. Moreover, Google relies on a globally distributed database, Spanner [58], to support its advertising backend, which accounts for the largest share of Google total revenue. Furthermore, Google runs complex learning algorithms to build its recommender system for YouTube in order to improve users' experiences [59].

Facebook. With more than 2 billion active users per month, writing posts, uploading photos, and sending messages; Facebook is the largest social network platform nowadays. To satisfy the timely response time to users' interactions, Facebook relies on a homegrown storage system, TAO, that runs on thousands of machines, provides access to petabytes of data, and handles millions of requests per second [43]. Moreover, Facebook runs thousands of analytical queries per day [277] to show relevant content and personalized advertisements to its users.

2.2.2 Scientific applications

Large Hadron Collider (LHC). The largest machine in the world, the LHC is built by the CERN (European Organization for Nuclear Research) to advance the research in particle physics. The performed experiments generate a tremendous amount of data that need to be stored and analyzed. As of the beginning of 2019, CERN manages about 330 petabytes of data [65].

Square Kilometer Array (SKA). The SKA project [274] is a set of radio telescopes to push the boundary of our understanding of the universe; from galaxies, cosmology, dark matter, and dark energy to search for extraterrestrial life. When completed, the aggregated data from these telescopes will attend 7.5 petabytes per second [42]. Many challenges will be raised, not only for storing the data but also for transferring and analyzing it.

2.3 Conclusion

Our digital universe is expanding by the generation of tremendous and heterogeneous amount of data. The data is produced from a variety of sources with distinct characteristics. This trend gives the opportunity for scientific research to better understand the world, and for business to improve the quality of our lives. However, getting the most benefit of these data poses many challenges that stress the need for large-scale infrastructures and systems. In the next chapter (Chapter 3), we review the clouds as an enabling infrastructure for Big Data analytics. In Chapter 4, we present the storage systems and analytics frameworks for Big Data processing.

Chapter 3

Distributed Clouds: Empowering Infrastructures for Data-Intensive Applications

Contents

In his speech at MIT in 1961, John McCarthy envisioned computing as a kind of *Utility Computing* that could be used by the public as "Pay and Use" [95]:

"If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry."

Cloud can be considered as the realization of McCarthy vision in this century. Though virtualization technologies – which are the main technologies behind the cloud – are developed in the 70s, commercial clouds were popularized in the late 2000s by large enterprises, such as Amazon, Microsoft, and Google, that build and start to offer cloud services. This chapter reviews the basics of clouds and their various models.

3.1 Cloud Computing: Definition

There are many cloud computing definitions available in the literature. One of them is proposed by the National Institute of Standards and Technology of the U.S. Department of Commerce [272]:

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

In other words, cloud computing is a computing model that provides remote access to compute, storage, and networking resources that are managed by a cloud provider. These resources can be instantaneously accessible, scalable, elastic, and billed on a pay-as-you-go pricing model [140, 127].

3.2 Cloud Architecture: From Centralized Data Centers to Massively Distributed Mini Data Centers

Clouds architecture has been evolving over the years: from a single data center in a single geographical area, to massively distributed mini data centers that are distributed over the world. In this section, we categorize clouds architecture into four categories.

3.2.1 Single "region" clouds

In its basic form, cloud services can be delivered by a single data center. A data center is a building hosting all the computation, storage, and networking resources, in addition to non-computation ones such as generators, cooling systems, and batteries. In data centers, computing and networking equipment are housed within racks. A rack is a group of 30-40 servers connected to a common network switch. This switch is called the top-of-rack (TOR) switch. The top-of-rack switches from all the racks are interconnected through one or more layers of higher-level switches (aggregation switches and routers) to provide connectivity from any server to any other server in the data center. Typically, the top-of-rack and higher-level switches are heavily oversubscribed [8, 104, 146, 232, 53].

To tolerate a complete data center failure (for example, due to generators failures), cloud providers build two or more data centers close to each other in what is called Availability Zones (AZ) [27, 293, 100]. Data centers in each AZ are equipped with independent power, cooling, and networking facilities.

To provide more resiliency, an AZ has at least another AZ that is geographically located within the same area, linked by highly resilient and low-latency private fiberoptic connections. However, each AZ is isolated from the others using separate power and network connectivity to minimize the impact to other AZs in case of complete AZ failure. The collection of AZs that are geographically located close to each other is called a region [27, 293, 100].

3.2.2 Geo(graphically)-distributed clouds

Geo-distributed clouds are the set of regions that can be managed collectively (e.g., by a single cloud provider). These regions are usually spread around the world, to provide a better quality of service, conform to privacy regulations, and provide higher catastrophic fault tolerance. However, in the case of network partitioning, every region can act independently of the others. For example, major cloud providers such as Microsoft Azure, Amazon AWS, and Google Cloud have 54, 21, 19 geographic regions, respectively [296, 26, 101]. Usually, these regions are connected using wide area network (WAN). However, some large Internet companies are even building their own private WAN such as Google [137] and Microsoft [115].

As data centers are spread in different regions, the connections between them are heterogeneous (in terms of bandwidth and latency). For instance, the bandwidth in-between 11 sites in Amazon AWS varies by up to 12x [119]. This heterogeneity is one of the main challenges towards realizing efficient geo-distributed data management.

3.2.3 Federated clouds

Federated cloud is a model that offers cloud services from different, possibly public and private, vendors in a transparent way [44, 163, 302]. Federated clouds can be also referred by Multi-Cloud [163] and Inter-Cloud [44]. Federated clouds provide high scalability and cost-efficient workload execution, as well as, they lower the adverse effects of vendor lock-in. However, the challenge to realize this model is to provide interoperability between the different providers to allows seamless integration and migration of users' services. In contrast to geo-distributed clouds, federated clouds belong to different administrative entities. Nevertheless, federated clouds could be geographically distributed.

3.2.4 Edge computing

Edge computing is a recent computing paradigm that pushes the virtualized computation and storage resources to the edge of the network [262, 261, 294, 194]. For instance, Micro data centers [30, 77], Cloudlets [253], and Point of Presences (PoPs) are some examples of Edge-servers. This geographical proximity provides local and fast data access as well as it enables in-place data processing. Therefore, Edge can be leveraged to run latency-sensitive applications such as smart city applications, video processing applications, and augmented reality applications [304, 125, 243, 280]. Moreover, processing data at the Edge can greatly reduce the network traffic to the central cloud and improve the performance [229]. For instance, running smart city applications at the Edge shows to be up to 56% more efficient compared to clouds [243].

A similar concept to Edge computing is Fog computing which was formalized by CISCO in 2012 [40]. Even though most of the work in literature uses the two words interchangeably [262, 261, 114, 294, 194], the OpenFog Reference Architecture [207] distinguishes Fog computing as a hierarchical infrastructure that, in addition to the Edge and the cloud, leverages all the resources in-between (in the core of the network).

3.3 Service Models

The following three service models have been extensively used to categorize services provided by clouds [139]:

Infrastructure-as-a-Service — IaaS. This is considered as the most basic cloud model. It provides users with computing, storage, and networking resources. For instance, users rent computing resources in the form of virtual machines. These VMs are launched from custom images and have specific capabilities depend on the number of CPUs, memory size, locally-attached disks, etc. Moreover, storage is available for users through object-store interfaces (e.g., REST). This model is suitable for experienced users who want to have full control over their execution environments. Elastic Compute Cloud (EC2) [23], Simple Storage Service (S3) [28], and Elastic Load Balancing (ELB) [24] are some examples of computing, storage, and networking services, respectively, provided by AWS IaaS.

Platform-as-a-Service — PaaS. This model provides computing platforms which allow users to run their application without the complexity of maintaining the underlying infrastructure. This level of services allows users to concentrate only on the application while abstracting the complexity of managing the underlying hardware and software layers. This model includes programming language execution environments, databases, web servers, etc. For instance, Google App Engine is a PaaS that offers hosting for web applications in Google-managed data centers. Another example of PaaS is Amazon Elastic MapReduce (EMR) [25], where users are provided with a ready to use installation of Hadoop framework to run their MapReduce jobs.

Software-as-a-Service — SaaS. This is the highest level in the cloud service model. In this category, the cloud provider takes control of the infrastructure and the platforms to provide ready-to-use software. The main advantage of this model is providing easy access for end users and ensuring minimum configurations. For instance, Microsoft Office 365 [190] and Google suite (Google Doc, Google Sheet, Google Slides, etc.) [90] are well-known examples of SaaS.

3.4 Deployment Models

The following three models can characterize cloud deployment [68]:

Public clouds. Public clouds are the most common model of cloud computing. Cloud providers offer their resources on-demand in a pay-as-you-go model to all users [127]. Cloud users can lease and access these resources over the Internet.

Private clouds. Private clouds are built to deliver services for a single organization. In this case, it is the organization responsibility to manage the hardware as well as the cloud software manager (or delegating them to a third party). This model is preferable for organizations that want to have full control over the infrastructure and software, usually for high security and privacy guarantee. However, private clouds might suffer from limited scalability and lower resource utilization as the workloads change over time.

Hybrid clouds. In this model, both public and private clouds are used at the same time. The common scenario for hybrid clouds is when an organization deploying its private cloud wants to offload some burst less-sensitive computations to the public cloud. Hybrid clouds could be considered as a special case of federated clouds [302], therefore, compatibility issues represent the main challenge to be considered to enable efficient resource migration between the two clouds.

3.5 Virtualization as Cloud-enabling Technology

Clouds services run in virtualized environments, usually as Virtual Machines or containers. Virtualized environments greatly reduce the overhead of service management, provide isolation between different tenants, and increase resource usage by means of consolidation.

3.5.1 Full virtualization: Virtual machines and Unikernels

Virtual Machines

A virtual machine (VM) is a computer program capable of running an operating system and applications [265]. It runs on top of a physical machine but gives the illusion of a separate physical machine to the user. Therefore, multiple VMs can share the same physical machine. The software responsible for managing resources of VMs in the same physical machine is called a virtual machine monitor or hypervisor (e.g., Xen [299], KVM [150], Microsoft Hyper-V [191], VMware vSphere [287], etc.).

A VM is characterized by the number of vCPUs (virtual CPUs) and memory size, in addition to the disk image. A cloud user can choose the VM type (according to the number of vCPU and memory size) from a catalog provided by the cloud provider with a different cost for each type. The disk image of the VM (the VM image) is usually provided by the user and it contains the operating system and application-specific software.

Unikernels

A unikernel [278, 185] is a specialized, single-address-space machine image constructed by using library operating systems such as MirageOS [185], ClockOS [188], OSv [155], Tinyx [186] etc. A unikernel consist of a single process which (generally) runs one application. Thanks to this single-minded design, unikernels offer fast boot time, high performance, low memory footprint, and reduced attack surface [278, 185, 186]. Also, like Virtual Machines, unikernels can run directly on a hypervisor.

Recently, unikernel are gaining popularity in cloud [278, 185, 188, 155] especially for mission-specific applications such as Network Function Virtualization [185, 188]. However, there are several challenges which hinder the wide adoption of unikernels in clouds, especially the complexity of building and packaging them and the difficulty of porting existing applications [206].

3.5.2 Operating-System-level virtualization: Containers

The introduction of lighter virtualization technologies has been motivated by the high overhead encountered with VMs, as numerous studies have shown [257, 85]. *Containers*, *Zones*, or *Jails* are a form of Operating-System-level virtualization that runs directly on top of the host machine Operating-System. This eliminates the need for a hypervisor and reduces the size of their disk images which results in their lightweight overhead [257]. Treating a container as a lightweight VM is not accurate; for example, complete emulation of different hardware architecture is not possible with containers. Linux cgroups and namespaces are the underlying Linux kernel technologies used to isolate, secure, and manage containers [257]. While these technologies have been included into the Linux kernel for over a decade [257], the widespread deployment of containers starts for real after the emergence of Docker [72]: Docker introduced a layered file system (that represents the container image) as well as many software engineering benefits to ease container management.

3.5.3 On the role of service images in the cloud

All services in the cloud are launched from a service image; a virtual machine image (VMI) or a container image. The service image should be available on the host machine to run the service, otherwise, it should be transferred from a central image repository. Such an image includes the software program of the services as well as all its dependencies including the operating system itself in case of VMIs. Therefore, these images could be large in size; up to one gigabyte for a container image [109, 17] and up to tens of gigabytes for a VMI [35, 258]. Moreover, large number of images are now hosted in clouds, for instance, at least 2.5 million images [74] are stored in Docker Hub [73], while AWS has more than 20,000 different public images[1].

Many software solutions are available to build these images. These solutions, besides the construction of the required image, focus on the easiness of the operation, the diversity of output image formats, and the reproducibility of the build process. For instance, Docker [72] can create container images from a `dockerfile` that describes the building instructions. Packer [215], Oz [214], and Kameleon [249] are some examples of solutions for building VM images.

3.6 Public Cloud Offerings

Public cloud offerings are exploding over the years. Nowadays, dozens of public cloud providers are offering a wide range of services. Hereafter, we list three of the most popular public cloud providers.

[1]We obtained the number of images from Amazon EC2 Dashboard on Jul. 26th, 2019.

Amazon Web Services (AWS). Initially launched in 2006, Amazon Web Services is a collection of IaaS and PaaS services. The first two services were Elastic Cloud Computing (EC2) [23] and Simple Storage Service (S3) [28] that provide on-demand computation and storage, respectively. Now, AWS offers include dozens of other services such as Elastic MapReduce (EMR) [25], Elastic Container Service (ECS) [13], etc. As of 2017, AWS has a market share of 34% of cloud offers, more than the three closest competitors combined (i.e., Google, Microsoft, and IBM) [244].

Google Cloud Platform (GCP). Initially released as a PaaS (i.e., Google App Engine) in 2008, now, GCP offers a variety of IaaS and SaaS services. Kubernetes Engine [103] (for container orchestration) and Cloud BigTable [54] (for NoSQL databases) are some examples of Google cloud offers. The offered cloud computing services run on the same infrastructure that Google uses internally for its end-user products, such as Google Search and YouTube.

Microsoft Azure. Microsoft Azure, officially released in 2010, provides IaaS, PaaS, and SaaS cloud services. For instance, Azure Data Lake [64] (for scalable data storage) and Azure HDInsight [112] (for Big Data processing) are among the services provided by Azure. Currently, Azure has the largest public cloud infrastructures with 54 regions around the world counting for more than 140 Availability Zones (each containing at least two data centers) [296].

3.7 Cloud Management Solutions

Several management solutions have been developed to facilitate the orchestration, the configuration, and the automation of virtual machines and containers in virtualized clusters. In this section, we list some popular frameworks while presenting in detail OpenStack [209] as the de-facto platform for IaaS clouds and Kubernetes [159] as the most popular container orchestration framework.

OpenStack. OpenStack [209] is an open-source IaaS manager for public and private clouds, licensed under Apache 2.0. OpenStack aims at providing a massively scalable and efficient cloud solution for corporations, service providers, small and medium enterprises, and researchers. OpenStack consists of 5 core projects; *Nova* provides access to on-demand, elastic, and scalable virtual machines similar to Amazon EC2. *Glance* is the component responsible for the management of the virtual machine images, however, it relies on a backend storage system for the actual storage of these images. *Swift* is the distributed eventually-consistent object-store project for OpenStack. It is also used as a backend for Glance [57]. *Neutron* is responsible for the networking part while *Keystone* represents the identity management component. OpenStack was born in 2010 as a joint project of Rackspace Hosting and NASA. Currently, it counts more than 500 collaborators. Due to its success, OpenStack is used by many enterprises (e.g., Walmart and Verizon), universities and research centers (e.g., CERN and University of Edinburgh), as well as governmental entities (e.g., UK Civil Service and France's Interior Ministry). Moreover, at the time of writing, 18 public cloud providers use OpenStack to empower their infrastructures [273].

Docker. Docker [72] is a popular opens-source container management framework. Docker Engine orchestrates the execution of containers on a single machine. Also, Docker provides an image service (i.e., Docker registry) to management the container images. Docker has the Swarm mode to deploy containers on a cluster of machines [268]. However, Docker Swarm mode is surpassed by Kubernetes described next.

Kubernetes. Kubernetes [159] is open-source software for deploying and managing containers, including Docker containers, at scale. Kubernetes works by managing a cluster of compute instances and scheduling containers to run on the cluster based on the available compute resources and the resource requirements of each container. Containers are grouped into pods, the basic operational unit for Kubernetes. Containers belonging to the same pod are deployed to a single machine sharing the IP address and hostname. Pods abstract network and storage away from the underlying container which facilitates the management of the containers (e.g., migration and scaling). Kubernetes supports efficient management of multiple clusters. These clusters can span hosts across public, private, or hybrid clouds.

OpenNebula. OpenNebula [208] is an open-source cloud computing platform which provides an enterprise IaaS ready deployment for private clouds. It fully supports efficient management of virtualization over computing, storage, and network resources. OpenNebula design aims to provide a standardization of IaaS clouds with full support of interoperability and portability e.g., OpenNebula supports several cloud interfaces and hypervisors. OpenNebula started as a research project in 2005 and the first public release was in 2008.

Eucalyptus. Eucalyptus [204, 79] is an open-source cloud manager that provides full compatibility with AWS services for building private and hybrid clouds. This provides a seamless migration of services between private and the public AWS cloud. Organizations can use AWS-compatible tools, images, and scripts to manage their own on-premises IaaS environments. The Eucalyptus started as a research project at University of California in 2007. After two years, the project was commercialized through Eucalyptus company.

3.8 Conclusion

Clouds provide a large-scale pool of resources that can be leveraged to store and process data. Data analytics services in clouds are leased as Virtual Machines or containers in virtualized environments to allow management flexibly and resource efficiency. While important, virtualization poses a new challenge in the Big Data era, which is related to the management of VMIs and container images. In addition to their sizes, which could be infeasible to store locally (especially in Edge-servers), retrieving and placing these images is rather important as it has a direct impact on the performance of service provisioning. Hence, in Part II, we present our contribution towards efficient service provisioning in distributed clouds.

Chapter 4

Data Management Systems: An Overview

Contents

In his last talk, Jim Gray described how using the Big Data to extract useful information and push the boundary of science as a *paradigm shift* [113]. To enable this paradigm shift, there is a call for new data management systems, in particular, storage systems that can store and mange this tremendous amount of data at scale and data analytics frameworks that can facilitate large-scale data processing. In this chapter, we discuss current state-of-the-art distributed storage systems including the ones focusing on services images and input data of data-intensive applications and then present the main frameworks to enable fast and efficient Big Data analysis.

4.1 Design Objectives of Distributed Storage Systems

Distributed storage systems (DSSs) are designed to aggregate a large number of nodes to give the impression of a single large machine. However, storage systems have different design objectives. For instance, archival systems focus on durability at low cost while caching systems aim for high performance and availability. These design objectives are conflicting and cannot be achieved at once. For example, data consistency and data availability cannot be guaranteed at the same time in case of network partitioning, which is known as the CAP theorem [41]. In this section, we present the main objectives when designing distributed storage systems. As our goal is to achieve scalable and efficient data management in the large-scale clouds where ensuring data availability is a must, we dedicate the next section to discuss data availability and highlight the performance and storage cost of the main techniques used to ensure data availability.

Performance. By performance, we refer to the performance of data access (i.e., read and write). Performance could be measured by the latency or the throughput (byte per second, request per second, etc.) of accessing the data. For example, in-memory caching systems aim for low access latency while data analytics backends storage optimize data access throughput.

Scalability. Scalability is the property of a system to handle a growing amount of work by adding resources to the system [39]. Scalability could be *horizontal* by adding new machines to the system, or *vertical* by making machines more powerful.

Data availability. Data availability means that data is available and accessible when requested (read, written, updated). Data could become unavailable due to hardware or software errors that render the data partially or completely inaccessible for a period of time. When a piece of data is updated, it also becomes unavailable until all replicas are updated (in case a strong consistency is applied, see below).

Data durability. Data durability refers to the long-term survival of the data. Data durability is particularly important in archival systems where the data should be stored *forever*. Similar to data availability, data durability is mainly impacted by software bugs and hardware failures that may result in permanent data loss.

Storage cost. Storage cost represents the total physical size of the actual stored data including the metadata, replicas, etc. For instance, lossless compression can be used to reduce the data size. Deduplication is another example of data compression technique. Usually reducing storage cost incur extra computation overhead, and its effectiveness depends on the type of the data.

Data consistency model. Data in a storage system could have different consistency models. For instance, strong consistency means that replicas of the same data are always perceived in the same state, however, this results in limited availability, performance, and scalability. Eventual consistency, on the other hand, allows some temporal inconsistencies at some points in time but ensures that all replicas will converge to a consistent state in the future [67, 166]. Other consistency models are also available (e.g., weak consistency, causal consistency, etc.) however, they are out of the scope of this thesis.

4.2 Data Availability in Distributed Storage Systems

Data should be available to be processed, however, with the scale of current DSSs, fail-stop failures and transient failures are common [98, 88, 106, 136, 96]. These failures, that might have either software or hardware root, impact the availability of data. In this section, we present two widely used techniques to ensure data availability in DSSs.

4.2.1 Replication

Replication is simply repeating the original data; the same piece of data is copied (replicated) to distinct failure domains (e.g., disks, machines, racks, data centers, regions, etc.) in order to tolerate maximum resource unavailability. For simplicity, we refer here to any of these failure domains as nodes. The replication of data could be performed synchronously (i.e., a piece of data is only available once all its replicas are persisted, or at least cached, in the DSS) or asynchronously (lazy replication).

Recovering a lost replica can be performed by re-replicating any of the surviving replicas. This comes with a network and (disk) read cost equals to the data size, and has a negligible CPU overhead. Moreover, under failure, the storage system can still serve users' requests by directing the requests to a life replica. This introduces almost no overhead except for the failure detection time of the failed node. Besides, replication can provide efficient data access, in terms of latency and/or throughput, under high data request rates. Users' requests could be served by any replica to balance the load between nodes. Selective replication, which assigns distinct replication factors for each object, is employed to increase the performance of data access for popular data by increasing their number of replicas [308].

On the other hand, replication has a high storage overhead, for example, a 200% storage overhead is needed to maintain 3 replicas, that can tolerate two simultaneous failures. Moreover, replication can impact the write latency of data as the write request should wait for all the replicas to write the data before being considered successful. This is important to ensure the consistency of the data. However, maintaining consistency is usually expensive and limit the scalability of the system, especially for applications with heavy writes pattern.

Replication has been employed as the main redundancy technique in many DSSs [263, 291, 98] for two main reasons: (1) its simplicity (and thus less error-prone implementations) and (2) as the storage is considered a *cheap* resource compared to computation and network (thus the overhead is acceptable).

4.2.2 Erasure coding

Erasure coding (EC) is a data redundancy technique that can provide the same fault tolerance guarantee as replication but with lower storage overhead [247]. Erasure codes have been widely used in various data storage systems, ranging from disk array systems [48], peer-to-peer storage systems [290], distributed storage systems [263, 291, 86], to cloud storage systems [78, 195], and in-memory caching systems [312, 230, 256, 307].

Reed-Solomon codes (RS) [239], and more generally *Maximum Distance Separable* (MDS) codes [184], are the origin of erasure codes, and therefore, they are the most deployed codes in current systems. $RS(n, k)$ splits the piece of the data to be encoded into (n) smaller chunks called *data chunks*, and then computes (k) *parity chunks* from these data chunks. This collection of $(n + k)$ chunks is called a *stripe*, where $n + k$ is the stripe width. In a system deploying an RS code, the $(n + k)$ chunks belonging to a stripe are stored on distinct failure domains (for similar reasons as replication, see the previous section). MDS codes (e.g., RS codes) have the property that any (n) out of $(n + k)$ chunks are sufficient to reconstruct the original piece of data, thus, MDS codes can tolerate (k) simultaneous failures.

For example, suppose the case for $RS(2, 2)$ where the two data chunks (d_1 and d_2) have the values of a and b respectively. The two parity chunks (p_1 and p_2) could have the values of $p_1 = a + b$ and $p_2 = a + 2b$. In this case, d_1 could be recovered by computing $d_1 = p_1 - d_2$ and d_2 could be recovered by $d_2 = p_2 - p_1$.

Figure 4.1 depicts the workflow of EC; *encoding* is the operation of generating the parity chunks where *decoding* is the reconstruct operation. Encoding and decoding operations are considered CPU intensive.

RS codes present a trade-off between higher fault tolerance and lower storage overhead depending on the parameters (n) and (k). $RS(6, 3)$ and $RS(10, 4)$ are among the most widely used configurations.

EC vs. replication

Erasure codes can achieve high reduction in storage overhead compared to replication. For instance, $RS(6, 3)$ has a storage overhead of 50% (for each piece of data that is split into 6 data chunks, 3 other parity chunks are needed) and delivers the same fault-tolerance as 4-way replication that incurs a storage overhead of 300% (for each piece of data, three other identical replicas are needed). On the other hand, in addition to CPU overhead, EC brings considerable network and disk overhead in case of failure (i.e., data unavailability or data loss). In particular, to reconstruct a missing chunk, n chunks (original and/or parity) should be read (from disk) and transferred over the network. Therefore, the reconstruction cost of a chunk is n times its size in terms of (network) data transfer and (disk) read.

EC families

Besides Reed-Solomon (RS), we present here some example of erasure codes that are widely deployed in storage systems. Local Reconstruction Codes (LRCs) are also well-known codes that have been deployed in production clusters [123, 252]. LRCs split the data chunks into m groups, one *local parity chunk* is computed for each group in addition to m *global parity chunks*. Parity chunks are usually computed using RS codes. The chunks of each group are often sufficient to recover a missing chunk of the same group, otherwise, global parities are used. Therefore, LRCs reduce the reconstruction overhead of missing chunks compared to RS codes in terms of network bandwidth and disk I/O as fewer chunks are needed. However, LRC codes are not MDS codes, and therefore, cannot tolerate arbitrary failures. Regenerating codes achieve an optimal trade-off between the storage overhead and the amount of data transferred (repair traffic) during the repair process [69]. They are traditionally employed in peer-to-peer systems, and later optimized to minimize disk I/O in distributed storage systems [233]. XOR codes are simple and fast codes that create one parity chunk which is the bitwise xor of all the data chunks. However, only one data loss can be tolerated. Finally, erasure codes can be seen as a software RAID [48], for instance, RS codes are used to implement RAID-6.

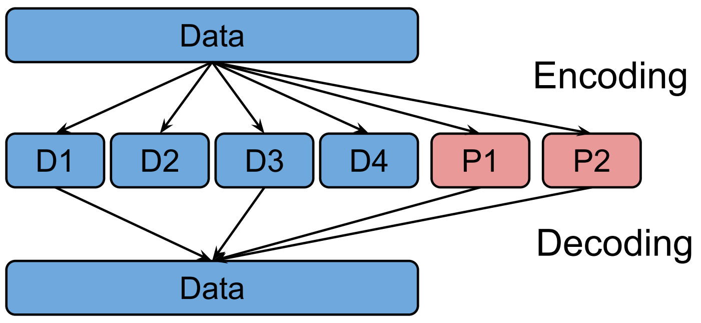


Figure 4.1: Erasure coding: encoding and decoding with $RS(4, 2)$. The original piece of data is split into 4 data chunks, and 2 parity chunks are computed.

4.2.3 Data deduplication: Optimizing storage cost under availability constraints

Direct application of redundancy techniques discussed in the previous sections could result in high storage overhead. However, data of the same applications may exhibit high similarities (e.g., VMIs and container images) and this introduces new opportunities to reduce their sizes without impacting the availability of the storage system. Deduplication technique can greatly reduce the storage overhead while maintaining the same level of availability of data. For instance, deduplication has been adapted in archiving systems where datasets contain a lot of duplicates [321].

Deduplication remove duplicated segment of data from a dataset, by keeping only unique segments. Each piece of data is represented by a list of pointers to the unique segments in order to reconstruct it. Figure 4.2 depicts the deduplication process on a dataset.

Deduplication can be applied on the binary-level or object-level. The *binary-level* treats the data as a raw array of bytes and creates segments from this array (could have the same size or not). On the other hand, the *object-level* treats each object of the data set as a segment of data.

The most common method to identify identical segments is the use of cryptographic hash functions. Segments that have the same hash digest are considered identical. Even though the probability of hash collision is very small, but it is not zero. Therefore, identical segment comparison (when their hash digests are matched) should be used to avoid data corruption. Moreover, hash functions are CPU intensive, therefore, to enable online deduplication, dedicated computation servers or devices could be used.

Deduplication is often paired with data compression for additional storage saving: Deduplication is first used to eliminate large segments of repetitive data, and compression is then used to efficiently encode each of the stored segments (chunks).

4.3 State-of-the-art Distributed Storage Systems

As the focus of this thesis is scalable and efficient Big Data management in distributed cloud, we list here state-of-the-art storage systems that are widely used to store service

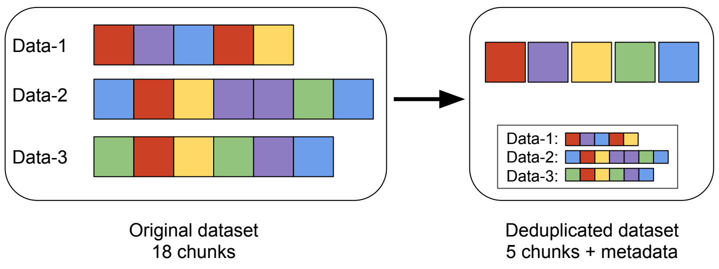


Figure 4.2: Deduplication: an example dataset where deduplication reduces its size from 18 chunks to 5 chunks plus the metadata.

images or employed as underlying storage system for data analytics. In particular, we present first OpenStack Swift [210] and HDFS [263] which are the most deployed storage systems for image storages and data processing, especially in private distributed clouds. Then, we describe two representative storage systems in public distributed clouds (i.e., Amazon S3 [28] and Apache Cassandra [166]). We also discuss new evolving storage systems which are optimized for high speed storage devices (i.e., DRAM) including Alluxio [11] and Redis [238]. Finally, we present a new emerging file system that can be used in geo-distributed deployment, IPFS [38].

OpenStack Swift

OpenStack Swift [210] is a cloud-based distributed object store. It provides a simple key/value dictionary interface for accessing the data. To achieve high scalability, Swift relies on eventual-consistency as consistency model. Swift is optimized for storing unstructured large binary files which have a write-once read-many access pattern (e.g., VM images, log files, backups, multimedia files, etc.). For instance, Swift is used as a *backend store for the image service* in OpenStack (Glance) [57]. Swift consists of a set of *proxy* nodes and a set of *object* nodes. The object nodes are responsible for the data storage while the proxy nodes receive users' requests and forward them to the appropriate object node. Swift organizes objects into containers, where a container is a logical grouping of a collection of objects. Access policies and redundancy level (i.e., replication and EC) are applied on the container level. Swift supports geographically distributed clusters, thanks to the read and write affinity properties: regions can be statically prioritized to favor read and write from/to "nearby" sites (i.e., normally sites that have a higher bandwidth between them).

Hadoop Distributed File System (HDFS)

HDFS [263] is a distributed file system that is designed to store multi-gigabytes files on large-scale clusters of commodity machines. HDFS is optimized to access large files. This is achieved by relaxing the POSIX interface (e.g., random write inside a file is not supported). To ease the management of its data, files stored in HDFS are divided into blocks, typically with a size of 64 MB, 128 MB, 256 MB, etc. HDFS employs replication to ensure data availability in case of failures. By default, each block is replicated on *three* different machines, with one replica in another rack. HDFS is used as *backend storage for analytics frameworks* (e.g., Hadoop [19], Spark [20], and Flink [18]) to store input and output data. HDFS consists of a *NameNode* and a set of *DataNodes*. The *NameNode (NN)* process is responsible for mapping each file to a list of blocks and maintains the distribution of blocks over the *DataNodes*. The *DataNode (DN)*, on the other hand, manages the actual data on its corresponding machine.

AWS Simple Storage Service (S3)

AWS S3 [28] is a cloud-based object storage that is part of the IaaS offers of Amazon. S3 has a key/value interface for accessing the data. Similar to OpenStack Swift, S3 uses eventual consistency to achieve high data availability at scale. Virtual Machine Images and analytical datasets are among the data that are usually stored in S3.

Cassandra

Cassandra [166] is a highly available, scalable, distributed storage system which was introduced by Facebook. Cassandra is designed for managing large objects of structured data spread over a large amount of commodity hardware located in different data centers worldwide. For its data model, Cassandra uses tables, rows, and columns, but unlike a relational database, the names and format of the columns can vary from row to row in the same table. However, the stored values are highly structured objects. Such a data model provides great abilities for large structured data, as it offers a more flexible yet efficient data access. Moreover, Cassandra provides a tunable consistency model per operation.

Alluxio

Alluxio [11] (successor of Tachyon [170]) is an in-memory caching solutions for Big Data analytics which is employed in modern data-intensive clusters. Alluxio provides data orchestration for analytics and machine learning applications in the cloud. In essence, Alluxio is a middleware that sits on top of a disk-based storage system (e.g., S3, HDFS, etc.) and provides fast data access (e.g., through memory data locality) to accelerate the execution of computation workloads.

Redis

Redis [238] is an open-source in-memory distributed key-value store. Redis is one of the most popular NoSQL databases in the industry. It provides low access latency for small objects. Redis supports a richer interface than a simple key/value store. For example, lists, sets, hash tables, streams, and geospatial data are supported. Redis uses replications for higher data availability. Moreover, Redis supports optional durability by persisting data to disk.

InterPlanetary File System (IPFS)

IPFS [38] is a peer-to-peer distributed file system that is designed for massively distributed environments. IPFS stores blocks of data indexed by their fingerprint (i.e., cryptographic hashes) and therefore can perform deduplication at the complete file system level. Furthermore, IPFS exchanges chunks using a BitTorrent [56] inspired protocol, named BitSwap. However, to reduce the complexity of finding the optimal plan to pull a file – due to the large number of replicated chunks – IPFS simply pulls chunks from all available sites and therefore introduces a high network overhead.

4.4 Data Analytics Frameworks: Turning Volumes into Value

To extract knowledge from Big Data, data processing frameworks are employed to run analytics jobs on it. In this section, we first present the key features of data analytics frameworks and then list some of the most widely used open-source frameworks.

4.4.1 Key features of data analytics frameworks

Distributed analytics frameworks are designed to enable running data-intensive applications on large-scale infrastructures. Hereafter, we present the key features that characterize these frameworks.

Performance. Analytics frameworks try to achieve high performance, however, performance metrics differ depending on the type of computation. For instance, batch processing frameworks focus on optimizing the throughput, while interactive and streaming processing aim to minimize the response time. Other metrics such as resource utilization and energy efficiency can also be used as performance metrics.

Scalability. This means that applications are parallelized into possibly thousands of tasks that are distributed and concurrently executed in a cluster.

Programming model. Analytics frameworks provide users with easy-to-use programming model and primitives. Programs written in these frameworks are usually sequential, however, they are automatically parallelized before being executed. This facilitate the adoption of these frameworks by developers.

Fault tolerance. Failures are common at large-scale, therefore, having tasks killed during the execution is inevitable. Handling failures gracefully while maintaining the continuity of the processing is, therefore, essential.

Generality. Traditionally, analytics frameworks focus on batch processing. However, currently, analytics frameworks support many other applications such as stream processing, SQL queries, graph processing, and machine learning applications.

4.4.2 State-of-the-art data analytics frameworks

In this section, we describe first the original MapReduce programming model [66, 141], then its open-source implementation, Hadoop [19], which is considered as the de-facto framework for data analytics. Next, we review Spark [20] and Flink [18], two analytics frameworks that support a wider range of applications.

MapReduce

MapReduce is a programming model – originally popularized by Google [66] – to provide an easy-to-use and scalable parallel programming model [141]. MapReduce applies sequential functions provided by the user (i.e., map and reduce) on a dataset in parallel. The MapReduce framework is responsible for task creation and scheduling, handling failures, and managing the computation and storage resources. Even though the MapReduce model is independent of the underlying hardware architecture (e.g., it could be implemented as multi-threading on top of shared memory), it has mainly employed for processing large datasets stored in distributed file systems (DFSs), e.g., Google File System (GFS) [98].

Originally, MapReduce was designed to run over commodity servers; machines that are error-prone, have limited computation and storage capacities, and connected with low network bandwidth. Therefore, to achieve parallel and scalable data processing, MapReduce has two design features: First, mitigating failures by relaunching only the tasks of the failed machines. Second, adopting *moving computation to data* principle to reduce data traffic over the network which results in more efficient execution. In essence, if a machine holding the input data of a task is not available (i.e., computation resources are occupied running other tasks), the task can be run on another machine which has a replica of the same data. MapReduce framework benefits from data replication employed by underlying DFSs to increase the data locality of running (map) tasks.

Hadoop MapReduce

Apache Hadoop [19] is the de-facto system for large-scale data processing in enterprises and cloud environment. Hadoop MapReduce framework is an open-source implementation of the Google MapReduce [66] in Java programming language. Nowadays, Hadoop is widely used for Big Data processing by both academia and enterprises [222]. Moreover, MapReduce algorithms have been used to solve several non-trivial problems in diverse areas including data processing, data mining, and graph analysis [187].

Hadoop MapReduce jobs run on top of Hadoop Distributed File System (HDFS) [263] (which is inspired by Google File System [98]) to read their input data and write their outputs. However, other storage systems can also be used (e.g., OpenStack Swift and Amazon S3).

In the early releases of Hadoop, resource management and job scheduling were handled by the framework. A master node runs the *job tracker* to globally manage the execution of jobs while each worker machine runs a *task tracker* which is responsible for managing the task executions on the worker node that hosts it.

Starting from version 2 of Hadoop. The resource management is refactored out from the original code into a separate project, Yet Another Resource Negotiator (YARN) [281]. In YARN, the resource management and job scheduling are separated; The *ResourceManager* daemon has a global view on the resources of the cluster and orchestrates them between the applications, while the *ApplicationMaster* is a per-application process that is responsible for the scheduling and the monitoring of a single job. Each worker machine runs a *NodeManager* process who is responsible for managing the local resources of the machine.

In a typical Hadoop deployment, one or more dedicated nodes run the master processes of HDFS and YARN (i.e., the *NameNode* and the *ResourceManager*) while other nodes in the cluster act as workers and run the slave processes (i.e., the *DataNode* and the *NodeManager*). In the rest of this thesis, we refer to Hadoop MapReduce as Hadoop for simplicity.

Spark

Apache Spark [20] is an open-source computing framework that unifies streaming, batch, and interactive Big Data workloads to unlock new applications. It fills the gaps where Hadoop cannot work efficiently, in particular, iterative applications and interactive analytics [310]. To realize efficient processing, Spark relies on an in-memory data structure (Resilient Distributed Dataset (RDD)) that stores the partial and intermediate results in memory, which avoid unnecessarily disk access in case of iterative and interactive applications. Moreover, Spark extends the API of MapReduce to richer transformations such as FlatMap, Filter, Join, etc. which enables writing more powerful applications. Many large companies rely on Spark for their data processing pipeline as Yahoo, Baidu, PanTera, and TripAdvisor [224].

Flink

Apache Flink [18] is an open-source framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink leverages a processing paradigm that unifies all types of processing (including real-time analytics and batch processing) as one unique data-stream model. Flink is based on a distributed streaming dataflow engine which is written in Java and Scala. Flink is designed to process real-time streaming data, and to provide high throughput with low latency streaming engine. Besides stream processing, Flink also provides support for batch processing, interactive processing, graph processing, and machine learning applications. Flink is widely employed in industry and large enterprises such as Netflix, Alibaba, Ebay, and Uber [223].

4.5 Conclusion

To perform Big Data processing in clouds, large-scale storage systems and analytical frameworks are indispensable to deal with the ever-growing datasets. Besides scalability, distributed storage systems have to ensure the availability of data during the processing. At the same time, analytics frameworks should efficiently leverage the underlying storage systems for higher performance. In Part III, we investigate erasure codes as a cost-effective redundancy technique in storage systems for efficient data analytics. We characterize the performance of data-intensive applications under EC and we propose a data placement algorithm to improve their performances.

Part II

Optimizing Service Provisioning in Distributed Clouds: Data Placement and Retrieval

Chapter 5

State-of-the-art: Service Image Management in Distributed Clouds

Contents

Clouds have become the dominant platforms for running a wide range of services and applications including enterprise applications [198] and scientific applications [289]. The main reason behind such wide adoption is the promise of providing fast and agile service deployment in a cost-effective manner [97] and therefore responding quickly to the changing demands of users. Thus, it is important to ensure fast service provisioning (i.e., the process that occurs between requesting the service and when the service is running and ready to handle users' requests).

In general, to start a service in the cloud, a disk image (virtual machine image or container image) is required to be available locally on the target machine. Such image includes an operating system (in case of VMI) and a "service-specific" customized software stack. Therefore, they could be large in size (i.e., up to tens of gigabytes [35, 258]). Usually, services' images are created once, stored centrally, and then transferred to the target machine(s) when the service is provisioned. Given their sizes, it has been pointed that transferring these images (when not available locally) from the central storage repository to the compute hosts is the dominant part of the provisioning process [219, 109]. This highlights the importance of image retrieval in the service provisioning process.

The number of cloud services and applications is continuing to increase and therefore the number of VMIs to run them. Accordingly, cloud providers are experiencing a new phenomenon called *Image Sprawl* [241], which is due to the large number of possible combinations of different operating systems and software stacks, and the need to keep older versions of the images for archiving or for reproducibility reasons. This results in gigantic image datasets and stresses the importance of efficient image management at scale to provide fast service provisioning. Furthermore, scalable image management is especially important in the new emerging infrastructures such as geo-distributed clouds and Edge environments which are featured with heterogeneous WAN links [119] – thus amplify the impact of image transfer – and “limited” storage capacities.

In this chapter we present an overview of image management for virtual machines and containers in Sections 5.1 and 5.2, respectively. Afterward, in Section 5.3, we discuss the related work on image management optimizations to improve service provisioning in clouds. Finally, Section 5.4 concludes this chapter by highlighting some challenges.

5.1 Virtual Machine Image Management

Virtual Machine Images (VMIs) are considered the basic building block for services in clouds. A service image encapsulates the software of the service in addition to its dependencies including the operating system itself.

In this section, we first present the different formats used to represent VMIs. Then, we briefly introduce the notion of the image repository. Finally, we describe the workflow of VM provisioning in clouds.

VM Image formats

VMIs are currently distributed as disk-image files, which are files that mirror the content of physical disks. The most basic and compatible format is the raw image (.img) that contains sector-by-sector contents of a disk. Other formats are available to provide more sophisticated features such as the support for sparse disk, copy-on-write, compression, encryption, snapshotting, etc. For example, the `qcow2` [189] disk image format uses a disk storage optimization strategy that delays allocation of storage until it is needed which results in smaller disk image size. Furthermore, several vendor-specific disk-image formats are also available, including `vdmk` from VMware [284], `vdh` from Microsoft [192], and `vdi` from VirtualBox [282].

Image repository

Image management in the cloud is usually dedicated to an image service. This service provides a library for the images or an *image repository*. In addition to storing and serving the images, the image service provides more functionalities including browsing, access control, and deployment. Moreover, features such as image content introspection and manipulation capabilities can also be provided. For actual data (i.e., images) storage, image services usually rely on backend storage systems. For instance, in OpenStack, the image service (Glance) relies on a distributed object-store (Swift) to persist the VMIs.

VM provisioning workflow

The service provisioning process can be divided into two major parts including (i) the scheduling process: choosing which server to run the VM, and (ii) the launching process: the process of starting up the VM on the chosen server. This includes transferring the image (from the storage nodes to the compute node, if it is not available locally) and booting up the VM by the hypervisor.

Many works are dedicated to optimizing the scheduling process (i.e., the mapping between VMs and physical machines) to optimize resource utilization, monetary cost, or energy efficiency [260, 176, 46, 251, 228, 154, 221]. However, in this work, we focus on the second part (i.e., the launching process). To start a service on a server, the image should be read from the image repository, transferred over the network to the host server, and then written to its local disk before being booted by the hypervisor. As VMIs are in the order of gigabytes, launching a VM can take a long time. Even worse, the impact of disk and network becomes more obvious in the case of multi-deployment (i.e., when a large number of VMs are requested at the same time) and therefore increases further the provisioning time [202, 240, 314].

5.2 Container Image Management

Containers are attracting more attention in clouds as they provide more lightweight process isolation compared to conventional virtual machines [266]. Like VMIs, container images are growing in numbers [264]. Moreover, they present high similarity as users build their container images on top of other images.

The container image should be available locally to start the corresponding container. Though smaller in size compared to VMIs, pulling a container image from the image repository (i.e., storage nodes) to the compute node accounts for 76% to 92% of the total provisioning time of a container [109, 75]. Therefore, reducing the transfer time of container images – from the repository to the host machine – plays a vital role in improving containers provisioning time.

Without loss of generality, in this work, we focus on Docker containers. Docker is a mature container technology and has a well-defined image format. Besides, it is widely adopted in both industry [102] and academia [162].

Docker

Docker is a container management framework which provides a set of tools to simplify container creation, execution, and sharing. Docker is composed of three components:

- The **Docker daemon** is responsible for managing containers on the host machine. It runs and stops the containers, create images and communicate with the registry (i.e., the image repository) to pull and push images. It accepts commands from the Docker client.
- The **Docker client** issues commands to the daemon using RESTful API.
- The **Docker registry** is the image repository for Docker. The registry stores, tags, and serves docker images.

Docker Images and Layers

Docker container images are structured as layers. Each image consists of an ordered set of layers and a *manifest* that describes the layers of that image. Each layer is a collection of read-only files that are stored as a gzip-compressed tar file. Layers could range in size from few kilobytes to several hundreds of megabytes [17]. The layered image structure supports layer-sharing across different containers. Docker identifies layers by hashing over their contents.

Docker registry

The *registry* is the component responsible for container image management in the Docker ecosystem. It stores all the layers (from all the container images). Since layers are read-only and could be shared by multiple container images, Docker registry uses layer-level deduplication to reduce the size of the image dataset. Docker registry could be a remote online registry (e.g., Docker Hub [73]), a central private registry, or a local registry on the same machine as the Docker daemon. Docker Hub [73], one of the largest public Docker repositories, contains more than 2.5 million images [74].

Provisioning workflow

Once a container is scheduled to run on a compute node, the corresponding image is requested to be transferred to this node, if not available locally. Docker daemon first fetches the missing layers from a registry. The received layers are then extracted locally. To boot up the container, the read-only layers of the image are stacked on top of each other and joined to expose a single mount-point via a union file system such as AUFS [22] or OverlayFS [212]. An extra writable layer is added on top of them to store all the changes to the container file system using the copy-on-write (COW) mechanism. This layer will be lost when exiting the container unless it is explicitly saved.

5.3 Towards Efficient Service Image Management in Distributed Clouds

There has been much work on optimizing service provisioning in cloud. Hereafter, we review those related to efficient virtual machine provisioning and container provisioning. After that, we highlight some state-of-the-art data transfer over the WAN.

5.3.1 Efficient virtual machine management

Previous literature has shown high similarities across VMIs, up to 80% [142, 138, 219, 35, 241, 254]. These similarities are attributed to the common base images and image build tools that are used to construct the services' images. As a result, deduplication techniques are widely adopted not only to reduce the image storage footprint but also to improve the service provisioning time by leveraging common chunks from other images that are already available on the host machine.

In this section, we review some literature about VMI management. First, we present works that explore deduplication to reduce the size of the image repository. Then, we discuss research works on improving the transfer time of VMIs. Finally, we describe the related work on optimizing the boot time of the VMs.

Optimizing storage overhead

Content Addressable Storage (CAS) systems have been investigated as a storage backend for VMIs [196, 175]. CAS applies deduplication to store the data. As a result, a large saving in storage cost can be achieved (up to 70%). However, these systems cannot ensure fast access to the VMIs – which is important in cloud – as they are originally designed for archival data where fast data insertion and retrieval are not critical.

The Mirage library implements the Mirage Image Format (MIF) that exposes the semantic of the images [241]. Mirage employs file-level deduplication to the VMI library achieving more than 10x reduction in storage cost. Moreover, exposing the file system structure of images allows several maintenance operations at a lower cost. For example, applying a security patch can be performed by modifying the corresponding files without the need to boot a VM and take another snapshot of its disk. Importantly, these modifications are applied to all the concerned images at once. Similar work applies the same technique of file-level deduplication [254]. In addition to the features provided by Mirage, they support Copy-on-Write (COW) file systems and allow direct mount of stored images.

LiveDFS [199] is a file system that enables deduplicated storage for VMIs. LiveDFS has several distinct features including spatial locality, prefetching of metadata, and journaling. LiveDFS is POSIX-compliant and is implemented as a Linux kernel-space file system. LiveDFS can save at least 40% of the storage cost while achieving reasonable performance in importing and retrieving VM images.

Optimizing VMI transfer time

One of the first attempts to improve the VMI transfer time is the use of a modified version of BitTorrent to distribute VMIs at CERN [289]. Following this approach, images should be completely copied to the destination hosts before running the VMs. However, this approach is effective only if a large number of instances are started from the same image as common contents across different VMIs cannot be leveraged.

Kochut et al. [157] decompose VMIs into clusters of data. Each image is composed of one or more clusters with some clusters shared between more than one image. When provisioning a VM to a compute node, only the missing clusters are transferred, reducing the network traffic and the provisioning time.

Karve et al. [149] extend the previous work and propose to leverage data deduplication for VMIs transfer in geo-distributed data centers. The master site maintains the global view of the system and the distribution of the chunks; thus, it is responsible for creating the transfer plan of chunks to the destination site. However, this work is limited to small-scale systems and may suffer a noticeable performance degradation due to the centralized management overhead when dealing with a large number of VMIs. Besides, it does not consider network heterogeneity between data centers [119].

In Virtual Distribution Network (VDN) [219], the authors opt for more fine-grained control by employing fixed-chunk deduplication. Therefore, when provisioning a VM, just the missing chunks are transferred to the destination host. Moreover, VDN takes the data center network topology into account and always tries to retrieve the missing chunks from the nearest node.

Xu et al. [300] propose zone-based storage to improve chunks distribution (i.e., image transfer). In-memory caches are deployed for each group of nodes (i.e., a zone) to cache the hot chunks instead of retrieving them from the central repository. The zone-based caching approach avoids the bottleneck caused by global caching and reduces the storage overhead caused by pure local caches.

Further works propose that VM schedulers leverage the chunks distribution to select the host that has the highest percentage of chunks for the corresponding VMI. While this can greatly reduce the amount of transferred data, and therefore, reducing the transfer time of the image, it may adversely impact the performance of the running services and applications. These approaches have been discussed in [219, 35, 300].

Optimizing VM boot time

Fast Virtual Disk (FVD) is a new virtual machine image format [269]. FVD introduces, among others, copy-on-read and data prefetching. Hence, when booting a VM with the image stored remotely, the requested chunks can be cached locally instead of being requested again over the network. This new image format provides a solution in both cloud and non-cloud environments, however, it cannot leverage the similarities in-between the images of the VMs running on the same host.

Lean Virtual Disk [34] is another virtual machine disk format that eliminates the redundancy between multiple VMIs on the same host by employing deduplication. This can reduce disk space and disk I/O leading to faster boot time as similar chunks could be leveraged locally from other images. However, this approach breaks the modularity of images and introduces high management overhead.

VMTorrent [240] employs peer-to-peer chunks distribution to transfer the image of a VM. However, it decouples the chunks distribution from data stream presentation mechanism, allowing hypervisors to access this data stream as if it was stored locally thus the image could be booted while the chunks are arriving. VMThunder [314] employs a similar method but it provides more features such as Cache-on-Read (CoR) to further reduce the network traffic.

Razavi et al. [236, 235] followed another approach by just caching the chunks required for the boot process. The size of this boot working set is in the order of hundreds of megabytes. The caches can be deployed in the compute node to reduce the network traffic or in the storage nodes to reduce disk access. YOLO [200] improves over the previous works by introducing a non-intrusive solution for prefetching and caching the boot data and can work for virtual machines as well as containers.

ACStor [297] takes the current network traffic into account when retrieving the requested chunks while booting the VM. In particular, the requested chunks are retrieved from nodes with light traffic-load. This is especially important in a large-scale environment where the overall network traffic is unbalanced.

5.3.2 Scalable container image management

Docker provides built-in deduplication for its images as common layers are stored once. Even though layer deduplication reduces the storage overhead, however, pulling the images to the compute host consumes a considerable time of the provisioning process. Therefore, many efforts have focused on optimizing the Docker registry performance as well as container image placement and retrieval to improve the provisioning time.

The Docker registry can be seen as a data-intensive application; it is estimated [17] that at least hundreds of terabytes of data, and at least 2.5 million images [74], are stored in Docker Hub [73]. With the increasing number of concurrent clients, the registry could become a bottleneck in the provisioning process.

Docker image placement and container provisioning have been studied in [197]. The authors propose a collaborative Docker registry where the private registries on the cluster (compute) nodes can collaborate to store and retrieve Docker images instead of relying on a central or remote registry. Moreover, the authors opt for layer placement rather than complete image placement to avoid redundancy in common layers. They employ simple heuristics to distribute the layers to the local registries. Bipartite graphs have been used to balance the retrieval of missing layers from multiple nodes. They ensure image availability by 3-way replicating each layer.

Anwar et al. [17] analyzed IBM cloud registry traces in eight data centers and characterized Docker registries workloads. Based on their findings (e.g., the registry access patterns are heavily skewed, 65% of the layers are smaller than 1 MB, strong correlation between adding a new image and subsequent retrievals of that image, etc.), they designed caching and prefetching techniques to speed up image pulls.

BitTorrent protocol has been used to distribute containers' layers in a single data center setup [148]. Several instances of the peer-to-peer registry are deployed in the system, moreover, the cluster's nodes contribute to the distribution of images. A great reduction in provisioning time could be achieved when the same image is requested by a large number of machines simultaneously. However, for single image provisioning, no improvement can be achieved. Moreover, BitTorrent protocol cannot guarantee a predictable performance.

Docker-PI [5] overcomes the limitations of Docker engine when pulling images (from a Docker registry) to improve the image retrieval time. In essence, when retrieving the layers of an image, Docker performs the retrieval phases sequentially (i.e., pulling, decompression, verification, and extraction). Docker-PI allows the overlapping of these phases while ensuring the consistency of the final image when creating it. This can efficiently utilize the resources of the destination machine which result in faster retrieval time.

Some works opt for centralized management where all containers are served remotely from a single distributed storage system. Slacker [109] uses chunk-level deduplication to speed up container provisioning time by transferring the required chunks on demands, similar to previous work on VMIs. This is motivated by the fact that just a small fraction of the image is needed to boot the container (around 6%). However, Slacker requires flattening a Docker image into a single layer, breaking the modularity presented in the original design of Docker images.

In Cider [75], rather than storing images in the Docker registry, all the layers are kept in a shared storage system (Ceph), while the registry serves as a metadata server. As Slacker [109], Cider eliminates the initial image transfer (thus, achieves a 85% faster boot time), however, it still incurs a network overhead while containers are accessing their data over the network.

Similar to Cider, Wharf [317] allows Docker daemons to share a distributed storage instead of relying on the local storage of the host nodes. This approach requires modifications to the Docker daemons as originally Docker daemons are designed to run alone (i.e., one daemon per host) and share nothing with other daemons. Even though it speeds up image retrievals by up to 12x, Wharf imposes noticeable performance overhead on the running services and applications as all the requests to the image are now made over the network.

5.3.3 Optimizing large-scale data transfers: Beyond service images

Various solutions and tools have been proposed to optimize large data transfer over the Internet (i.e., WAN). We present here some representative work while arguing why they are not adequate to optimize service image transfer.

General purpose content delivery systems such as BitTorrent [56] are used to distribute large files, especially multimedia files, over the Internet. However, BitTorrent does not take into account network heterogeneity when retrieving data as peers are selected randomly. Moreover, BitTorrent does not guarantee predictable performance. IPFS [38] (described in Section 4.3) is a peer-to-peer distributed file system that has built-in deduplication across all its content. However, IPFS addresses network heterogeneity by pulling chunks from multiple peers to avoid the impact of weak links. This may lead to better performance but at the cost of high network overhead.

Sharing large datasets (e.g., scientific datasets) between multiple Grid sites [89] has increased the demand for efficient data transfer over Internet. GridFTP [9, 10] is an extension of File Transfer Protocol (FTP) [87] that is optimized for secure, reliable, and high-performance data movement. To achieve high end-to-end data transfer, GridFTP employs techniques such as multiple TCP streams per transfer, striped transfers from a set of hosts to another set of hosts, partial file transfers, and automatic negotiation of TCP buffer/window sizes. As extensions to GridFTP for higher data transfer, the authors in [181] develop tuning mechanisms to select at runtime the number of concurrent threads to be used for transfers, while multi-hop path splitting and multi-pathing have been discussed in [152]. GridFTP, alongside other file transfer protocols and applications such as FTP [87] and FDT [84] as well as file-based transfer commands, such as scp [178] and rsync [248] only provide point-to-point transfer, thus, do not benefit from data replicas in other sites.

Minimizing the transfer time of a set of files distributed across multiple machines to another set of machines in a wide-area environment is studied in [153]. Files are split into chunks so that the chunks of a file could be retrieved from different replicas. However, the chunk size is large (the smallest considered chunks is one third of the file size), and decreasing the size of the chunk increases substantially the complexity of the proposed algorithm. Moreover, this approach does not consider the similarities between files leading to high network traffic when transferring large similar files.

With the proliferation of geo-distribute clouds, the demand for inter-data center data transfers in large volumes (e.g., migration of Big Data) has significantly grown. Tudoran et al. [276] propose an adaptive data management system that automatically builds and adapts performance models for the cloud infrastructure to offer predictable data handling performance for the transfer cost and time. Wu et al. [298] leverage Software Defined Networking (SDN) techniques to design efficient bulk data transfer service at different urgency levels (different finishing deadlines), in order to fully utilize the available inter-data center bandwidth. Lin et al. [177] optimize files transfer under delay constraints by leveraging multi-path transfer. However, these approaches either increase the monetary cost of the transfer to increase its throughput or perform on the low network level and this requires a knowledge of the underlying network structure which is only available for the cloud provider, and therefore, these solutions complement ours.

Finally, cloud providers do not offer dedicated services for data transfer, however, object storage solutions (e.g., Amazon S3 [28], Azure Blob Storage [29], OpenStack Swift [210]) can be used to perform the transfer by reading from one site and writing to another one. However, these systems usually provide data transfer by the means of the underlying point-to-point protocol (e.g., rsync in case of OpenStack Swift [205]).

5.4 Discussion: Enabling Efficient Service Provisioning in Distributed Clouds

Service image management is essential to provide fast service provisioning in clouds. Provisioning a service requires considerable network and disk access overhead to ensure the availability of the large service image on the destination compute node. Hence, a large body of work has focused on service image management to optimize service provisioning. However, several new challenges arise when provisioning services in distributed clouds:

- Low WAN links bandwidth: the low bandwidth of network links (e.g., as low as 35 Mb/s [119]) prolongs the provisioning of services as the corresponding image takes a longer time to be transferred. Transferring as little data as possible might reduce the impact of low bandwidth links.

- Heterogeneous WAN links bandwidth: the heterogeneity between links bandwidth could be 12x [119]. This impacts the provisioning time if the available bandwidth is not fully utilized. Adapting the amount of transferred data to the link bandwidth might reduce the impact of this heterogeneity.

- Monetary cost of data transfer: unlike local area network (LAN), transferring data over the WAN in clouds has a monetary cost [12]. Sending less data reduces the monetary cost of image transfer.

- Limited available storage of Edge-servers: as Edge-servers have limited storage capacity, storing all the images locally is not possible. This requires pulling images over the network. However, exploiting a small fraction of their capacities to store some data might improve the provisioning time.

Unfortunately, most state-of-the-art work for service provisioning attack the problem in a *single* data center setup. The proposed methods are not adequate for distributed clouds as they do not consider the previously mentioned challenges. On the other hand, works targeting data transfer over WAN are usually general-purpose methods, in most parts they are point-to-point transfer, or allow parallel transfer but do not take the specific features of service images into account (i.e., high similarity) which result in redundant data movement and extra monetary cost.

In response, in the following two chapters, we show how to improve service provisioning in distributed clouds by means of efficient network-aware retrieval and placement of services' images (i.e., VMIs and container images).

Chapter 6

Network-aware Service Image Retrieval in Geo-distributed Clouds

Contents

Nowadays, major cloud providers deploy their services on geo-distributed infrastructures. This worldwide distribution provides low latency for the end users of the services. For example, Amazon AWS currently has 18 geographically distributed service regions [26], and Windows Azure operates in 54 geographical locations [296]. This geo-distribution may result in severe performance degradation for service provisioning (i.e., long and unpredictable provisioning times). This is because VMIs may need to be transferred over WAN links which are featured with low bandwidths and usually exhibit high variability. While the provisioning time is dictated by the distribution of VMIs and network bandwidth in-between sites, unfortunately, existing provisioning methods do not consider these aspects and therefore are not adequate for geo-distributed clouds.

To mitigate the impact of WAN links, in this chapter, we present the design and implementation of Nitro, a novel VMI management system for geo-distributed clouds. Nitro focuses on minimizing the transfer time of VMIs over a heterogeneous WAN which is the main performance bottleneck when provisioning services in geo-distributed clouds. To achieve this goal, Nitro incorporates two complementary features. First, it makes use of deduplication to reduce the amount of data which is transferred due to the high similarities within an image and in-between images. Second, Nitro is equipped with a network-aware data transfer strategy to effectively exploit links with high bandwidth when acquiring data and thus expedites the provisioning time. The network-aware strategy embraces an algorithm that produces an optimal chunk scheduling in polynomial time based on graph flow algorithm. To reduce the overhead and improve the scalability of our designed algorithm, we propose a grouping optimization to reduce the number of chunks in the graph. Experimental evaluation of Nitro – on top of Grid'5000 [105] – demonstrates its utility while outperforming state-of-the-art VMI storage systems.

This chapter is organized as follows. In Section 6.1, we define the challenges of geo-distributed service image management. In Section 6.2, we present the design principles of Nitro. Then, in Section 6.3, we explain the network-aware chunk scheduling algorithm. While in Section 6.4, we present Nitro and its workflow. Experiment methodology and results are discussed in Sections 6.5 and 6.6, respectively. The applicability and limitations of Nitro are discussed in Section 6.7. Finally, Section 6.8 concludes this chapter.

6.1 Challenges of VM Provisioning in Geo-distributed Clouds

As fast service provisioning is essential in clouds, VMI management has become an important issue. Prior literature has mainly focused on leveraging *deduplication techniques* to eliminate redundant blocks in-between VMIs and therefore reduce the storage space as well as the provisioning time within a *single* data center [157, 219, 300]. However, few works have explored VMI management in geo-distributed clouds [149]. The geographical spread of data centers, the continuous increase of VMIs number, in addition to the limited bandwidth and heterogeneity of the WAN connections, have elevated VMI management to a key issue in geo-distributed clouds. Nevertheless, several major challenges arise when dealing with geo-distributed VMIs:

- **Challenge 1**: The size of VMIs is critical for fast service provisioning, as the size of a single VMI can reach dozens of gigabytes [35, 258, 286]. Previous efforts try to reduce VMI size by exploiting similarities within and in-between VMIs, through deduplication, to reduce the storage cost. However, they do not evaluate the impact of deduplication on the service provisioning time, when the VMI is pulled from different geo-distributed sites over high latency and low bandwidth WAN.

- **Challenge 2**: On the one hand, it is not practical to replicate VMIs on *all* sites. The cost, in terms of data transfer across data centers, may become prohibitive as the size and number of VMIs increases. For example, maintaining one single VMI introduces high transfer cost when it is updated frequently; security patches alone may result in almost 150 updates per week [320]. On the other hand, replicating VMIs on (few) geo-distributed sites to ensure high availability and meet users' needs poses a challenging issue when provisioning VMs, especially as the large size VMIs must be pulled – over WAN – from multiple geo-distributed locations.

- **Challenge 3**: Previous solutions which adopted deduplication techniques assume a constant cost when retrieving the VMI chunks and thus use simple and random retrieval methods. This results in long provisioning time when applied directly in geo-distributed clouds, due to the link heterogeneity. For example, the bandwidth in-between 11 sites in Amazon EC2 varies by up to 12x [119]. An optimal retrieving plan is therefore imperative to improve the provisioning time. However, finding the optimal solution comes with a high computation overhead due to the large number of chunks. For example, as shown in Section 6.6, it takes almost 267s to find the optimal plan to pull 10,000 *chunks* from 4 *sites*.

Our work addresses the aforementioned challenges of VM provisioning in geo-distributed clouds and takes a step forward toward scalable and efficient VMI management through the introduction of Nitro. Nitro, by exploiting deduplication and being aware of the network characteristics, aims to improve the provisioning time in geo-distributed clouds.

6.2 Nitro: Design Principles

Nitro is a novel VMI management system responsible for storing and retrieving VMIs to and from different sites in geo-distributed clouds. This section presents the design principles of Nitro, while the following two sections focus on the network-aware VMI pulling and the implementation details of Nitro. Nitro is designed with the following goals in mind:

- **Reduce network overhead:** This is critical in geo-distributed clouds when transferring the large-size VMIs. Previous works [142, 138] show that exploiting similarities within and in-between VMIs may result in a reduction in storage space by up to 80%. Therefore, Nitro aims at reducing the amount of data transferred over WAN by leveraging deduplication. This does not only reduce the size of acquired VMIs but also effectively increases locally available chunks (on destination site).

- **Network-aware data retrieval:** The bandwidth and latency in-between data centers vary significantly [119]. For that reason, Nitro employs a network-aware chunk scheduling algorithm to find the optimal plan when acquiring chunks from different sites. Thus, we can effectively leverage links with high bandwidth and reduce the number of chunks retrieved over weak links.

- **Minimize provisioning time:** Through reducing the amount of transferred data, exploiting chunks locality, and carefully pulling chunks from different sites, Nitro can minimize the transfer time and thus improve the provisioning time.
- **Ensure minimal runtime overhead:** Finding the optimal plan to pull a VMI comes with a high computation overhead due to the large number of replicated chunks. Therefore, we propose a grouping optimization to reduce the problem size (i.e., reduce the number of chunks) when finding the optimal solution. This optimization allows our scheduling algorithm to run in sub-second while preserving its optimality.
- **Storage backend independent:** Since Nitro is implemented as a separate layer on the top of the cloud storage system, it does not impose any modifications to the cloud system code. Consequently, Nitro can use any storage systems (i.e., key/value store) to store the chunks. This modularity is important as new emerging key/value storage systems can be used to further improve the provisioning time through optimizing data transfer within a data center.

6.3 Network-aware Chunk Scheduling Algorithm

This section presents the network-aware chunk scheduling algorithm used in Nitro. This algorithm produces an optimal chunk scheduling that minimizes the transfer time of chunks over heterogeneous networks. It is based on min-cost max-flow graph algorithm and it has polynomial time complexity.

6.3.1 Problem definition

Consider the scenario of I VMIs and a geo-distributed cloud composed of N sites. Each image is divided into C *equal* chunks and the chunks can be spread to any of the N sites. We suppose that each pair of sites is connected with a dedicated link [283]. When a VMI is requested from a site, we first look for the chunks of the VMI in the local site. If there are not enough chunks to reconstruct the VMI locally, we need to decide which sites to pull the missing chunks from (i.e., the chunk scheduling problem). Our goal is to minimize the time needed to pull all missing chunks from remote sites.

We formally model the chunk scheduling problem using a bipartite graph $G = (V,E)$. The vertex set V includes two types of vertices, namely the set of requested image chunks and the set of all sites. E is the set of directional edges from the chunk nodes to site nodes. An edge from a chunk c to a site s represents that there is a copy of chunk c on site s.

Under the bipartite graph model, the chunk scheduling problem can be described as finding an assignment from chunk nodes to site nodes that reduces the transfer time. The number of chunks assigned to a site represents the relative time needed to complete the transfer from that site (assuming equal chunk size and homogeneous bandwidth). Therefore, as the transfers can be performed in parallel from different sites, minimizing the total transfer time can be achieved by minimizing the maximum transfer time from each site. Assignment problems in bipartite graphs are often solved using network min-cost max-flow algorithm. Flow algorithms have been used in the literature to optimize job placement in data centers [134, 99] and data transfer between data centers [153, 298]. However, classical matching algorithms try to find the maximum match regardless of the mapping. In the following, we introduce the basics of the algorithm and how we have adapted it to solve our problem.

6.3.2 Maximum-flow algorithm

The maximum flow algorithm tries to find the maximum flow that can go through the network from the source node to the sink node, respecting the following two conditions:

- **(1)**: For each edge, the flow going through the edge should not exceed its capacity.

  $$F(e) \leqslant C(e), \forall e \in E$$

  where F and C are the flow and capacity of edge e, respectively.

- **(2)**: For each vertex, the incoming flow should be equal to the outgoing flow.

  $$\sum_u F(e_{uv}) = \sum_k F(e_{vk}), \forall v \in V$$

  where u is an incoming neighbor of v and k is an outgoing neighbor of v.

In our problem, the capacity of edges between the source node and the chunk nodes is 1. The capacity of the edge connecting a site node with the sink node represents the maximum chunks that can be pulled from the site. Initially, it can be equal to the total number of chunks. However, these capacities will change during the execution of the scheduling algorithm. Figure 6.1 shows the graph representation of a simple example where 5 chunks spread over 3 sites are required.

Directly applying the maximum flow algorithm to our problem using the above graph representation might not generate the edge assignment that we are looking for. This is because the algorithm always tries to maximize the flow regardless to which sites these chunks are assigned to. However, in our problem, it is important to study how the maximum flow is distributed among edges. We design our chunk scheduling algorithm based on this observation.

6.3.3 Chunk scheduling algorithm

As our goal is to find the chunk assignment solution which minimizes the time needed to acquire all requested chunks, we design a max-flow based algorithm to find the maximum flow solution that also provides a balanced load distribution between sites as much as possible. For simplicity, we first assume the network bandwidths between all sites are homogeneous and discuss how to extend our algorithm to heterogeneous network environment later.

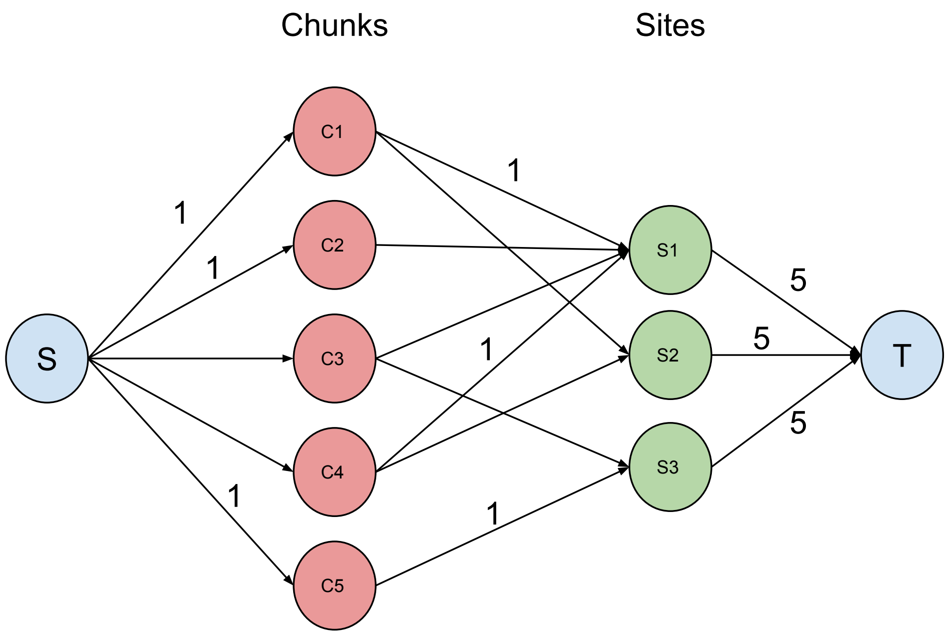


Figure 6.1: Graph example: 5 chunks which are spread over 3 sites are required. Initial edges capacities are also shown.

As described in the previous subsection, the capacity of an edge represents the maximum amount of flow that can go through it. In our case, the amount of flow going through edges connecting site nodes with the sink node represents the number of chunks assigned to each site, therefore, the data size. As the bandwidth is homogeneous between sites, the edge flow represents the transfer time. Hence, minimizing the flow is equivalent to minimizing the transfer time from all sites. Consequently, the idea behind chunk load balancing is how to control the *capacity* of edges connecting the sites with the sink node. Intuitively, a capacity equal to or greater than the number of chunks can always guarantee a max flow solution, whereas a capacity less than $\frac{|C|}{|S|}$ cannot lead to a feasible solution because some chunks remain unassigned. Thus, our goal is simply to find the minimum capacity that can provide a max flow solution in the range $[\frac{|C|}{|S|}, |C|]$. By searching for the minimum threshold – which represents the maximum number of chunks that can be pulled from each site – that can produce a feasible flow solution, we find the solution with the most even (i.e., load-balanced) chunk distribution.

So far, we have provided a solution for the case of homogeneous WAN bandwidth between sites. However, in reality, the WAN bandwidth between different sites is heterogeneous and assigning an equal number of chunks to each site, if possible, does not minimize the total time of chunk acquisition, as more time is spent to pull the same amount of data from different sites. To address the heterogeneity issue, we scale the capacities of the edges connecting site nodes with the sink node according to the available bandwidth (between each site and the destination site) before running the flow algorithm and for every iteration of the algorithm. The intuition behind capacity scaling is that the site with higher network bandwidth can pull more chunks within a unit of time than sites with lower bandwidth.

For example, consider that 4 chunks are required and all of them are available on two

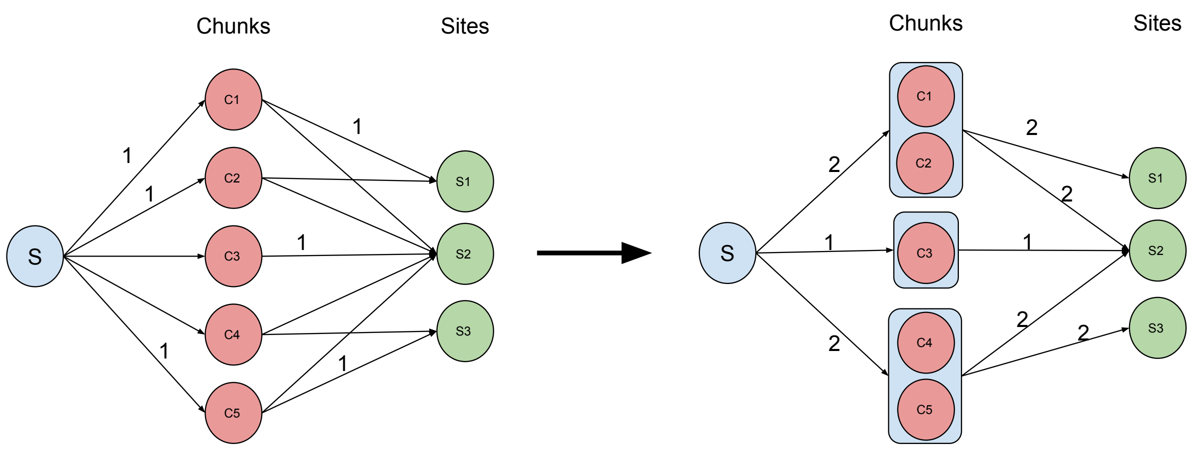


Figure 6.2: An example of the grouping optimization.

sites. One of the sites has a network bandwidth of 50 Mb/s to the destination site and the other site has a bandwidth of 150 Mb/s. Without considering network heterogeneity, the scheduling algorithm retrieves two chunks from each site. However, the optimal solution would be to retrieve one chunk from the first site and three chunks from the second one.

6.3.4 Grouping optimization

As a VMI is usually composed of dozens of thousands of chunks on average, the bipartite graph can become large. To reduce the overhead and improve the scalability of our designed algorithm, we propose a grouping optimization to reduce the number of chunk nodes in the graph.

The number of chunk nodes can be reduced by grouping the chunks that can be found in the same set of sites into one chunk node, denoted as *Mega Chunk* (MC) node. The number of mega chunk nodes has an upper bound equal to $2^{|sites|} - 1$, which is the cardinality of the set of sites subsets. This upper bound is reached only if there is at least one chunk that is connected to each subset of sites. Also, the number of mega chunk nodes does not exceed the number of chunk nodes, which means that this optimization is at least as fast as before applying it. Note that, creating the MC nodes can be performed in linear time complexity.

After the grouping, the capacity of edges from the source node to the mega chunk nodes has to be modified according to the sizes of these mega chunk nodes. For example, as shown in Figure 6.2, the capacity of the edge from the source node to the first mega chunk node is changed to two, as there are two chunks in the mega chunk. A mega chunk can be matched to multiple sites at the same time, which is different from the simple chunk case where each chunk is matched to one and only one site. For example, given a mega chunk node of five chunks, if this mega chunk is matched to two sites, with a flow of two to the first site and three to the second one, we randomly select two chunks to pull from the first site and three from the second site, as we do not differentiate between the chunks.

6.3.5 Algorithm overview and analysis

A summarized pseudo-code of our chunk scheduling algorithm is presented in Algorithm 1. It receives as parameters the requested chunks, the sites to pull from, the current mapping of chunks and the bandwidth scaling (i.e., the relative bandwidths between the destination site and other sites). The algorithm returns an assignment of the requested chunks to the sites. The functions used inside the algorithm are described below:

- **create_mega_chunks** implements the grouping optimization and groups the input chunks which can be found in the same subset of sites into mega chunks.
- **build_bipartite_graph** builds the graph from the chunks and sites and sets the capacity of the edges as discussed before.
- **add_capacity_to_sink_edges** sets the capacity of the edges linking the site nodes to the sink node.
- **max_flow** is the *preflow-push algorithm* [6] for computing the maximum flow. It takes a graph instance as input and updates the flow of its edges.
- **remove_set_k** is a helper function that removes k elements randomly from the input set and returns them.

The algorithm starts by grouping the chunks into mega chunks as we have discussed previously. Then, the graph structure is built from the set of mega chunks and the set of sites. The capacities of the edges are set accordingly, however, the capacities of the edges linking the site nodes to the sink node are left for a later step. The main loop performs a binary search on the capacities range to find the optimal one; in each iteration, we first compute the scaled capacities and then we set the capacities of the sink edges. Later on, we run the flow algorithm to compute the total flow that goes through the network. If the total flow is equal to the number of chunks, that means we can reduce the search range to search for another solution that provides more load balancing. The other case (total flow is less than chunk size) means that not all the chunks are matched and the current solution is not valid, so we increase the lower limit of the search. When the loop is finished, we run the flow algorithm with the optimal capacity found. At the end, we create the set of chunks that should be requested from each site with respect to the output of the flow algorithm.

The complexity of the algorithm is mainly related to the maximum flow algorithm and the binary search. The complexity of the maximum flow algorithm (i.e., preflow-push algorithm) is $O(|V'|^2\sqrt{|E'|})$. Where V' and E' are the sets of mega chunks and corresponding edge respectively. The range length of the binary search is $|C|$, which means that the algorithm iterates no more than $log_2(|C|)$ iterations. Putting together the complexities of the two algorithms we find that the complexity of the scheduling algorithm is $O(log_2(|C|) * |V'|^2\sqrt{|E'|})$.

6.4 Implementation

Nitro consists of roughly 1500 lines of Python code. The current implementation uses Redis [238] as a storage backend to store VMIs, i.e., the chunks of the images. However,

Algorithm 1: Network-aware chunk scheduling

```
Input  : chunks, sites, chunks_mapping, bw_scaling
Output: chunks_request
1  $mega\_chunks \leftarrow$ **create_mega_chunks** $(chunks, chunks\_mapping)$ ;
2  $G \leftarrow$ **build_bipartite_graph** $(mega\_chunks, sites)$ ;
3  $low\_cap, high\_cap \leftarrow 1, nb\_chunks$ ;
4  while $low\_cap < high\_cap$ do
5      $cap \leftarrow (low\_cap + high\_cap)/2$ ;
6      $scaled\_cap \leftarrow cap * bw\_scaling$ ;
7      **add_capacity_to_sink_edges** $(G, scaled\_cap)$ ;
8      **max_flow** $(G)$ ;
9      $total\_flow \leftarrow sum_{site=1}^{nb\_sites} F(G[site][sink])$ ;
10     if $total\_flow = nb\_chunks$ then
11         $high\_cap \leftarrow cap$ ;
12     else
13         $low\_cap \leftarrow cap + 1$ ;
14     end
15 end
16 $optimal\_capacity \leftarrow low\_cap$ ;
17 $scaled\_optimal\_capacity \leftarrow cap * bw\_scaling$ ;
18 **add_capacity_to_sink_edges** $(G, scaled\_optimal\_capacity)$ ;
19 **max_flow** $(G)$ ;
20 $chunks\_request \leftarrow \{\}$ ;
21 foreach $site \in sites$ do
22     $chunks\_request[site] \leftarrow \cup$ **remove_set_k**
        $(mc, F(G[mc][site])) : \ for\ mc\ \in\ mega\_chunks$
23 end
24 return $chunks\_request$ ;
```

any key/value storage system can be used as a storage backend to Nitro. The source code is publicly available at `https://gitlab.inria.fr/jdarrous/nitro`.

As shown in Figure 6.3, Nitro consists of two components: a *proxy server* and a *storage engine*. The proxy server is responsible for handling all the requests to VMIs including add, retrieve, etc. The storage backend is where the actual chunks are stored and is implemented as a key/value store. Hereafter, we describe the system workflow.

6.4.1 System workflow

Bootstrapping

To deploy Nitro in a distributed environment, a daemon has to run in each data center. On bootstrapping, these daemons are provided with the endpoint addresses of other daemons.

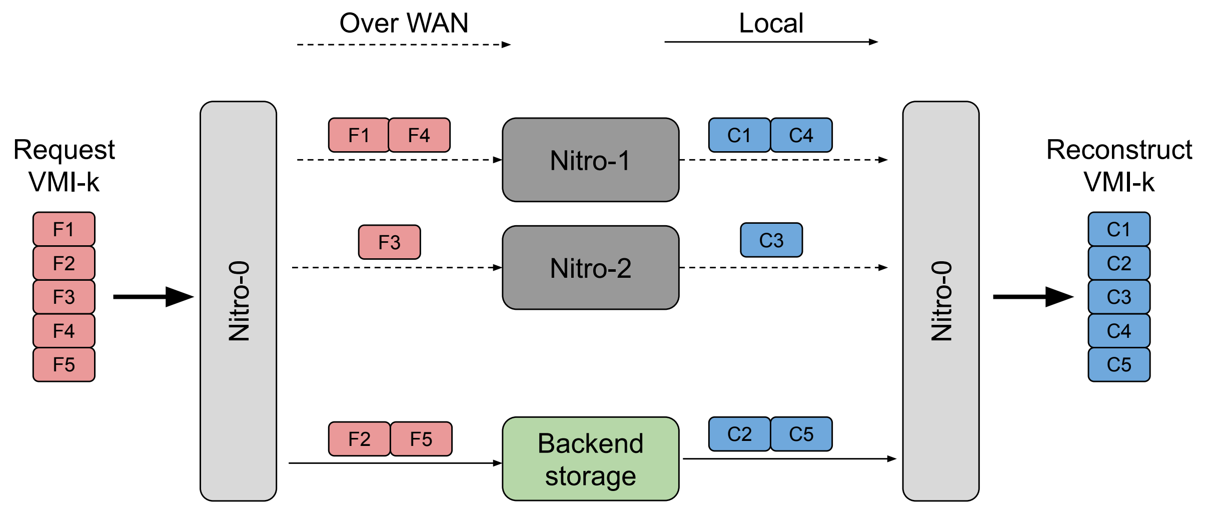


Figure 6.3: VM provisioning workflow. A VMI with 5 chunks is requested; 2 chunks are available locally and 3 chunks are requested from other data centers. F for $fingerprint$ and C for *chunk*.

Adding a new image

New VMIs can be added to any site running the Nitro system. The image is split into fixed chunk sizes of 256 KB. A list of references to these chunks is also created. This list is used later to reconstruct the original VMI. These chunks are stored into the backend key/value store along with the fingerprint list. The metadata is then propagated synchronously to all other daemons. Then, the universally unique identifier (UUID) of the image is returned to the client indicating the successful termination of the process.

Retrieving an image

The generated UUID, when adding the VMI, is used later to retrieve it from any other location. Nitro reads the fingerprint list related to that UUID and checks the chunks which are available locally. The list of missing chunks, in addition to the currently available bandwidths between the current site and the other sites, are then transferred to the scheduling module that computes the optimal transfer plan to retrieve the chunks from other sites. After receiving all the previously missing chunks, these chunks are stored (in Redis) and the original VMI is reconstructed and returned to the user.

6.4.2 On compressing data chunk in Nitro

To reduce the size of data transferred over WAN, we further enable data compression on the chunks in Nitro. Chunk compression may result in different chunk sizes, and therefore, may impact the optimality of the scheduling algorithm when retrieving chunks. We discuss the trade-off between data compression and optimal chunks mapping in the evaluation section (Section 6.6).

6.5 Experimental Methodology

6.5.1 Dataset

The dataset consists of 24 VMIs (in raw format) with a size ranging from 2.5 GB to 6.5 GB for each. As our scheduling algorithm works on the chunk level and is not aware of the total number of images in the system, a dataset of 24 images is sufficient to evaluate the performance of Nitro.

The dataset is built by provisioning the following versions of eight base images (3 Debian: Wheezy, Jessie, and Stretch; 3 Fedora: Fedora-23, Fedora-24, and Fedora-25; and 2 Ubuntu: Trusty and Xenial) with three software (Apache Cassandra, Apache Hadoop, and LAMP stack) using Vagrant [279].

Table 6.1 presents the sizes of our dataset with different storage formats. The *deduplication* and *deduplication with compression* rows represent the cases where each image is deduplicated separately, *without* and *with* further chunk compression, respectively. Whereas the corresponding rows with the *(dataset)* tag represent the case where all the images of the dataset are deduplicated together, i.e., similar chunks of different VMIs are discarded. The *compression* row gives the size of the dataset compressed in `gzip` format. From the presented results, we can notice that applying compression with deduplication (where each image is deduplicated separately) is slightly better than compression alone in terms of data size reduction. However, deduplication with compression can by far reduce the size of the data compared to compression alone, as the complete dataset is taken into account. Moreover, the efficiency of deduplication – in contrast to compression – grows when increasing the size of the dataset.

Table 6.1: Dataset sizes under different storage formats.

Image storage format	**Total size (GB)**
raw	104.89
deduplication	50.65
deduplication (dataset)	25.11
compression	19.99
deduplication with compression	17.99
deduplication with compression (dataset)	8.66

6.5.2 Testbed

We perform our experiments on top of Grid'5000 [105], a French platform for experimenting distributed systems. We used for our experiments the Nova cluster at the site of Lyon. Each machine in the cluster is equipped with two Intel Xeon E5-2620 v4 CPUs, 8 cores/CPU, 64 GB RAM and 600 GB HDD. The machines are interconnected with 10 Gbps Ethernet network. The nodes run Debian Linux 8.0.0 (jessie). The latency and bandwidth between machines are emulated using `tcconfig` [270], a wrapper tool for the Linux Traffic-Control tool [179].

Table 6.2: Emulated network properties of 11 AWS regions.

	Bandwidth (Mb/s)	**Latency (ms)**
mean	67.28	179.97
std	41.70	79.19
min	16.76	27.20
25%	40.75	125.63
50%	56.20	171.40
75%	71.21	227.60
max	212.20	359.24

Network emulation

We emulate 11 AWS regions in our experiments, including Virginia, California, Oregon, Ireland, Frankfurt, Tokyo, Seoul, Singapore, Sydney, Mumbai, and São Paulo. The statistical description of the emulated network latency and bandwidth values between those data centers can be found in Table 6.2. The actual values for bandwidth are taken from a recent work [119], whereas the latency values can be found online[1]. The minimum bandwidth is between Singapore and São Paulo while the maximum bandwidth is between California and Oregon. The minimum latency is between California and Oregon while the maximum latency is between São Paulo and Sydney.

6.5.3 System setup

We compare Nitro with three existing systems, namely OpenStack Swift, InterPlanetary File System (IPFS) and BitTorrent.

Nitro. The key/value backend storage of Nitro is implemented using Redis [238], an in-memory database. We configure Redis to use append-only log and to sync the changes to the disk every second to ensure data persistence. Our chunk scheduling algorithm is implemented on top of the min-cut max-flow algorithm provided by the `networkx` python library version 2.0. We choose a chunks size of 256 KB which provides an appropriate trade-off between higher compression ratio and minimal metadata overhead. Also, 256 KB is the default chunk size for BitTorrent and IPFS.

OpenStack Swift. Swift is the distributed object-store system in OpenStack, and it is usually used as backend storage for image service. Swift supports multi-region deployment. It has been running in production to store containers image for 8 data centers in IBM clouds [17]. We perform our experiments with the *Ocata* version of Swift [210]. The read and write affinity properties are configured according to the bandwidth between the data centers.

IPFS. IPFS manifests as a potential solution for VMIs as it has built-in deduplication and it is designed to work in distributed environments. The IPFS version

[1] https://www.cloudping.co, November 2017

0.4.10 [132] is used for evaluation. IPFS uses Leveldb [169], a key-value store, as its backend storage system.

BitTorrent. BitTorrent is a classical peer-to-peer content distribution protocol. It is widely used to distribute large multimedia files and linux distribution images. We used the open-source BitTorrent client `libtorrent` [174] version 1.1.4 and open-tracker [211] as a BitTorrent tracker in the experiments.

In our experiments, each data center is represented by one machine. For BitTorrent, an additional machine is used to run the tracker with an emulated latency of 25 ms and a bandwidth of 100 Mb/s to other machines. The images in the compressed format are used as input by BitTorrent and Swift, and in the raw format by IPFS and Nitro as they are going to be deduplicated (and compressed) later by each system.

6.6 Evaluation Results

For the evaluation, we first evaluate Nitro internals. Then, we evaluate Nitro against other VMI management systems.

6.6.1 Effectiveness of Nitro

We evaluate the effectiveness of Nitro in three ways: 1) we show the impact of the global deduplication; 2) we compare the network-aware scheduling with random scheduling; and 3) we evaluate the runtime of the network-aware algorithm.

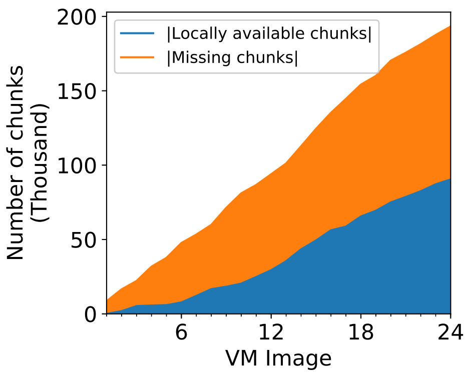
Global deduplication

As we have discussed, the strength of deduplication compared to other compression techniques is the detection of identical blocks of data (i.e., chunks) on the dataset level. Therefore, with a larger dataset, we may obtain a higher compression ratio, as the chances of finding identical blocks become higher. Here, we confirm the efficiency of deduplication for VMIs and present its impact on our dataset.

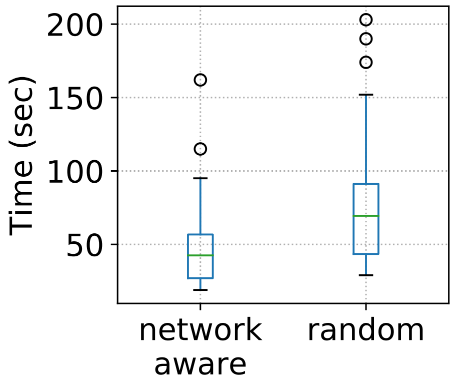
To show the reduction in storage cost (and thus the reduction in network traffic when retrieving a VMI), we measure the amount of locally available chunks for each newly added VMI while adding the images sequentially (in a random order) to the same node. In Figure 6.4a, the x-axis represents the size of the dataset and the y-axis shows the number of locally available chunks and the missing chunks when adding (requesting) the current VMI. We notice the increase of locally available chunks by each newly added image. The lower area in blue represents the sum of chunks that are found locally for the complete dataset, i.e., the save in the storage (network) cost. Note that the number of locally available chunks for each image could be slightly different depending on the order in which the images have been added, but we can always see the same trend.

Advantages of network-aware scheduling

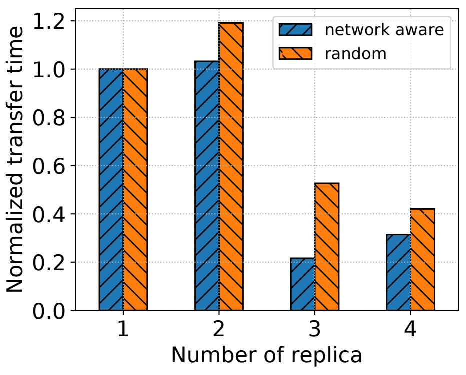
We evaluate the effectiveness of our network-aware chunk scheduler by first comparing it with the random chunk scheduler. In this experiment, each image in the dataset is



(a) Effectiveness of global deduplication



(b) Effectiveness of network-aware scheduling



(c) Replication sensitivity

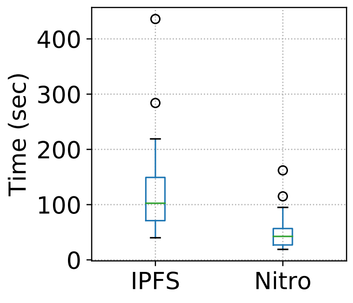
Figure 6.4: Evaluation results for Nitro internals: (a) Show the constant increase in redundant chunks (locally available) with the increase of dataset size (number of images) by using deduplication. (b) Compare the transfer time of network-aware scheduler and random scheduler. (c) Show how the transfer time of the two schedulers varies according to the initial number of replicas.

initially added to three sites randomly chosen and then requested from a fourth site. In Figure 6.4b, we present the provisioning time, i.e., network transfer time, for both schedulers. Results show that the network-aware scheduling of Nitro reduces the average provisioning time by 38% compared to the random scheduler.

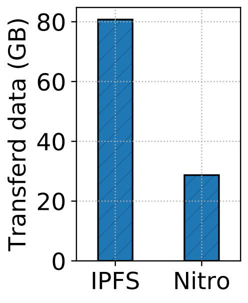
We also evaluate the impact of the number of replicas on the transfer times as shown in Figure 6.4c. The y-axis shows the normalized transfer time, however, note that the reported transfer times for different number of replicas are not directly comparable as they are measured on different nodes. With more replicas, the chunk scheduling problem has a larger solution space as the chunks to be pulled are available on more sites. Thus, it is possible to obtain better transfer time results with Nitro when the number of replicas is large. For example, when the number of replicas is three, Nitro reduces the average transfer time by 58% compared to the random scheduler. Whereas, if the chunks are available in a single site, the algorithm has no choice other than pulling all missing chunks from a single site (as we can see in the first column of Figure 6.4c). In conclusion,

Table 6.3: Scheduling algorithm runtime (seconds).

Sites	Number of MC	Runtime with MC	Runtime w/o MC
2	3	0.016	153.648
4	15	0.048	267.057
6	63	0.185	426.606
8	255	0.973	298.053
10	1023	8.035	2065.434



(a) Transfer time



(b) Network cost

Figure 6.5: Comparison between Nitro and IPFS in terms of (a) total transfer time during the experiment and (b) total amount of transferred data over WAN.

having more sources of data can help to reduce the provisioning time.

Runtime of network-aware scheduling

The runtime of the scheduling algorithm depends on many factors, such as the number of sites, number of chunks and the distribution of the chunks on sites as it determines the number of Mega Chunks (MCs). We measure the runtime while increasing the number of sites when pulling 10,000 chunks (representing 2.5 GB of data). We consider the worst-case scenario for chunks mapping, i.e., there is at least one chunk that can be found in each subset of sites. In this case, the number of mega chunks is $2^{|sites|} - 1$, as discussed in Section 6.3. In Table 6.3, we compare the runtime of the algorithm *with* and *without* the grouping optimization, i.e., Mega Chunks. The runtimes are reported in seconds and measured using a machine with 2,7 GHz Intel i5 CPU and 16 GB RAM. Results show that the grouping optimization can greatly reduce the runtime of the network-aware scheduling algorithm by up to 99.6%.

6.6.2 Nitro vs. IPFS

We choose IPFS for comparison with Nitro as it shares some similar design principles as Nitro. IPFS uses content-addressable storage to store data with deduplication applied. It transfers data by exchanging chunks between peers. Also, similar to Nitro, IPFS does not use compression. However, IPFS uses a Distributed Hash Table (DHT) to locate the chunks, whereas Nitro maintains the global distribution of chunks.

We compare the two systems in the same scenario as in the previous section with three replicas for each image. Figure 6.5a shows that Nitro reduces the network transfer time by 60% on average and 62% in the worst-case compared to IPFS. This can be explained by the fact that IPFS requests data from all available peers which have the data. As can be observed in Figure 6.5b, around 80 GB of data are transferred across the network with IPFS which is almost three times the size of the dataset. This technique is used by IPFS to tolerate network partitioning and weak links, but it results in high network cost.

6.6.3 Nitro vs. Swift and BitTorrent

We further compare Nitro against two other systems, namely Swift and BitTorrent. We enable the chunk compression in Nitro as in the compared systems. Chunk compression leads to different chunk sizes. As a result, the network-aware scheduling of Nitro does not guarantee the optimal solution. However, we keep this setting for fair comparisons with the other two systems. Although BitTorrent divides images into pieces, it does not apply any kind of deduplication. On the other hand, Swift uses point-to-point replication and internally relies on a modified version of `rsync` [205].

Provisioning time for single-site VMI request

In this scenario, we measure the transfer time for a single VMI request. Again, we set three replicas for each image and request each image from a fourth site. Figure 6.6a presents the results obtained by the studied systems. Nitro obtains the best network transfer time results. On average, it reduces the time by 77% compared to Swift and 46% compared to BitTorrent. Another observation is that, although Nitro does not guarantee optimal provisioning time for compressed chunks, the optimized VMI transfer time is still better than without compression when comparing Figure 6.6a with Figure 6.5a.

Provisioning time for multi-site VMI request

In our design, Nitro provides chunk scheduling solution without being aware of the current status of the cloud system. Thus, it is possible to cause network contention when multiple sites are requesting VMIs at the same time. To evaluate how serious the network contention problem can get, we provision three VMs at the same time from three different sites, where five replicas of each image are available on the same sites. Figure 6.6b shows the obtained results. Nitro is still able to outperform the other two systems, with 53% and 90% reduction in the network transfer time compared to BitTorrent and Swift, respectively. This is mainly because the deduplication with compression in Nitro can reduce the size of data to be transferred across the network compared to compression alone in Swift and BitTorrent (refer to Table 6.1) by 56%. Moreover, the fact that Swift relies on point-to-point communication and cannot do the copy in parallel from different sites lies it behind the other two systems.

Sensitivity study on replication

The efficiency of peer-to-peer systems increases when the number of participating peers increases. Thus, we evaluate Nitro and BitTorrent when the participating sites, i.e., sites

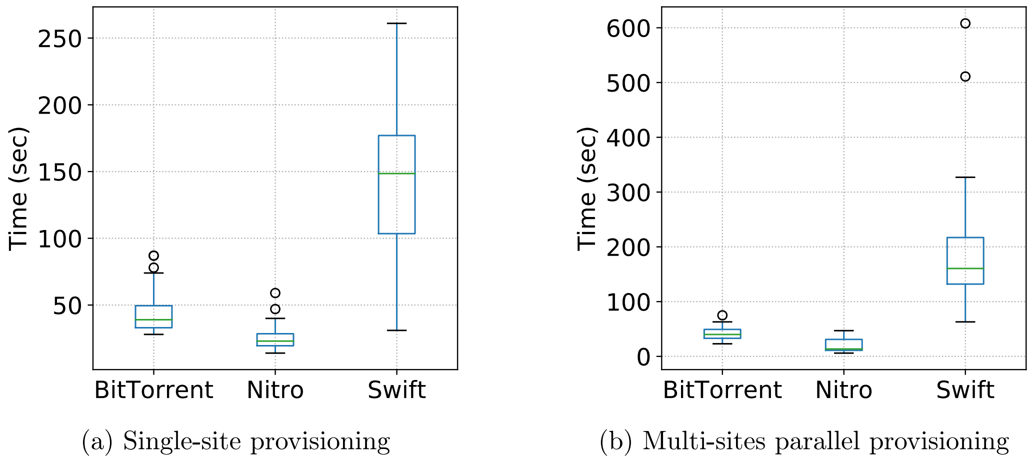


Figure 6.6: Comparison results of Nitro, Swift and BitTorrent: (a) The transfer time of the systems when requesting one VMI to one site. (b) The transfer time of the systems when requesting three VMIs to three sites, at the same time.

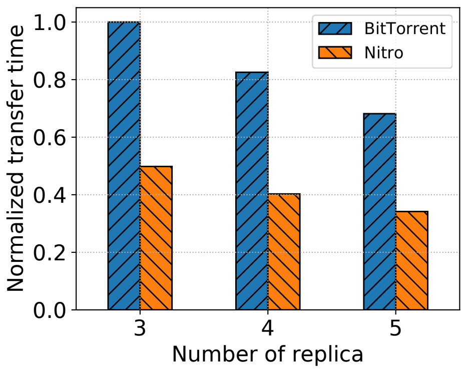


Figure 6.7: Sensitivity study on replication for Nitro and BitTorrent when the data can be found in 3,4, and 5 sites.

that have copies of the requested chunks, increase from three to five. Each image is initially uploaded to three random sites. Then, three different random sites request the image sequentially: the first site can request the image (chunks) from three locations, while the second and the third sites can request the chunks from four and five locations, respectively. We repeat the same steps for all the images. Figure 6.7 shows the normalized network transfer time results.

We have two observations. First, the network transfer times of both Nitro and BitTorrent decrease with the increase of the number of replicas. This is consistent with our expectation. Second, Nitro outperforms BitTorrent in all cases, by up to 50%, mainly because Nitro leverages local chunks, thus, it transfers less amount of data over the network compared to BitTorrent.

6.7 Applicability and Limitations

In this section, we discuss the applicability and limitations of Nitro under different network models, when requesting more than one VMs, and when employing file-level deduplication.

6.7.1 Wide area network models

In the literature, there are mainly two models to represent WANs at the application level:

- **pair-wise bandwidths**: In this model, there is a specific bandwidth between each pair of data centers. Therefore, the network is considered as a complete graph; hence, a site can receive data from all the other sites in parallel without any degradation in aggregated incoming bandwidth (i.e., each site has an infinite download bandwidth). This model has been employed in many recent works [156, 283, 122, 119, 47].

- **uplink/downlink bandwidth**: In this model, each data center has a specific bandwidth for its uplink/downlink with the assumption that the core network connecting these data centers is congestion-free (i.e., infinite core bandwidth). Therefore, the bandwidth between two sites is computed as the minimum of the sender upload link bandwidth and the receiver download link bandwidth. However, in this case, other parameters are not taken into account (e.g., geographical proximity) which lead to non-consistent pair-wise bandwidths. Moreover, selecting the reference point to compute the upload/download bandwidths is also problematic. This model has been employed in some recent works [226, 318, 126].

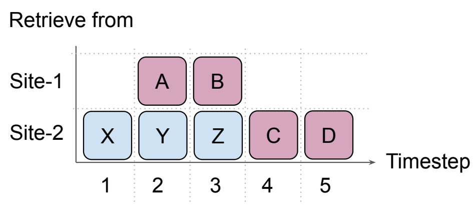
Both models provide a good enough simplification for the application as the physical characteristics of the network are hidden (e.g., redundant links, routers, switches, etc.). In our work, we rely on the first network model as current software level approaches as Traffic Engineering (TE) [115] and Software-Defined Networking (SDN) [31] are widely employed to abstract the physical network topology from the application and ensure some network properties (e.g., bandwidth) between sites. As a consequence, concurrent retrieval of VMIs from different sites do not impact each other as these sites are connected with dedicated links. However, the second network model has an impact on concurrent retrieval of VMIs.

6.7.2 Network performance variability

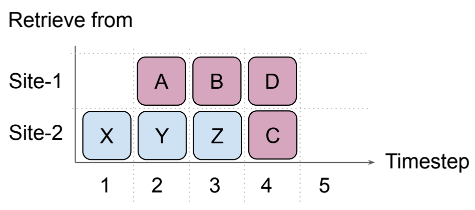
Currently, Nitro has no mechanism to deal with the variation of network performance (i.e., bandwidth) during the retrieval of an image. Previous works have shown that the available bandwidth between two sites is relatively stable in the granularity of minutes [226], or even 10 minutes [283] which is sufficient to complete a transfer (see Section 6.6).

6.7.3 Requesting multiple VMs from the same site

Although in our work we focus on single VMI request, our scheduling algorithm also works when requesting multiple VMIs from the same site *at the same time*. This is



(a) Current scheduling



(b) State aware scheduling

Figure 6.8: The chunk schedule of the current and modified algorithms. Image-2 (X,Y,Z) is requested at timestep 0 and image-1 (A,B,C,D) is requested at timestep 1. The bandwidths are homogeneous.

because our algorithm is oblivious to the number of VMIs and works at the chunk level. On the other hand, in multi-tenant environments, requests for VMIs could come while other VMI are being retrieved. However, the current scheduling algorithm assumes that the network links are fully available, leading to non-optimal scheduling for the newly requested images.

To illustrate the problem, we suppose a scenario with two images (image-1 and image-2) and two sites (site-1 and site-2) connected with a third site with homogeneous bandwidth for simplicity. Image-1 consists of four chunks: A, B, C, and D and it is available on both sites, while image-2 consists of 3 chunks: X, Y, and Z and it is only available on site-2. If image-2 is requested at timestep 0, its chunks will be pulled from site-2 as there is no other option. Later on, if image-1 is requested at timestep 1, the current scheduler will decide to pull two chunks from one site, and the other two chunks from the other site to balance the load across sites, as it supposes that the links are fully available. However, as the link to site-2 is busy with image-2, the scheduling of image-1 is not the optimal one. The optimal scheduling is to retrieve one chunk (C or D) from site-1 and the three remaining chunks from site-2, as shown in Figure 6.8.

To cope with this behavior, the current system state (i.e., the bandwidth usage) should be considered by the chunks scheduler. In particular, the modifications only concern the **add_capacity_to_sink_edges** method in Algorithm 1; when the capacities of the sink edges are set, the current remaining chunks (i.e., chunks that are already requested but not yet received) are taken into account (i.e., subtracted from the capacities of the links). These modifications can be integrated easily into Nitro.

6.7.4 Discussion on file-level deduplication in Nitro

Nitro relies on fixed-size deduplication to reduce the network overhead and provide flexible retrieval of images. However, using file-level deduplication instead could further reduce the network overhead and provide more image management functionalities [241, 180]. Employing file-level deduplication leads to heterogeneous chunk sizes as file sizes vary significantly; between few kilobytes and hundreds of megabytes [180]. A possible solution is to apply fixed-size deduplication to the individual unique files. This solution can be integrated seamlessly in Nitro and does not impact the optimality of the chunk scheduling algorithm, but it may result in higher metadata overhead.

6.8 Conclusion

In contrary to a single data center, service provisioning in geo-distributed clouds is impacted by the WAN. Provisioning services in geo-distributed clouds require transferring VMIs across the low-bandwidth and highly heterogeneous WAN. This might result in longer provisioning time.

In this chapter, we introduce Nitro, a new VMI management system that is designed specifically for geographically-distributed clouds to achieve fast service provisioning. Different from existing VMI management systems, which ignore the network heterogeneity of WAN, Nitro incorporates two features to reduce the VMI transfer time across geo-distributed data centers. First, it makes use of deduplication to reduce the amount of data which is transferred due to the high similarities within an image and in-between images. Second, Nitro is equipped with a network-aware data transfer strategy to effectively exploit links with high bandwidth when acquiring data and thus expedites the provisioning time. We evaluate Nitro by emulating real network topology. Results show that the network-aware data transfer strategy offers the optimal solution when acquiring VMIs while introducing minimal overhead. Moreover, Nitro outperforms state-of-the-art VMI storage system (OpenStack Swift) by up to 77%. Finally, we pointed out the applicability of Nitro for file-based deduplication and continuous image requests while discussing its limitations when applied in different network representations.

Chapter 7

Network-aware Service Image Placement in Edge Infrastructures

Contents

Edge computing extends the cloud by moving computation and storage resources physically close to data sources. The motivation behind that is to facilitate the deployment of short-running and low-latency applications and services [93, 261].

In general, services are deployed as containers in Edge [135, 37, 5], therefore, container images are needed to run those services. Provisioning a service in the Edge usually requires pulling the corresponding container image over the wide area network (WAN) from a central repository. This may result in long provisioning time which is unacceptable for most Edge services and applications (i.e., latency-sensitive applications).

To this end, in this chapter, we consider another approach: instead of putting all the container images in a central repository, we distribute these images across the Edge-servers. Thus, trying to leverage the network bandwidth in-between them and exploit small fractions of their available storage capacities. Accordingly, we propose and evaluate through simulation two novel container image placement algorithms based on k-Center optimization. In particular, we introduce a formal model to tackle down the problem of reducing the maximum retrieval time of container images. Based on the model, we propose two placement algorithms which target reducing the maximum retrieval time of container images to any Edge-server. Simulation results show that the proposed algorithms can outperform state-of-the-art algorithms.

This chapter is organized as follows. First, Section 7.1 introduces the limitation and challenges of container provisioning in Edge computing. Next, the problem alongside the proposed algorithms are formalized in Section 7.2. Simulation methodology is discussed in Section 7.3. The obtained results are presented in Section 7.4 and discussed in more details in Section 7.5. Finally, Section 7.6 concludes the chapter.

7.1 Container Provisioning in the Edge: Limitations and Challenges

Traditional approaches for service provisioning, which rely on centralized repository located in the cloud to store container images, may hinder the benefits of Edge in providing fast service provisioning: the bandwidth between cloud and Edge-servers can be a few tens of Mb/s [7, 107], thus, an image of a size of 1 GB needs at least 100 seconds to be transferred to the Edge-server if the bandwidth is 10 Mb/s. Consequently, when scaling out one of the killer applications in Edge such as live video stream analytics [237], it is not acceptable to wait for hundreds of seconds to provision a new container while the response time of this application is in the order of milliseconds. On the other hand, keeping the container image in the cloud and pulling only a part of the image which is enough to boot the container might improve the provisioning time, but it results in a performance overhead to the running services as requests to the image will be made over WAN [109, 75, 317]. Moreover, approaches which try to optimize the provisioning time, by overlapping data transfer and data extraction [5] or by prefetching and caching important layers [17], are still dictated by the amount of data which is pulled over WAN. Hence, current efforts may fail in practice to achieve the desired provisioning time of services in the Edge.

Alternatively, storing all the images locally is not feasible, especially as Edge-servers have limited storage capacity and the local storage is better to be exploited to store application data instead of container images. However, given that Edge-servers are featured with high network bandwidth among them compared to the bandwidth with clouds (e.g., running distributed IoT application on a cluster of Edge-servers is 5.3x times faster than running it on clouds [304]); a possible solution is to distribute container images across Edge-servers, thus, the network bandwidth and available storage (a small fraction) can be exploited efficiently. Unlike most approaches in the cloud, container image retrieval in Edge environments needs to be aware of the network heterogeneity between Edge-servers. Even worse, container images and layers are highly heterogeneous. As a result, the retrieval time depends on the image sizes and the distribution of their layers (and the replicas: usually layers are replicated for performance and fault-tolerance); hence, it is hard to predict the retrieval time of a container image.

In this chapter, we argue that the initial placement is important for *fast* service provisioning in Edge environments. Moreover, a service can be provisioned on multiple Edge-servers and an Edge-server may host multiple applications (e.g., camera devices host multiple applications [237]), therefore, it is essential to ensure *predictable* provisioning time as well. Our work tackles this problem (i.e., fast and predictable service provisioning in the Edge) by introducing novel placement algorithms that target reducing the maximum retrieval time of an image to any Edge-server.

To the best of our knowledge, no previous studies have worked on container image placement in Edge environments or targeted reducing the maximum retrieval time of container images.

7.2 Container Image Placement

In this section, we introduce the formal model we use to study the container image placement problem. We also introduce the two heuristics we propose to distribute a set of replicated layers through a network (across Edge-servers). In Section 7.2.1, we focus only on a set of individual layers and try to optimize the maximal retrieval time. In Section 7.2.2, we show how to extend the problem to a set of images (that are themselves sets of layers) and how to adapt our placement.

7.2.1 Layer placement

Formal model

For the moment, we focus on layers placement and put aside complete images. First, a *layer* l_i is defined by its *size* s_i (i.e., its storage cost) and its *replication number* n_i (i.e., how many times a layer is replicated). We denote by $\mathcal{L}_i = \{l_i^1, \ldots, l_i^{n_i}\}$ the replicas of l_i. In the following, the complete set of layers is denoted as $\mathcal{L}$ and $\mathcal{L}^R$ represents the set of replicas ($\mathcal{L}^R = \bigcup \mathcal{L}_i$).

In our model, the infrastructure is defined as a set of *nodes* V that are fully connected (thus a complete graph). We denote as c the *storage capacity* of all nodes (the allocated storage space on each node). For all the pairs of nodes u,v, we denote as b_{uv} the bandwidth between these two nodes (if $u = v$ then $b_{u,v} = +\infty$).

Given a set of layers $\mathcal{L}$, a set of nodes V and a storage capacity c, we define a *placement* as a function σ from $\mathcal{L}^R$ to V (we want to place all replicas). A placement is said to be *valid* if for each $u \in V$, $\sum_{l_i^k \in \sigma^{-1}(u)} s_i \leqslant c$ (i.e., the sum of the sizes of stored replicas does not exceed the storage capacity) and for each $k,k' \in [1,n_i]$, $\sigma(l_i^k) \neq \sigma(l_i^{k'})$ (i.e., replicas of the same layer have to be placed on different nodes).

From a valid placement, we derive the *retrieval time* of a layer l_i on a node u as follows. Let u_i^σ be the node owning a replica of l_i that is the closest to u (i.e., with maximal bandwidth, formally $u_i^\sigma = \arg(\max_{v \in \sigma(\mathcal{L}_i)} b_{uv})$. The retrieval time of l_i on u is thus $T_i^u = \frac{s_i}{b_{uu_i^\sigma}}$. Our goal here is to minimize the maximal retrieval time for all layers on all nodes. We denote this problem MaxLayerRetrievalTime.

Problem 1 (MaxLayerRetrievalTime)**.** *Let V be a set of nodes with storage capacity c and $\mathcal{L}$ be a set of layers. Return a valid placement that minimizes:* $\max\limits_{u \in V,\ l_i \in \mathcal{L}} T_i^u$.

MaxLayerRetrievalTime is close to the k-Center problem, that aims to place facilities on a set of nodes to minimize the distance from any node to the closest facility. See below for a formal definition.

Problem 2 (k-Center)**.** *Given a set V with a distance function d (defined between all elements of V), and a parameter k, return a set $S \subseteq V$ such that $|S| = k$ that minimizes:* $\max\limits_{u \in V} \min\limits_{v \in S} d(u,v)$.

MaxLayerRetrievalTime is similar to the k-Center problem if considering only one layer. k-Center problem is known for being NP-complete [94] and thus MaxLayerRetrievalTime is also NP-complete. In addition, it has been proven that the best possible approximation is a 2-approximation (unless P=NP) [120].

The solution we introduce to solve MaxLayerRetrievalTime is based on a solver for k-Center. The basic principle of this heuristic, KCBP (k-Center-Based Placement), is to sort the layers in descending order by their sizes and then use this solver several times to place replicas, one layer after another. The distance used is the inverse of bandwidths. A pseudo-code of KCBP is provided in Algorithm 2.

For our implementation, we use SCR, a polynomial k-Center problem solver that was introduced by Robič and Mihelič [246]. SCR is based on a pruning technique (i.e., removing some edges and find a solution on the induced subgraph) and a Dominant Set problem solver (as the Dominant Set problem is also NP-complete, SCR relies on a heuristic). Note that SCR is a 2-approximation, but its experimental average approximation factor is far better and as far as we know the best among polynomial heuristics: 1.058 for SCR on a classical benchmark for graph partitioning [246].

The time complexity of SCR is $O(m^2 \log m)$, where m is the number of nodes. Hence, the overall complexity of KCBP is $O(|\mathcal{L}||V|^2 \log |V|)$.

7.2.2 Image placement

MaxLayerRetrievalTime focuses on layer retrieval. However, as we target the retrieval of complete container images, in this section, we formalize and present MaxImageRetrievalTime.

Algorithm 2: k-Center-Based Placement

```
   Input  : $\mathcal{L}$, $V$, $c$
   Output: $\sigma$
1  Sort $\mathcal{L}$ by decreasing size ;
2  foreach $u \in V$ do
3      $c_u = c$ ;
4  end
5  foreach $l_i \in \mathcal{L}$ do
6      $V' = \{u \in V, c_u \geqslant s_i\}$ ;
7      $S = Scr(V', n_i)$ ;
8      $k = 1$ ;
9      foreach $u \in S$ do
10         $\sigma(l_i^k) = u$ ;
11         $k++$ ;
12         $c_u \leftarrow c_u - s_i$ ;
13     end
14 end
15 return $\sigma$ ;
```

Formal definition

We define an *image* as a set of layers $I_j = \{l_{i_1}, \ldots, l_{i_q}\}$. The complete set of images is denoted as $\mathcal{I}$. To retrieve an image I_j, a node u has to download a replica of each layer that is in I_j. As a first approximation, we could consider the downloads are performed in parallel and thus the retrieval time is defined by the largest retrieval time among these different layers (as in MaxLayerRetrievalTime). However, multiple downloads from the same source may degrade the performance by reducing the bandwidth. Therefore, in our model, we consider that if a node requests an image that requires several layers where the closest replicas are on the same node, then the download of these replicas is made sequentially (that is equivalent to do it in parallel with shared bandwidth). More formally, given an image I_j, a valid placement σ, and a node $u \in V$, let $V^{\sigma}_{u,I_j} = \{v \in V, l_i \in I_j \text{ and } u_i^{\sigma} = v\}$ the set of nodes that are the closest nodes to u for at least one replica of the layers of I_j. The retrieval time of an image I_j is thus:

$$T^u_{I_j} = \max_{v \in V^{\sigma}_{u,I_j}} \frac{\sum\limits_{i, u_i^{\sigma} = v} s_i}{b_{uv}}.$$

We define now MaxImageRetrievalTime where the goal is to minimize the maximal retrieval time of a set of images.

Problem 3 (MaxImageRetrievalTime)**.** *Let V be a set of nodes with storage capacity c and $\mathcal{I}$ be a set of images. Return a valid placement that minimizes:* $\max\limits_{u \in V, I_j \in \mathcal{I}} T^u_{I_j}$.

Algorithm 3: k-Center-Based Placement Without-Conflict

```
Input  : $\mathcal{I}$, $\mathcal{L}$, $V$, $c$, $f$
Output: $\sigma$
1  Sort $\mathcal{L}$ by decreasing size ;
2  foreach $u \in V$ do
3      $c_u = c$ ;
4  end
5  foreach $l_i \in \mathcal{L}$ do
6      $V' = \{u \in V, c_u \geqslant s_i\}$ ;
7      if $l_i$ is one of the $f\%$ largest layer then
8          foreach $I_j \in \mathcal{I}$ such that $l_i \in I_j$ do
9              $V' \leftarrow V' \backslash \{u \in V', \exists l_{i'} \in I_j, \sigma(l_{i'}^k) = u\}$
10         end
11     end
12     $S = Scr(V', n_i)$ ;
13     $k = 1$ ;
14     foreach $u \in S$ do
15         $\sigma(l_i^k) = u$ ;
16         $k++$ ;
17         $c_u \leftarrow c_u - s_i$ ;
18     end
19 end
20 return $\sigma$ ;
```

Without-Conflict

If two layers are part of the same image, then their replicas should not be on the same nodes. However, applying this constraint to all layers can lead to a large spreading of the replicas and even to a lack of eligible nodes (i.e., nodes with enough remaining storage capacities and have no conflicting layers). Thus, we limit the number of layers that are concerned. More precisely, we add a parameter f that is a percentage of the layers. If a layer l_i is among the $f\%$ largest layers, then this layer cannot be placed on a node that already has a replica of a layer $l_{i'}$ which belongs to the same image I_j. We denote this algorithm KCBP-WC (KCBP-Without-Conflict) and a pseudo-code is provided in Algorithm 3.

7.2.3 Limitations

To simplify the modeling of the problem, we assume that there is a direct link between each pair of Edge-servers. Though, this is a strong assumption, but it may be somewhat satisfied with current (continuous) progress in software level approaches, as discussed in Section 6.7, which tries to abstract the complexity of the underlying network hardware. Thus, the general trend of the relative performance between the placement algorithm could be preserved. Note that this is the first work on the domain of container image placement in Edge environments where the main goal is to shed light on the importance

Table 7.1: The sizes and links bandwidths characteristics of the studied networks.

Network	**Number of nodes**	**Links bandwidths (b/s)**				
		min	**25th**	**median**	**75th**	**max**
Homogeneous	50	4G	4G	4G	4G	4G
Low	50	8M	763M	1G	2G	8G
High	50	478M	5G	6G	7G	8G
Uniform	50	8M	2G	4G	6G	8G
Renater	38	102M	126M	132M	139M	155M
Sanet	35	63M	6G	8G	8G	10G

of container image placement in the Edge, thus assuming this simplification.

7.3 Simulation Methodology

We developed a simulator in Python to evaluate the performance of the two proposed placement algorithms on different networks and using a production container image dataset.

Our simulator is written in Python and the source code is publicly available at `https://gitlab.inria.fr/jdarrous/image-placement-edge`.

7.3.1 Network topology

We generate synthetic networks with different bandwidth characteristics and use real-world networks topologies for our evaluation. All the networks are described in Table 7.1.

Synthetic networks

As we consider that network topologies have no interference, we generate complete graphs (i.e., there is a direct link between each pair of nodes) and then assign bandwidths to these links. Four distributions have been considered: (1) *Homogeneous*: where all the links have the same bandwidth. (2) *Low*: where the majority of the links have low bandwidth. (3) *High*: where the majority of the links have high bandwidth. (4) *Uniform*: where the links bandwidths follow a uniform distribution between 8 Mb/s and 8 Gb/s.

Real-world networks

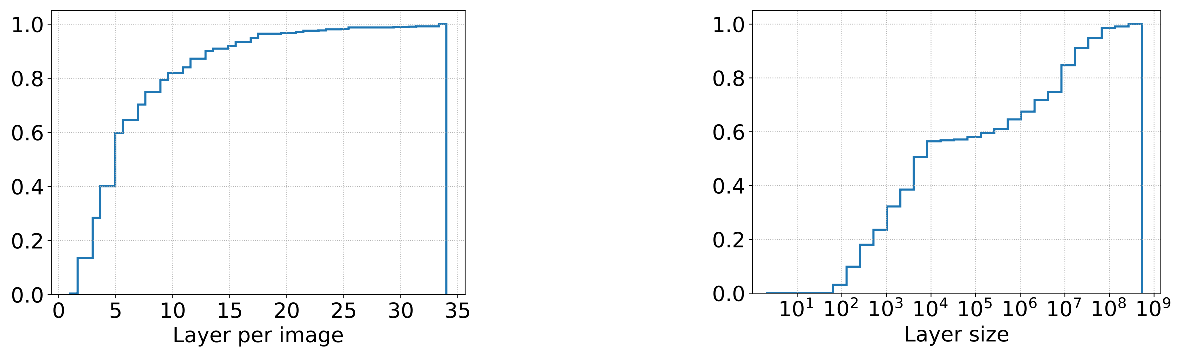
In addition to synthetic networks, we choose two real-world networks to demonstrate the applicability of our algorithms. We select the national networks of France (Renater) and Slovakia (Sanet) [271]. When two nodes are not directly connected, we set the bandwidth between these nodes to the minimum bandwidth of the links that form the shortest unweighted path between these two nodes [152].

7.3.2 Container images dataset

Container images and their corresponding layers are retrieved from publicly released IBM Cloud traces [17]. We extract the images and layers from the traces of *Frankfort* data center. The dataset is composed of 996 images with 5672 layers, see Table 7.2. The majority of these images (56%) have between 5 and 15 layers, however, some images are composed by up to 34 layers, see Figure 7.1a. The layers are highly heterogeneous in size (vary from 100 B to 1 GB). Moreover, 30% of the layers are larger than 1 MB, see Figure 7.1b.

Table 7.2: The characteristics of the considered image dataset.

Total number of images	996
Total size of images	93.76 GB
Total number of layers	5672
Total size of unique layers	74.25 GB



(a) CDF of the number of layers per image

(b) CDF of layer size (byte)

Figure 7.1: The characteristics of layers.

7.3.3 Node storage capacity

For each network, we limit the nodes' capacities according to the total dataset size and number of nodes. First, the theoretical minimum node capacity that is needed to store all the layers (considering that the layers can be split at a byte level) is equal to the dataset size (with replication) divided by the number of nodes. However, this capacity does not satisfy any placement in practice as the integrity of the layers should be preserved. This can be achieved by storing only a complete layer on the same node. Therefore, we set the nodes' capacities as the theoretical minimum capacity multiplied by a *capacity scaling* factor. In our experiments, we test the following values for the *capacity scaling* factor: 1.1, 2, and INF. We include INF – which represents unlimited storage – just for comparison.

7.3.4 State-of-the-art placement algorithms

We compare our proposed algorithms with two placements algorithms: Random and Greedy (Best-Fit). These algorithms are not network-aware (i.e., they do not take into account links bandwidths) and serve as a comparison baseline.

Best-Fit placement

The Best-Fit placement is a greedy algorithm to place layers on nodes. The algorithm places the replicas of a layer l_i on the n_i nodes with the largest remaining storage capacity. The algorithm iterates over all the layers sorted by their decreasing size. As a result, Best-Fit distributes the layers evenly on the nodes in such a way that the nodes have almost the same total storage cost. When the layers have the same size, Best-Fit behaves similar to a round-robin distribution. The algorithm is deterministic when the nodes are initially presented in the same order, therefore, we shuffle the initial nodes ordering in every iteration to obtain a different placement.

Random placement

The Random placement serves as a base solution. The algorithm distributes the layers randomly on the nodes. For each layer, we filter out the nodes that do not have sufficient storage space to host that layer, and then we select r random nodes to place the layer's replicas.

7.3.5 Methodology

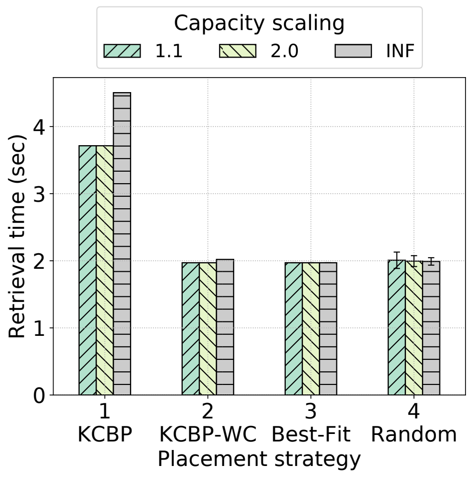
For our experiments, we consider a default replication factor of 3 (as many storage systems [263]) for each layer ($n_i = 3$ for all i), therefore, the total dataset size with replication is 3×74.25 GB. For KCBP-WC, we set the limit to define the "large" layers to 10%. For Best-Fit and Random placement algorithms, we run the placement 50 times and we draw the average retrieval time as well as the variation. KCBP and KCBP-WC are deterministic.

7.4 Simulation Results

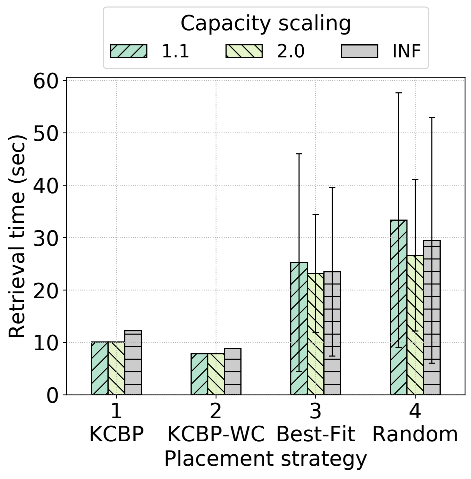
In this section, we present the results of our simulations on the container image dataset presented earlier. We first focus on synthetic networks before considering real-world networks.

7.4.1 Results for synthetic networks

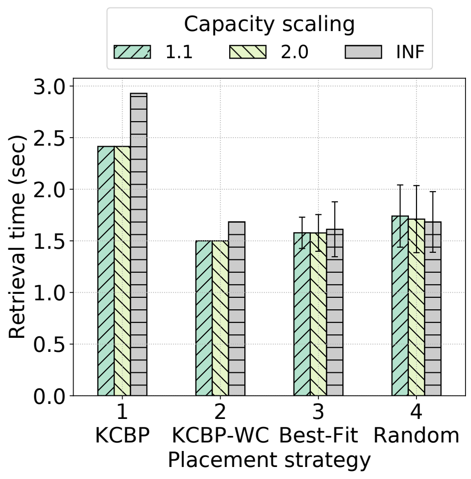
We would like to note that even though the retrieval times are presented in seconds, their relative values are more important than their absolute ones as the absolute retrieval time depends on the bandwidths of the network. Similarly, for the synthetic networks, the distribution of link bandwidths is the important factor, not their actual values.



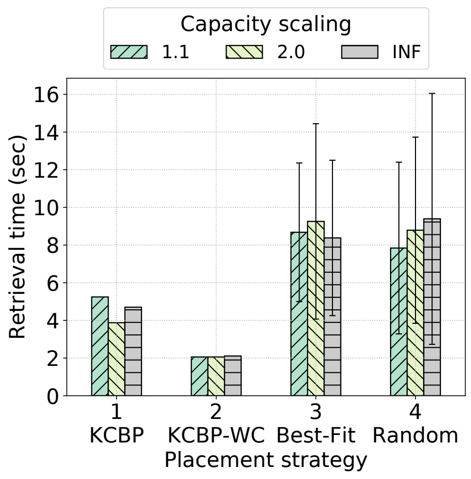
(a) Homogeneous Network



(b) Low Network



(c) High Network



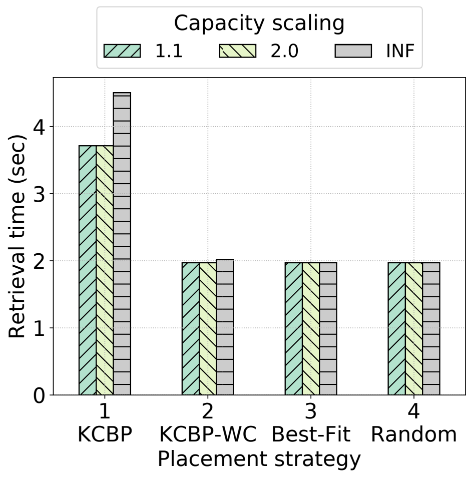
(d) Uniform Network

Figure 7.2: Retrieval time for synthetic networks.

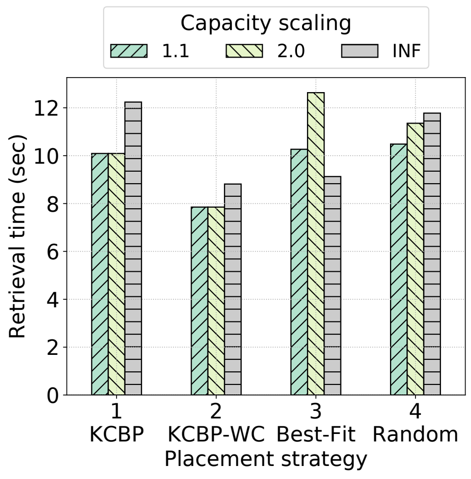
Homogeneous Network

All the links in the Homogeneous network have the same bandwidth, therefore, all the nodes have the same priority to place a layer. Figure 7.2a shows the retrieval times of the placement algorithms when varying the capacity scaling factor. When the capacity scaling factor is set to 2, KCBP-WC has a maximum retrieval time of 1.97s, which is 1.8x faster than KCBP that needs 3.71s.

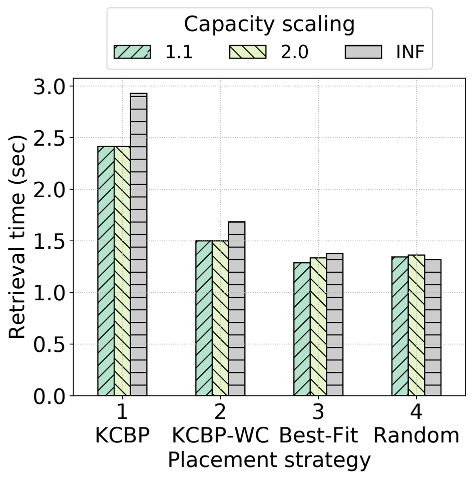
With homogeneous link bandwidths, placing layers of the same image on the same node can prolong the retrieval time as in the case of KCBP. KCBP-WC handles this by distributing the layers of the same images. Moreover, as the bandwidths are homogeneous, we can notice that the performance of KCBP-WC is similar to Best-Fit and Random because the links bandwidths have no impact on the optimal placement. Best-Fit has the same maximum retrieval time of KCBP-WC (i.e., 1.97s), while the maximum retrieval time of Random is 2.05s.



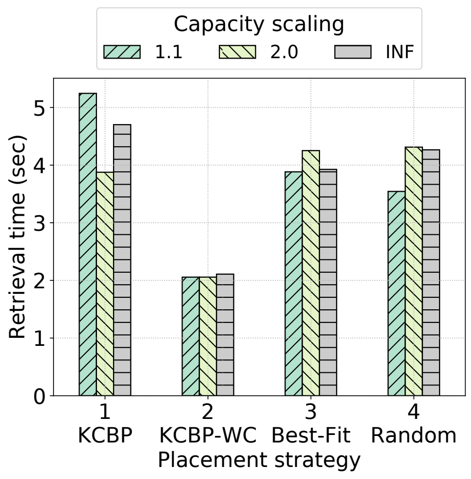
(a) Homogeneous Network



(b) Low Network



(c) High Network



(d) Uniform Network

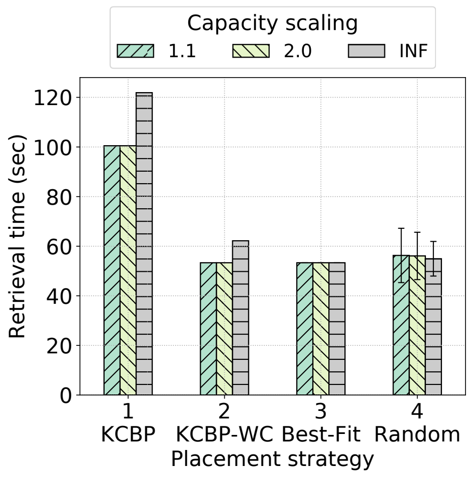
Figure 7.3: Retrieval time for synthetic networks where the best obtained solution is shown for Best-Fit and Random.

Low Network

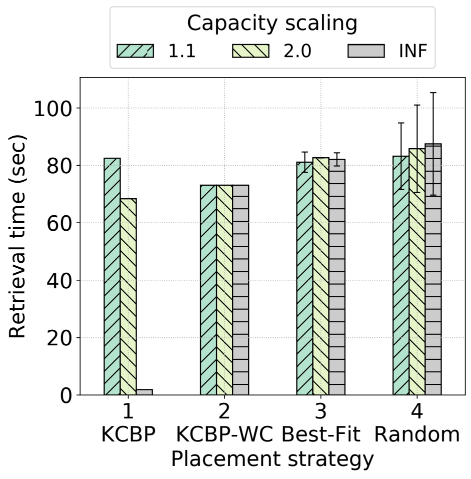
In this network, the majority of the nodes are not well connected, therefore, the placement of the layers (especially large ones) is critical for the retrieval performance. Best-Fit and Random experience a high variation in retrieval times and their best-found placements are still worse than that of KCBP-WC (Figure 7.2b and Figure 7.3b). For example, for a capacity scaling factor of 2, KCBP-WC achieves 7.85s while KCBP requires 10.09s. Best-Fit and Random have an average retrieval time of 23.15s and 26.63s, while their best retrieval times are 12.63s and 11.35s, respectively.

High Network

In the High network, the majority of links have high bandwidths, which shows that many nodes are well connected to the rest. In contrary to the Low network, the probability



(a) Renater Network



(b) Sanet Network

Figure 7.4: Retrieval time for real-world networks.

of placing a “large” layer on a low-connected node is smaller, and therefore, we notice a smaller variation in the performance for Best-Fit and Random. Moreover, Best-Fit and Random have better retrieval times than KCBP-WC in their best case (Figure 7.3c). They achieve 1.34s and 1.36s, respectively, while KCBP-WC has a retrieval time of 1.50s in case of capacity scaling factor of 2. Figure 7.2c depicts the results.

Uniform Network

The Uniform Network (Figure 7.2d) shows a similar trend to the Low network as both networks have a high percentage of low-bandwidth links and therefore low-connected nodes. We can notice that Best-Fit and Random exhibit high variation and KCBP-WC has better retrieval time even compared to their best case (Figure 7.3d).

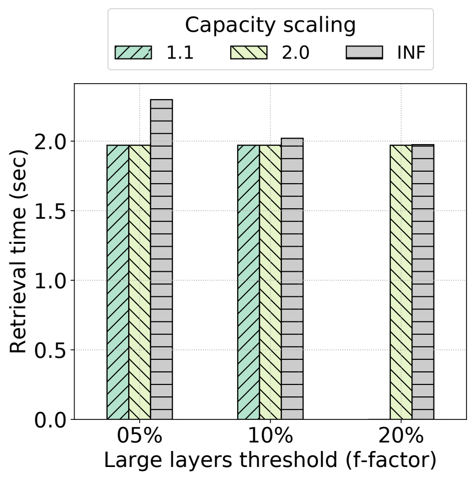
7.4.2 Results for real-world networks

Renater Network

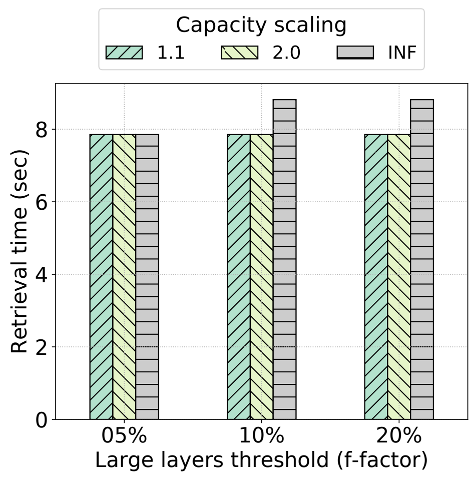
Renater Network exhibits only small variations for links bandwidths, therefore, it shows similar behavior to the Homogeneous network. For example, as we can see in Figure 7.4a, for a capacity scaling factor of 2, KCBP-WC has a maximum retrieval time of 53s while KCBP achieves 104s, which is more or less the ratio expected according to previous results on homogeneous bandwidth. Similarly to the Homogeneous network, we can see that the retrieval times of KCBP-WC are close to those of Best-Fit and Random.

Sanet Network

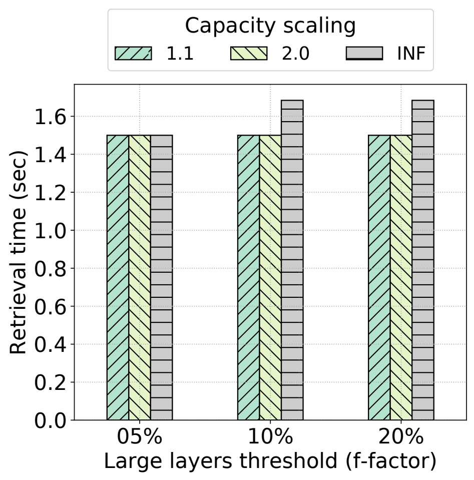
With Sanet network, the majority of links have high bandwidth (the bandwidths of 75% of the links are higher than 6 Gb/s). Thus, the results are close to that of High network. However, contrary to the High network, in this setup, KCBP performs better with increasing nodes capacities. For example, it achieves 82s, 68s, and 1.8s for 1.1, 2,



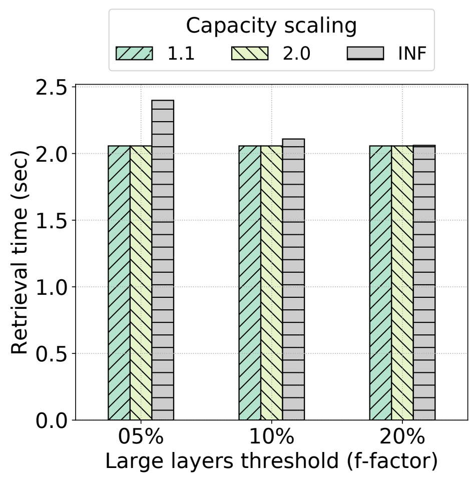
(a) Homogeneous Network



(b) Low Network



(c) High Network



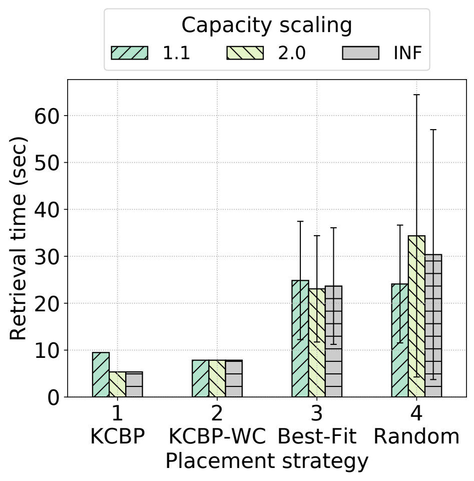
(d) Uniform Network

Figure 7.5: Retrieval time for KCBP-WC algorithm with different values for the f parameter on synthetic networks.

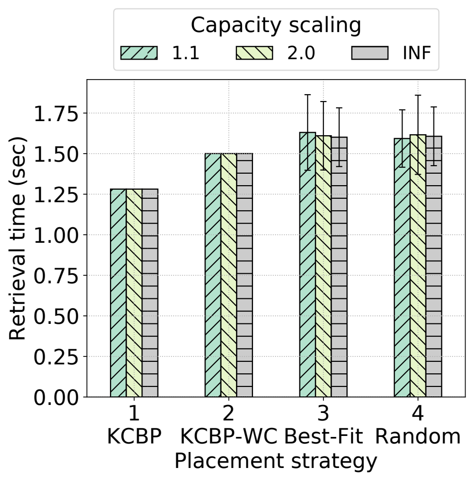
and INF capacity scaling factors, respectively (Figure 7.4b). The number of nodes (i.e., 35 against 50 for Sanet and High network, respectively) is the main reason behind the improvement of KCBP with Sanet compared to its performance with High network, as it increases the chance of placing layers on more central nodes. Similarly to the High network, KCBP-WC archives lower maximal retrieval times compared to Best-Fit and Random ($73s$, $81s$, and $83s$ respectively for a capacity scaling of 1.1).

7.5 Discussion

In this section, we discuss the previous results and highlight our findings. We focus on six aspects: conflicts, heterogeneity of the bandwidth, storage capacity, percentage of layers considered as large for KCBP-WC, optimal retrieval, and maximal retrieval time per image.



(a) Low Network



(b) High Network

Figure 7.6: Retrieval time of individual layers for synthetic networks.

7.5.1 Impact of conflicts

In Figure 7.6, we provide the maximum retrieval time for layers instead of images to evaluate the impact of conflict. As expected, all strategies are not impacted the same way. For example, KCBP doubles its retrieval time when images are considered instead of individual layers on the High network (Figure 7.2c and Figure 7.6b). However, for KCBP-WC, the impact is small (more obvious with INF capacity) as the algorithm avoids as much as possible putting layers of the same image on the same node. In the case of Random and Best-Fit, as they tend to produce close to even distribution of layers across the nodes, the probability of having more than one "large" layer on the same node is less than the case of KCBP and KCBP-WC, thus, conflicts between large layers are rare. Thus, in general, avoiding conflict is an important factor to consider while placing replicas.

7.5.2 Impact of the heterogeneity of the bandwidth

As expected, having nodes with different connectivity change the behavior of the placement strategies. Here, bandwidth-aware strategies can deal more efficiently with this heterogeneity, even when it is small (as in case of Renater network). However, we note some important differences between High and Low networks. In the first case, the difference in retrieval times between KCBP-WC and average values of Best-Fit or Random is small, and the variation in the performances of the last two is rather low. For Low network, this variation greatly increases as does the average performance. In this case, the performance of KCBP is even close to KCBP-WC. However, this does not apply to High network. Hence, it seems that centrality of layers is more important for Low network than for High network. In the first case, it is important to target a few nodes with high connectivity while in the second it is important to avoid the few nodes with low connectivity (at least for the largest layers). Thus, in Low network, KCBP compensates the conflicts with general good connectivity in comparison to the other node, while in

High network, the nodes used by KCBP are not that much better than the average ones. At the same time, it is easier for Best-Fit and Random to avoid few low connectivity nodes than to reach the few high connectivity ones. KCBP-WC can manage both situations when the number of nodes is sufficient, otherwise, as in Sanet, it may suffer from the spreading of the layers, as Best-Fit and Random do. Note that in Uniform network, where bandwidths differ significantly, KCBP-WC and KCBP perform well, even against the best results from Best-Fit and Random for KCBP-WC.

7.5.3 The extra space effect

A phenomenon that is interesting to point out is the fact that the performances of KCBP and KCBP-WC can be improved by decreasing the node capacity. For example, this effect is visible for KCBP-WC on the Homogeneous network, and for KCBP on all the networks. The reason for this phenomenon is that several layers of the same image are more likely to be placed on the same node (and thus retrieved from the same node) when the node capacity is larger. As explained earlier, we proposed KCBP-WC to avoid such placements that are common with KCBP. However, to avoid having layers dispersed on too many nodes (or not being able to place all layers), we only apply this strategy on "large" layers, implying that, in some cases, the placement of layers of the same image still happens and thus slightly decreases the performances.

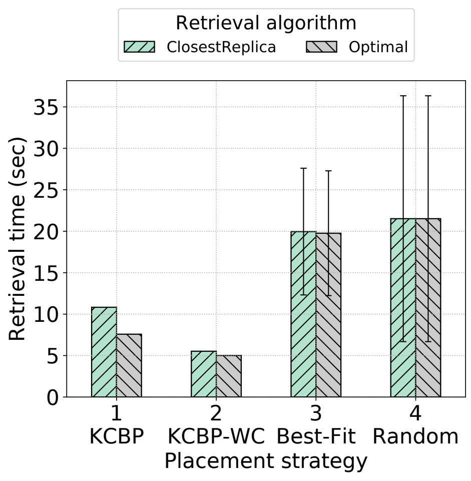
7.5.4 Impact of the percentage of layers concerned by KCBP-WC mechanism

Increasing the percentage of layers considered as "large" in KCBP-WC leads to a diminution of conflicts that should result in decreasing the retrieval time. However, it may also spread layers on nodes with low connectivity, leading to potentially longer retrieval times for some layers (and thus images). The extreme case is when there are no remaining nodes with enough storage capacity. As a result, KCBP-WC does not return a valid placement. This was the case on Sanet network which has only a small number of nodes (Figure 7.4b). When testing different values for this percentage (5%, 10%, and 20%) as shown in Figure 7.5, we observe almost no difference between them, except that with 20%. Moreover, the algorithm does not succeed to find a solution when the capacity factor is 1.1 on Homogeneous network.

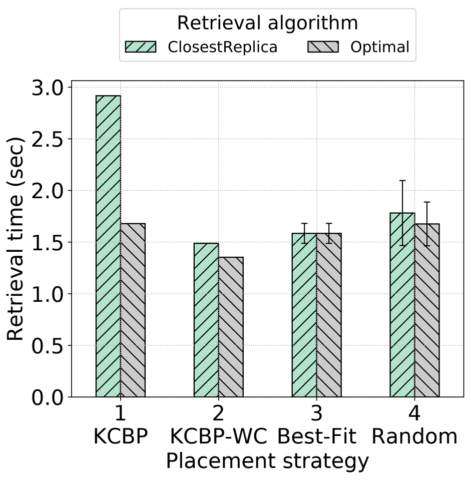
We cannot give a general conclusion from this result, as it is strongly correlated with the container image dataset, but it appears here that avoiding conflicts between only the large layers is enough and expanding this policy to smaller ones offers no clear gain.

7.5.5 The impact of optimal retrieval

As the goal of this work is the placement of container images, we rely on representative retrieval algorithm (i.e., retrieve from the closest replica) [227, 147] to evaluate the performance of the placement. However, to show the importance of data placement, we compared the performance of the different placement algorithms with closest replica retrieval and optimal retrieval. We implement the optimal retrieval of an image by simply testing all the possible combinations of its layers retrievals and select the best one.



(a) Low Network



(b) High Network

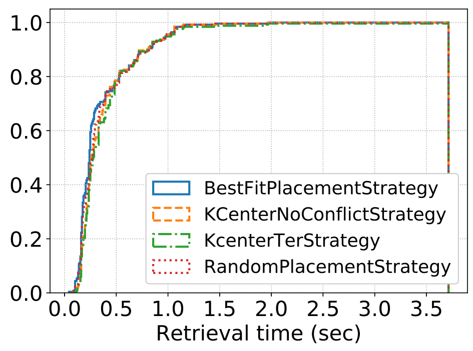
Figure 7.7: Optimal retrieval time for synthetic networks.

Figure 7.7 shows the maximum retrieval time for Low and High synthetic networks with both closest replica and optimal retrievals in case of INF capacity scaling. We observe that optimal retrieval shows no improvement under Best-Fit and Random for both networks compared to closest replica retrieval (expect a 5% improvement for High network under Random). More importantly, the performance of both retrieval algorithms is better under KCBP-WC compared to even the optimal retrieval algorithm when applied to Best-Fit and Random. This demonstrates the importance of the initial placement of layers on the retrieval time of container images.

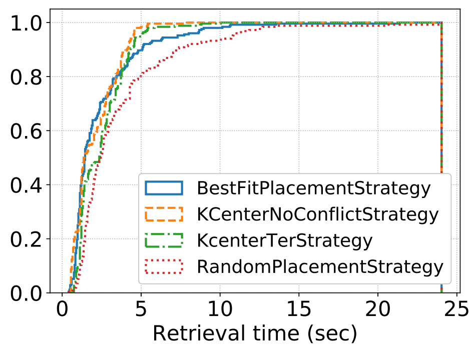
On the other hand, we observe that the highest improvement of optimal retrieval can be seen under KCBP (30% and 42% for Low and High networks, respectively). As KCBP concentrates layers on the same nodes, optimal retrieval prefers to retrieve some layers from another replica to increase the parallel retrieval. Optimal retrieval under KCBP-WC achieves around 9% improvement in maximal retrieval time for both networks. This is because some layers are not large enough to be included in the set of "large" layers of KCBP-WC, but are large enough to impact the retrieval time if placed on the same nodes that host "large" layers. Moreover, the results show that better maximal retrieval can be achieved with KCBP-WC (under both closest replica and optimal retrievals) than that with KCBP under optimal retrieval which validates the importance of the proposed KCBP-WC. In summary, while the placement as well as the retrieval of containers (i.e., layers) are important to achieve fast container provisioning time, however, the maximum benefit (the best provisioning performance) can be achieved when both are combined.

7.5.6 Maximal retrieval time per image

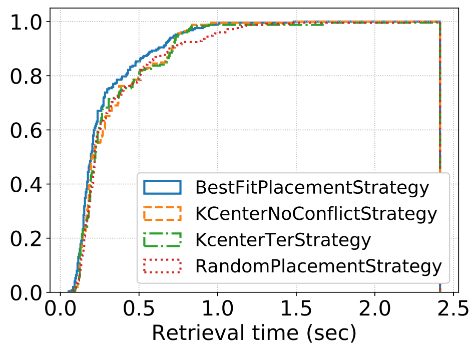
In this paragraph, we discuss the maximal retrieval time per image (i.e., the maximal time to retrieve an image to any node). The cumulative distribution functions (CDFs) are presented in Figure 7.8 for synthetic networks with capacity scaling factor of 2. We notice that the performances are close for all strategies in case of Homogeneous and High networks (Figure 7.8a and Figure 7.8c), with Best-Fit performing slightly better



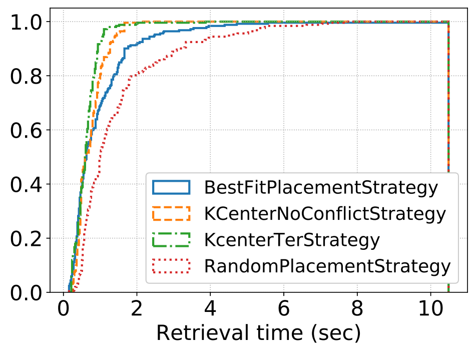
(a) Homogeneous Network



(b) Low Network



(c) High Network



(d) Uniform Network

Figure 7.8: CDF of maximal retrieval time per image for synthetic networks with storage capacity scaling factor of 2.

for a portion of the images (i.e., around 20% of the images on High network). On such networks, the overall high quality of links does not favor strategies that aim to spread as little as possible the different replicas, as pointed out earlier when evaluating the impact of bandwidth heterogeneity. Besides, KCBP and KCBP-WC favor the large layers, that may slightly degrade performances for images with smaller layers. Combined, these two reasons fully explain these results in comparison to the overall equal distribution of layers among the nodes proposed by Best-Fit (even if the difference is not that important). However, for Low and Uniform networks (Figure 7.8b and Figure 7.8d) the trends are different. In these networks, centrality is important and thus KCBP and KCBP-WC perform well, even when we are considering other images than the ones with maximal retrieval times. More precisely, KCBP and KCBP-WC present better maximal retrieval times for 20% of the images (images with longest retrieval time), with a small advantage for KCBP-WC that also performs better for other images. On Uniform network, KCBP outperforms all other strategies, at the exception of KCBP-WC that has finally a better overall maximum retrieval time (Figure 7.2d).

From these distribution functions, we observe that although KCBP and KCBP-WC mainly target large layers (and the images they belong to), their performances are "good enough" compared to Best-Fit, when considering all images. Note that, Best-Fit can propose better retrieval times for intermediate images when network bandwidth is overall high. In a network with lower connectivity, the centralization of layers we propose with KCBP and KCBP-WC allows general improvement of maximal retrieval time for images.

7.6 Conclusion

Service image management in Edge environments, especially container images, is gaining more attention with the wide adoption of Edge computing. In this chapter, we propose to store the images across the Edge-servers, in a way that the missing layers of an image could be retrieved from nearby Edge-servers. The main goal behind this approach is to ensure predictable and reduced service provisioning time. To this end, we have proposed two image placement algorithms based on k-Center optimization to reduce the maximum retrieval time for an image to any Edge-server. Through extensive simulation, using synthetic and real-world networks with a production container image dataset, we have shown that our proposed algorithms can reduce the maximum provisioning time by 1.1x to 4x compared to Random and Best-Fit based placements.

Part III

On the Efficiency of Erasure Coding for Data-Intensive Applications

Chapter 8

Erasure Coding in Distributed Storage Systems: From Data Archiving to Data Analytics

Contents

Replication has been successfully employed and practiced to ensure high data availability in large-scale distributed storage systems. However, with the relentless growth of Big Data and the wide adoption of high-speed yet expensive storage devices (i.e., SSDs and DRAMs) in storage systems, replication has become expensive [234, 252, 316, 110] in terms of storage cost and hardware cost. In recent years, erasure coding (EC) has become prevalent in distributed storage systems, owing to the low storage cost and the progress made in reducing its computation overhead [131]. For instance, EC has been integrated into the last major release of Hadoop Distributed File System (i.e., HDFS 3.0.0) which is the primary storage backend for data analytics frameworks (e.g., Hadoop [19], Spark [20], Flink [18], etc.). This facilitates running cost-effective data-intensive applications under EC, but introduces several challenges related to data locality and network overhead. Accordingly, in this part, by the means of experimental evaluation, we study the performance of data-intensive applications under EC and reveal its main effects of their performances. In addition, we design and evaluate a new data placement algorithm in Hadoop to improve the performance of data processing under EC.

In this chapter, we review the different application domains where EC has been employed and then, we present how EC is implemented in HDFS. After that, we present the state-of-the-art data processing under erasure coding and conclude this chapter. We remind the reader that the principles of EC have been presented in Section 4.2.

8.1 Application Domain for Erasure Codes

Besides low storage overhead, erasure coding can provide high data availability and durability, moreover, it could be leveraged to provide fast data access. Hereafter, we show the evolution of the deployment of EC, from archiving systems that ensure the durability of cold data, to in-memory systems that are optimized for low-latency access to hot data.

8.1.1 Erasure codes in archival systems: Higher data durability

Thanks to its cost-efficiency while being able to tolerate several simultaneous failures, EC has been extensively employed in archiving and durable storage systems. Some examples of these systems are presented below.

OceanStore [160] is a peer-to-peer distributed storage system that provides continuous access to persistent information which is usually encoded and stored in an infrastructure composed of untrusted servers. Glacier [108] is a middleware that ensures the long-term durability of the data against large-scale and correlated failures by leveraging EC. Pelican [32] is a storage system that is designed to reduce the total cost of ownership by restricting the number of active machines and encoding its data. Moreover, EC (i.e., with XOR codes) is used in Facebook's f4 storage system to provide durability of the data in geo-distributed infrastructures [195]. Microsoft Giza [52] employs EC to provide catastrophic fault tolerance across geo-distributed data centers.

8.1.2 Erasure codes in storage systems: Lower storage overhead

To reduce the storage cost, mainly for cold data, EC has been integrated into many distributed storage systems. Hereafter, we present some examples of these systems.

Windows Azure Storage [123] is a cloud-based storage system that employs EC alongside replication to store data segments (i.e., collections of files and objects). The segments are scanned periodically and encoded when they are sealed (reached a specified size threshold and become immutable). Facebook's f4 storage system [195] employs EC as a cost-effective alternative to replication to store worm BLOBs. Worm BLOBs are large immutable binary data, written-once read-many, and are not frequently accessed. Baidu's key/value cloud-based storage system, Atlas [165], employs EC to achieve efficient storage for its data as most of the data in their cloud are rarely accessed. Atlas adopts hybrid data protection; while the metadata are 3-way replicated, the data are encoded, providing both space efficiency and access efficiency for metadata. Moreover, many cloud-based object stores and distributed file systems support EC such as Ceph [291], OpenStack Swift [78], Google Colossus (the successor of GFS) [86], QuantcastFS [213], and Liquid [183].

8.1.3 Erasure codes for in-memory caching systems: Fast data access

In two decades, the encoding and decoding speed of EC has increased significantly from 10 MB/s [245] to around 5 GB/s [312] thanks to Intel Intelligent Storage Acceleration Library (ISA-L) [131]. This reduction of CPU overhead, in addition to the reduced storage cost, motives the employment of EC for in-memory storage systems like key/value stores [312, 230, 256, 307].

EC-Cache [230] is an in-memory caching system that employs EC to provide more load balance between the servers compared to selective replication while reducing the storage overhead. This is done by using late binding technique (requesting $n + 1$ chunks and recover the data from the first n chunks that arrive), however, additional bandwidth is required. Cocytus [312] is an in-memory key/value store that is designed to handle small objects ranging from 1 KB to 16 KB. Cocytus employs a hybrid-encoding: by encoding the value part while it replicates the key, metadata, and small-sized values. Moreover, EC has been used with Remote Direct Memory Access (RDMA)-based key/value stores [256]. It has been shown that the proposed design can outperform synchronous RDMA-based replication for large key/value sizes ($>$ 16 KB).

8.1.4 Discussion

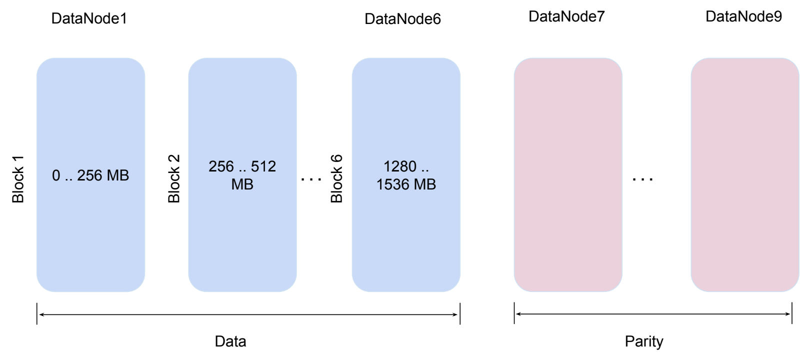
Erasure coding has been successfully applied in practice, from archiving to caching, to provide reliable and efficient data access. Moreover, it has been recently introduced in HDFS which is the primary storage backend for data analytics frameworks. This integration facilitates the execution of data-intensive applications under erasure-coded data which can greatly reduce the storage cost for data analytics, especially for large datasets.

8.2 Data Layout in Erasure Codes

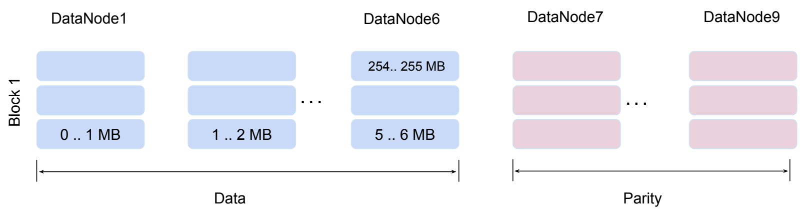
In general, storage systems use different ways to map logical blocks to physical ones. In this section, we present the two representations (i.e., data layouts) that are used to map erasure-coded data blocks.

The mapping between an original (logical) data block and its physical representation can be either *contiguous* or *striped*. This mapping does not impact the availability of the data, however, it greatly influences the access performance, the computation overhead, and the storage overhead (for small data blocks).

Figure 8.1 depicts the representation of 6 blocks with a *contiguous* layout and one block with a *striped* layout under $RS(6, 3)$. Under the contiguous layout, each physical block represents an original block, and the parity blocks are computed from a set of data blocks (6 in this example). While under the striped layout, one block is represented physically by multiple chunks (6 in this example), and the parity chunks are computed solely from the data chunks of that block.



(a) Contiguous block layout



(b) Striped block layout

Figure 8.1: Illustration of EC with contiguous and striped layouts (adopted from [316]).

8.3 Erasure Codes in HDFS

Facebook has implemented the first prototype that integrates EC with HDFS under the name of HDFS-RAID [111, 82]. Blocks are initially stored as replicated, while periodic MapReduce jobs are launched to scan all the blocks in the system and erasure code cold blocks (i.e., blocks that have been created for over one hour) asynchronously. This is done by reducing the replication factor of the data blocks to one and creating the extra parity blocks. However, EC was officially integrated into HDFS in 2018, in the third major release of Hadoop ecosystem [110]. In this thesis, we study EC in the context of HDFS as its the de-facto storage backend for data analytics. Hereafter, we describe the block layout, the concept of EC groups, and the storage policies in HDFS.

Data block layout

Previous efforts to adopt EC in HDFS have implemented EC with contiguous layout [82, 315, 252, 172], however, in the first official release of EC in HDFS, Hadoop community has adopted the *striped* layout. The main reasons for favoring EC with striped layout are: First, it is more efficient for small files [110] which are prominent in data-intensive clusters [82, 201]. For example, in case of $RS(6,3)$, a file with a size of one HDFS block incurs a 300% storage overhead [316], under contiguous block layout, as three parity blocks are still needed. Though aggregating these small files might reduce the problem [82, 315] but it results in more management overhead of these groups of files as files being updated and deleted. Second, encoding and decoding require less memory overhead (see Figure 8.1); for contiguous layout, the complete n blocks should be available in the machine's main memory for encoding and decoding (e.g., $9 * 256$ MB should be available in memory at the same time for encoding and decoding when the HDFS block size is 256 MB), while these operations are performed on the cell level with a striped layout, thus only $9 * 1$ MB (with 1 MB cell size) is required in the memory for encoding and decoding [316]. And finally, the striped layout allows parallel I/O, while reading data under contiguous layout is performed sequentially. Nevertheless, currently, there is a work in progress to design EC with contiguous block layout in HDFS[1].

In the remaining of this thesis, when we mention HDFS with EC, we mean the implementation of EC in version 3 of HDFS i.e., EC with striped block layout.

EC groups

Storing a block in HDFS under EC imposes higher metadata overhead at the NameNode (NN) than under replication (i.e., more memory is needed to maintain the same number of logical blocks). This is because a block is distributed to $n+k$ nodes while it is replicated on 3 nodes only under replication. Therefore, to reduce the metadata overhead every n blocks – belonging to the same file – are grouped into an *EC group*. Accordingly, all the blocks of the same EC group are placed on the same set of nodes.

Figure 8.2 depicts the representation of an EC group with $RS(6, 3)$ scheme in HDFS. To host an EC group, $n + k$ datanodes are needed; n nodes to host the data chunks and k nodes to host the parity chunks. Each (original) block is represented physically by n data chunks and k parity chunks distributed on different datanodes. A data (parity) block represents n data (parity) chunks that belong to different original blocks from the same EC group that are hosted on the same machine. A stripe is the collection of $n + k$ cells that are encoded together.

Storage policies

Since EC has been introduced to HDFS, each file and directory is associated with a storage policy. Apart from replication which has a special policy (i.e., "REPLICATION"), an EC policy is defined by a *scheme* and the *size of striping cell*. The scheme consists of the number of data and parity chunks alongside the codec algorithm (XOR and Reed-Solomon are supported). The striping cell size determines the granularity of data access and buffer size (1 MB by default). Increasing the cell size could limit the overlapping between computation (i.e., encoding) and network transfer on the client side [216], while smaller cell size results in inefficient access to the disk. The default policy under EC is "RS-6-3-1024k", which means that Reed-Solomon (RS) codes are used with 6 data chunks and 3 parity chunks and the encoding/decoding operations are performed with a striping cell of 1 MB.

[1]WIP `https://issues.apache.org/jira/browse/HDFS-8030`

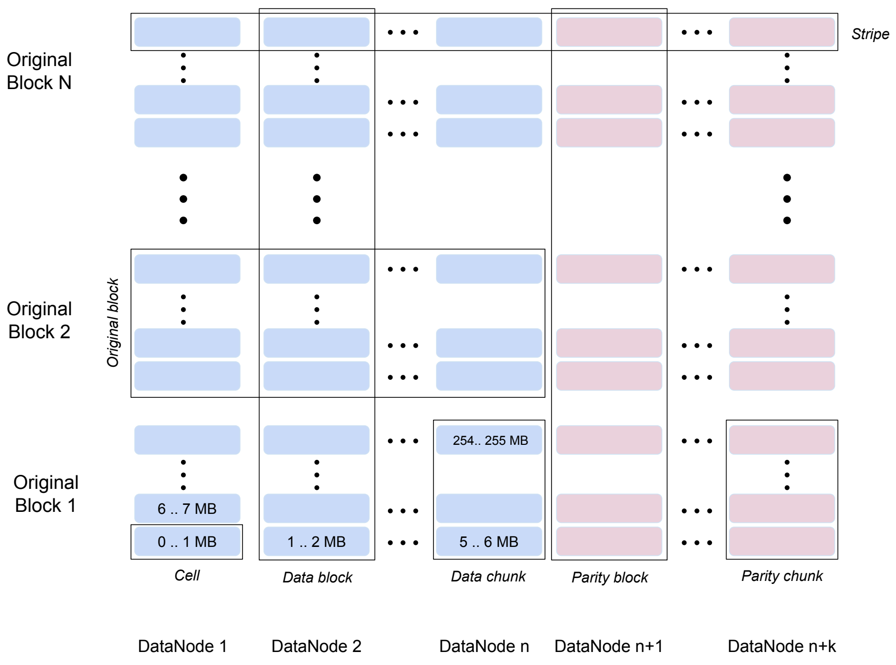


Figure 8.2: EC group representation in HDFS under $RS(n, k)$.

8.4 Data Processing under EC

8.4.1 Job execution under EC

The storage policy of data (i.e., replication or EC) impacts the jobs execution time when reading the data by the map tasks and when writing the data by the reduce tasks.

Under replication. Hadoop schedulers are designed to maximize the locality of map tasks [66, 128, 309]. Replication can improve the percentage of local tasks: if a machine holding the data block is not available, the task can be scheduled on a different machine holding another replica. Furthermore, replication increases the locality of speculative and recovery tasks [305, 319].

Under EC with contiguous layout. Running map tasks under EC with the contiguous layout is equivalent to having only one replica as the parity blocks cannot be used to perform any processing. This might lead to lower data locality especially for speculative and recovery tasks. Moreover, recovery requires rebuilding the complete data block before performing the processing [173].

Under EC with striped layout. Data locality is not fully exposed under EC with the striped layout as blocks are distributed on multiple nodes, therefore, apart from the local data chunk if available, all the remaining data chunks are read from remote nodes.

8.4.2 State-of-the-art: Data processing under EC

Hadoop is (still) a prominent framework for running data-intensive applications in clouds [285]. Substantial research efforts have therefore been dedicated to improve the performance of data processing in Hadoop through locality execution [128, 309] and skew handling [129, 164] or to improve the efficiency of data processing by mitigating stragglers [242, 220] and handling failures [305, 70]. However, all these works rely on the expensive replication.

Few works have investigated and optimized data processing under EC to provide efficient data processing. Zhang et al. [315] try to investigate the impact of EC in Hadoop clusters. They implement EC with a contiguous layout on top of HDFS on the critical path (i.e., encoding is performed online). They show that the execution times of MapReduce applications can be reduced when intermediate data and output data are encoded compared to 3-ways replication. This is due to the reduction in the amount of data which is written to disk and transferred through the network.

Non-systematic codes such as Carousel [171] and Galloper [172] codes have been introduced to improve data locality under EC. Instead of distributing the original data chunks and parity chunks on distinct nodes (nodes which host parity chunks cannot execute map tasks locally), parity data are appended to original blocks and thus all the nodes host both original data and parity data. However, this requires launching more map tasks to process the same amount of data.

To improve the performance of MapReduce applications under failure, Runhui et al. [173] propose degraded-first scheduling. They focus on task scheduling in case of failures for map tasks that may require degraded reads for their input data (i.e., read is degraded when the required block is not available and thus should be reconstructed on the fly by retrieving other data and parity blocks). This blocks the execution of map tasks and prolongs the execution time of MapReduce applications. To address this issue, degraded tasks are scheduled earlier when network resources are not fully used.

Finally, EC-Shuffle [303] proposes to encode the intermediate data of MapReduce like jobs (e.g., Spark). The goal is to provide faster recovery after failures especially for multi-stage jobs, however, this comes at the cost of extra storage cost and network traffic.

8.5 Discussion: Enabling Efficient Data Processing under EC

Big Data applications rely on analytics frameworks to process the constantly growing amount of data. These data need to be stored in distributed storage systems to ensure reliable and efficient data access. However, ensuring data availability by employing replication is becoming expensive. Therefore, sustaining large-scale data processing stresses the need for cost-efficient data analytics. With the continuous reduction of its computation overhead when encoding and decoding data, EC manifests as an alternative to replication that provides lower storage overhead with the same fault-tolerance guarantee.

Previous work that integrates EC into HDFS mainly used the contiguous layout as it preserves data locality, however, EC has been implemented with a striped block layout in last major release of HDFS (the primary storage system for data analytics). Towards understanding the complete picture of the performance of data-intensive applications under

EC, in Chapter 9, we focus on HDFS and evaluate its performance under EC. Chapter 10 completes the preceding chapter by focusing on the performance of MapReduce jobs. While Chapter 11 presents an EC-aware placement algorithm in HDFS that can improve the performance of data-intensive applications under EC.

Chapter 9

Understanding the Performance of Erasure Coding in Distributed Storage Systems

Contents

Erasure coding (EC) is an ideal candidate technique to enable high data availability with low storage cost in data-intensive clusters. However, using EC may raise several challenges, mainly related to the performance and network overhead. In an attempt to understand how practical is EC when employed in distributed storage systems that are used to serve data-intensive applications (e.g., read and write applications, MapReduce applications, etc.), in this chapter, we study the performance of accessing data in a representative distributed storage system (i.e., HDFS). Through extensive micro-benchmarks on top of Grid'5000 [105] testbed, we evaluate the performance of (concurrent) data accesses under erasure coding and replication in a complementary and contrast approach.

The remainder of this chapter is organized as follows. Section 9.1 presents the motivation of this study. The experimental methodology is explained in Section 9.2. Section 9.3 presents the experiments under write operation while Section 9.4 presents the different sets of experiments under read operation. Finally, Section 9.5 concludes this chapter.

9.1 Motivation

As a first step towards understanding the performance of EC for data-intensive applications, we investigate the performance of data access in HDFS under EC. HDFS acts as backend storage for analytics frameworks in data-intensive clusters. In this environment, critical operations that influence the performance of the storage systems are adding and reading data.

Write under EC

Why studying the write performance under EC? Data is generated at an extreme rate. To benefit from this data, it is usually stored – and then analyzed – in data-intensive clusters. Recent studies revealed that hundreds of terabytes of data are ingested every day in data-intensive clusters [53]. Those data, known as data inputs, vary significantly in their sizes. For example, traces from production Hadoop cluster [55] reveal that the size of input data varies from 3 GB to 13 TB. Furthermore, these input data are populated frequently by multiple concurrent users and applications. For example, one computation requires 150 pipelined jobs to complete [70]; hence, the output of a job is used as an input for the later one. This motivates us to study the cost of adding data under erasure coding compared to replication. We further study the impact of concurrency on the performances of both erasure coding and replication.
What is the impact of EC on write performance? Writes are pipelined under replication (i.e., the client sends the data block to the first node which, in turn, pipelines it to the second and then the third). However, under EC, the client encodes the data and then writes the data and parity chunks directly to all the nodes in parallel. This parallelism can provide higher throughput but results in more data sent by the client stressing the link between the clients and the HDFS cluster.

Read under EC

Why studying the read performance under EC? Reading input data is the first step for any data-intensive application. Many studies have discussed the importance of optimizing the read step (i.e., reading input data) to improve the job execution time – mainly through targeting high data locality [51, 309, 128]. Therefore, it is important to understand read performance under EC. Typically, multiple jobs are running concurrently (to achieve higher cluster utilization) therefore we need to evaluate concurrent file read. Also, it is possible that multiple jobs (could belong to different users) use the same files as input, thus we study concurrent reads of the same file under both storage policies. **What is the impact of EC on read performance?** Clients reading data under replication can contact any replica and retrieve all the data from only one replica, this can provide load balance between the nodes hosting the replicas under concurrent read. On the other hand, to read a block under EC, the client needs to contact the data nodes and read the data block in parallel. This might result in higher read throughput if the network bandwidth is higher than disk read bandwidth, but may cause an imbalance in the load as nodes hosting parity chunks may serve fewer requests.

To better understand the performance of write and read operations under EC, we experimentally study their behaviors in detail as described in the following sections.

9.2 Methodology Overview

We conducted a set of experiments to assess the impact of data access pattern and concurrent data access when HDFS operates under replication (REP) and erasure coding (EC).

9.2.1 Platform

We have performed our experiments on top of HDFS. However, it is important to mention that our findings are not tight to HDFS and could be valid in other distributed file systems that implement a striped layout erasure coding policy.

9.2.2 Testbed

Our experiments were conducted on the French scientific testbed Grid'5000 [105] at the site of Nantes. Two clusters – Econome and Ecotype – are used for the experiments with 21 and 40 machines, respectively. Each machine of Econome cluster is equipped with two Intel Xeon E5-2660 8-core processors, 64 GB of main memory, and one disk drive (HDD) at 7.2k RPM with 1 TB. While Ecotype cluster's machines have the same CPU and memory but each machine is equipped with an SSD of 350 GB. The machines of each cluster are connected by 10 Gbps Ethernet network. The two TORs switches of both clusters are connected with four 40 Gbps links. The machines run 64-bit Debian stretch Linux with Java 8 and Hadoop 3.0.0 installed. All the experiments have been done in isolation on the testbed, with no interference originated from other users.

Econome runs HDFS (one node hosts the NameNode (NN) and the remaining 20 nodes act as DataNodes (DNs)) while Ecotype hosts the clients. To exclude the impact of the local disks at the client side, we store the dataset in the main memory.

9.2.3 HDFS configuration

HDFS block size is set to 256 MB (similar to [218, 70, 305]) and the replication factor is set to 3 (default value). For EC, if not otherwise stated, we use the default EC policy in HDFS, i.e., $RS(6,3)$ scheme with a cell size of 1 MB.

9.2.4 Benchmarks

We used the read and write operations to measure the performance of HDFS. These operations are issued from the clients using the HDFS commands `hadoop fs -get` and `hadoop fs -put` respectively. The test files are generated by the `dd` command from `/dev/urandom` as an input source.

9.2.5 Metrics

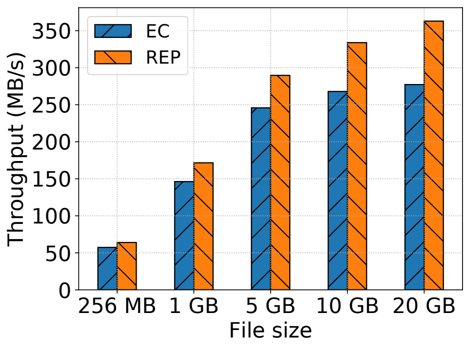
The throughput at the client side is the main metric used to measure the performance of read/write operations from/to HDFS. For one client, it is the amount of data read/written per second. In the case of concurrent read and write (by multiple clients), the average throughput per client is shown alongside the standard deviation of the clients' throughput. We use the *coefficient of variation* metric (i.e., the standard deviation divided by the mean) to measure the variation in the data and load (i.e., read and write) in-between DNs.

We also profile CPU utilization, memory utilization, disk and network I/O of the nodes using the python library `psutil` [225] version 5.4.8.

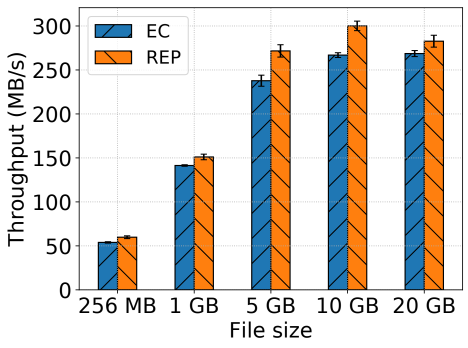
9.3 Cost of Adding New Data

9.3.1 Write dataflow in HDFS

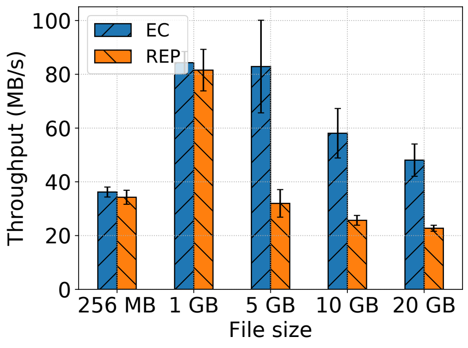
Regardless of the HDFS storage policy, the client splits the file into blocks equal to the HDFS configured block size. Under replication, and for each block, the client contacts the NN to obtain a list of DNs equal to the replication factor (e.g., 3 nodes for 3-way replication) to write the block to. The client streams each block to the first DN in the corresponding list which in turn pipelines the data to the second one and so on. On the other hand, the write dataflow under EC is a bit different; for $RS(6,3)$, after splitting the file into blocks, every 6 blocks are grouped into an *EC group*. As described in Section 8.3, the reason behind grouping blocks into EC groups is to reduce the metadata overhead at the NN. As a result, all the blocks of the same EC group are placed on the same set of nodes. For each EC group, the client contacts the NN to obtain a list of 9 DNs. To encode this group of blocks, the blocks are encoded sequentially; each block is split into cells (e.g., 1 MB), then the client encodes every 6 cells to generate 3 parity cells. The client sends these 9 cells to the 9 DNs, and then continues to do the same with the remaining cells, and repeats the same process for the remaining blocks in the group. Then, do the same for the remaining EC groups. In conclusion, three (number of replicas) DNs are contributing to a write operation at a time under REP, while 9 DNs $(n + k)$ are simultaneously writing data under EC. It is important to mention that blocks are



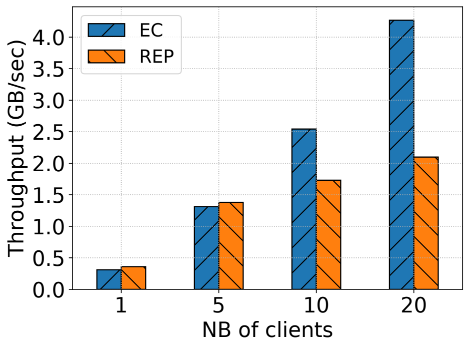
(a) Write one file per client



(b) Write 5 files by 5 clients



(c) Write 40 files by 40 clients



(d) Write 100 GB dataset (aggregated throughput)

Figure 9.1: Writing under REP and EC.

written sequentially to HDFS (the second block is sent once the first block is completely stored in a DN, persisted in disk or buffered in the memory). Moreover, when the data is stored completely in all DNs (at least buffered in the memory), the write operation is considered successful.

9.3.2 Results of single write

Figure 9.1a shows the write throughput of one client with different file sizes. The write throughput is 1.11x to 1.3x higher under REP compared to EC when increasing the file sizes from 256 MB to 20 GB, respectively.

When writing 20 GB file, REP achieves 363 MB/s write throughput, while EC obtains a throughput of 277 MB/s which is 76% of the throughput under REP. This can be explained due to the larger amount of data which is sent from the client to HDFS under EC (i.e., 30 GB accounts for the original and parity data) compared to REP (i.e., 20 GB) and the "low cost" data pipelining under REP. First, data stays in the buffers of the DNs when received from the client or pipelined from other DNs, hence, it is not synchronized to disks directly. As a result, data transfer time – to HDFS – strongly depends on the amount of data sent to the cluster. Given that the throughputs of inter-cluster traffic (network traffic between the clients and the HDFS cluster) are 444 MB/s and 399 MB/s under EC and REP, respectively, the file under EC takes 35% more time to be transferred

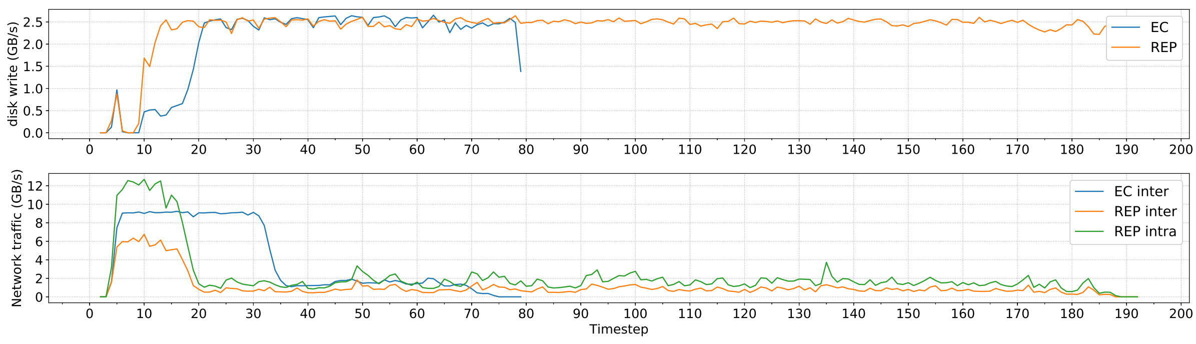


Figure 9.2: Disk, inter-network and intra-network throughput for concurrent writing of 5 GB files by 40 clients.

to HDFS compared to REP (the difference in transfer times between EC and REP is clearer when increasing the file size). Second, (buffered) data can be pipelined as soon as they reach the first DN because the intra-cluster (data transferred inside the HDFS cluster) throughput is almost two times the inter-cluster throughput (i.e., 801 MB/s), thus replicating (transferring) data inside the cluster does not introduce an extra cost. Finally, more data is persisted to disk under REP compared to EC (i.e., 21 GB under EC while it is 29 GB under REP); which in turn slightly slow the write operation under REP.

Note that, by default, Hadoop client utilizes only one thread to send the data and therefore the inter-cluster throughput may not be fully utilized. In case of REP, data is communicated between two nodes only and thus the inter-cluster throughput is limited to 399 MB/s. In case of EC, despite that the client is sending data to 9 DNs, inter-cluster throughput is limited to 444 MB/s because only one core is responsible for encoding and sending data (in case of 20 GB, we observe that one core is always at 100% utilization).

`Observation 1.` Unlike replication, more data (i.e., parity data) goes from the client to HDFS cluster under EC. This extra parity data (0.5x of the original data in our configuration) results in a loss of throughput compared to replication as data are pipelined (replicated) inside the cluster with minimal cost.

9.3.3 Results of concurrent writes

When increasing the number of concurrent clients to 5, EC starts to approach the throughput of REP (Figure 9.1b). Interestingly, with 40 clients, EC has a throughput 2 times that of REP for files equal and bigger than 5 GB as depicted in Figure 9.1c.

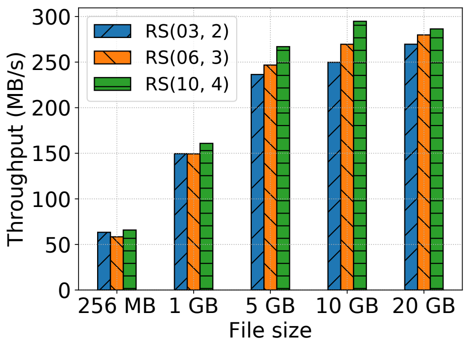
In case of writing 5 GB files by 40 concurrent clients, REP achieves 32 MB/s write throughput, while EC obtains a throughput of 83 MB/s which is 2.59x higher than the throughput under REP. Disks are saturated under both EC and REP (the average disk throughput is 110 MB/s and 124 MB/s, respectively). As more data is persisted to disks under REP compared to EC (i.e., 464.4 GB under REP while it is 160.5 GB under EC), writing data to disks under EC is almost 2.5 times faster. On the other hand, the throughput of the intra-cluster network is low (i.e., 183 MB/s to 233 MB/s when the files are bigger or equal to 5 GB), thus, the write performance further degrades under

REP. The low disk and intra-cluster performances limit the arrival rate of data to HDFS cluster under REP (the arrival rate is 1162 MB/s under REP compared to 4535 MB/s under EC). As shown in Figure 9.2, the network performance (inter and intra) drops once the buffers are filled and data starts to be synchronized to disks while still not completely replicated (transferred) to other DNs.

To further explain the impact of concurrency on the performance of write under both EC and REP, Figure 9.1d shows the aggregated write throughput when writing a data set of 100 GB while varying the number of clients and therefore the files sizes. When the number of concurrent clients is 5, REP achieves a throughput of 1410 MB/s while it is 1340 MB/s under EC. The aggregated throughputs increase by 1.94x and 3.25x when increasing the number of clients to 10 and 20 clients under EC, respectively. This increase is mainly due to the increase in the inter-cluster throughputs; On the other hand, the aggregated throughputs increase 1.2x and 1.5x under replication when increasing the number of clients to 10 and 20 clients, respectively. Here, the high cost of data pipelining limits the scalability of HDFS under REP when increasing the number of clients. Specifically, buffers under replication are filled faster (original data and replicated data) compared to EC. Thus, the arrival rate of data to a DN is limited by the disk throughput (125 MB/s under EC and 128 MB/s under REP). However, a DN under replication is receiving data from the clients and from other DNs, this reduces the inter-cluster throughput and prolong the writing time under REP compared to EC.

`Observation 2.` Under concurrent writes, in contrast to under EC, the performance of write workloads under replication is limited by the intra-cluster transfer and disk contention (as the total amount of written data is larger under REP compared to EC). Hence, due to the high network and memory cost of data pipelining, EC can even outperform replication – under heavy concurrent writes. This gap is more obvious for large files (e.g., in our experiments, the throughput under EC is at least 2x the throughput achieved under replication when 20 clients are writing 10 GB file each (Figure 9.1c).

9.3.4 Impact of EC policies



(a) Distinct RS schemes with 1 MB cell size

Figure 9.3: The impact of EC policies on single client write.

The number of data and parity chunks determines the fault tolerance degree and the storage overhead of the erasure codes. However, here we compare the following three schemes solely from a performance point of view: $RS(6,3)$, $RS(10,4)$ and $RS(3,2)$.

Figure 9.3a shows the write throughput of one client with different EC schemes. For 10 GB file, $RS(10,4)$ achieves a write throughput of 295 MB/s while $RS(6,3)$ achieves 270 MB/s followed by $RS(3,2)$ with 250 MB/s. The throughput per scheme is strongly related to the total amount of data written; For $RS(10,4)$, the parity data accounts for 40% of the original data size, while it accounts for 50% and 67% for $RS(6,3)$ and $RS(3,2)$, respectively. Even though the inter-cluster throughput is higher for $RS(3,2)$ and $RS(6,3)$, but the fact that sending fewer data compensates for the difference. For example, the throughputs of inter-cluster traffic are 435 MB/s, 444 MB/s, and 471 MB/s for $RS(10,4)$, $RS(6,3)$, and $RS(3,2)$, respectively. The reason behind higher inter-cluster throughput is because fewer nodes are involved, therefore, less probability to wait for the slowest response. The disk throughputs are 481.5 MB/s, 321.4 MB/s, and 389.8 MB/s for $RS(10,4)$, $RS(6,3)$, and $RS(3,2)$, respectively.

`Observation 3.` The performance of write operation by a single client depends on the total amount of data transferred to the HDFS cluster, therefore, EC schemes with less parity overhead have higher throughput.

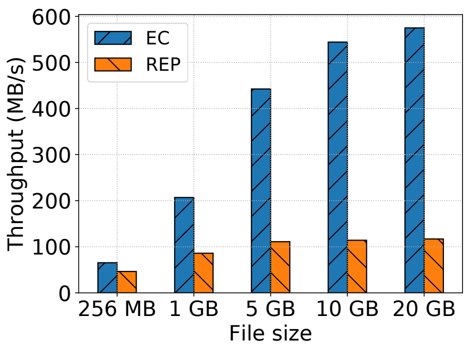
9.3.5 Implications

Replication is not only costly in terms of storage requirements but also contributes to the problems of oversubscribed networks [70] and poor disk performance [70] in production clusters (e.g., traces from production clusters at Facebook and Microsoft pointed out that replication results in almost half of all cross-rack traffic [53]). Applying EC as an alternative to replication does not only reduce storage cost and disk overhead but also results in lower network traffic (up to 50%). In conclusion, given its performance, EC is feasible for write applications, more importantly, EC sustains high throughput for write operations under high concurrency.

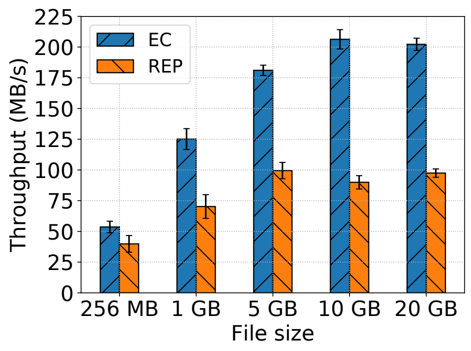
9.4 Reading Data under EC

9.4.1 Read dataflow in HDFS

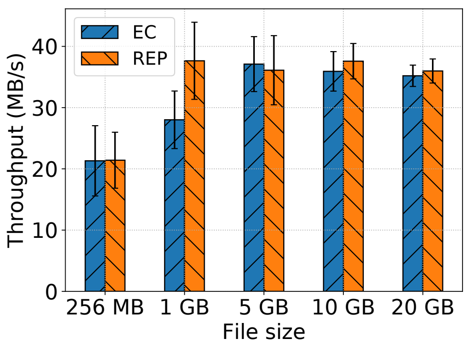
To read a file in HDFS, the client contacts the NN to obtain the addresses of the DNs hosting each block of the file. For each block, the DNs are ordered by the network distance to the client and are randomized for the nodes that have the same distance. The client then reads the blocks in order i.e., sequentially, one by one. Reading one block of data (without any failure) has the same cost under both EC and REP, as the same amount of data (equal to the block size) is read from disk and transferred over the network. However, under EC the data block can be read in parallel from multiple DNs (6 DNs in our configuration). This parallelization can achieve better performance especially when the network bandwidth is higher than the disk bandwidth [16].



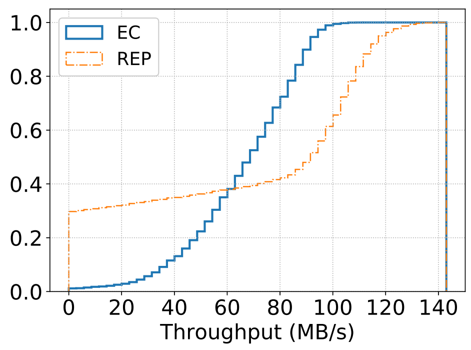
(a) Read one file per client



(b) Read 5 distinct files by 5 clients



(c) Read 40 distinct files by 40 clients



(d) CDF of disk throughput per DN during the concurrent read of 40 files by 40 clients each of 20 GB

Figure 9.4: Reading distinct files under REP and EC.

9.4.2 Results of single read

The read throughput for different file sizes by one client is depicted in Figure 9.4a. As we expect, EC performs better than REP. Specifically, the read throughput is 1.4x to 4.95x higher under EC compared to REP when increasing the file size from 256 MB to 20 GB, respectively.

When reading 20 GB file, REP achieves 116 MB/s read throughput, while EC obtains a throughput of 575 MB/s which is 4.95x higher than the throughput under REP. The reason behind this is that an EC block is read from 6 nodes, while under REP, a block is read from a single node. Therefore – as the performance of the read operation is limited by the disk – 6 disks are leveraged in parallel under EC, while a single disk is used under REP. Hence, the throughput of the read workload is strongly co-related with the performance of the disks. Therefore, the performance gap between EC and REP is clearer when increasing the file sizes, especially when the disk throughput under EC increases.

Under replication, a block is read sequentially from the disk, hence, the disk is exploited efficiently (i.e., the throughput of the disk is between 85 MB/s and 100 MB/s most of the time). On the other hand, under EC, a block is read in parallel from multiple nodes (6 nodes in our configuration), therefore, each node reads a fraction of the block (one data chunk of 43 MB) and thus cannot leverage the full throughput of the disk.

Table 9.1: Disk read throughput (MB/s) of active DNs in case of single client read.

	File size	mean	25%	50%	75%	max
EC	**256 MB**	35.42	26.62	42.00	42.75	43.00
	1 GB	71.28	57.00	71.25	85.50	86.00
	5 GB	74.59	65.51	85.33	85.51	85.67
	10 GB	83.76	85.35	85.45	85.61	102.80
	20 GB	83.39	77.44	85.39	86.85	102.80
REP	**256 MB**	85.33	85.33	85.33	85.33	85.33
	1 GB	85.42	85.33	85.33	85.42	85.67
	5 GB	85.19	85.46	85.67	85.67	102.40
	10 GB	88.44	85.33	85.55	89.40	102.80
	20 GB	86.63	83.19	85.50	92.86	102.60

As we can see in Table 9.1, when the size of the file is 256 MB, the disk throughput is limited by the physical block size (43 MB/s) under EC. However, when the client reads all the blocks of an EC group (6 blocks in our configuration), the disk throughout of an individual node is fully utilized as 6 chunks – belonging to different blocks – are read sequentially from the disk. For instance, when the file size is 20 GB, the average disk throughputs (of active disks) for both EC and replication are 83.4 MB/s and 86.6 MB/s (the disk throughput is higher than 80 MB/s most of the time), respectively. Hence, this explains the 4.95x performance difference.

`Observation 4.` Compared to 3-way replication, a block (256 MB) is scattered on more DNs under EC. Therefore, reading a file under EC has a clear advantage compared to under REP, as multiple disks are leveraged in parallel.

9.4.3 Results of concurrent reads

Reading distinct files

When increasing the number of concurrent clients to 5 – each reading a distinct file, the gap between EC and REP starts to decline as shown in Figure 9.4b. Interestingly, with 40 concurrent clients, REP achieves slightly higher throughput (clearer when the file size is 1 GB) as shown in Figure 9.4c. Moreover, we notice that the performance of read operations exhibits noticeable variation in-between different clients under both REP and EC.

When 5 clients are reading 5 files, each has a size of 20 GB, REP achieves 97 MB/s read throughput, while EC obtains a throughput of 202 MB/s which is 2.08x higher than that under REP. The average disk throughput of active DNs is 60 MB/s and 86 MB/s under EC and REP, respectively. The average disk throughput under EC decreases – compared to the one in case of 1 client (i.e., 83.4 MB/s) – which is due to data skew. As shown in Figure 9.5, although most of the DNs are serving data, some of them (around 6 DNs) serve more clients a time which in turn results in exploiting the full capacity of the disk (i.e., 125 MB/s) on those nodes. On the other hand, the majority of nodes were serving 1 client most of the time, thus low utilization of the disk. We observe that

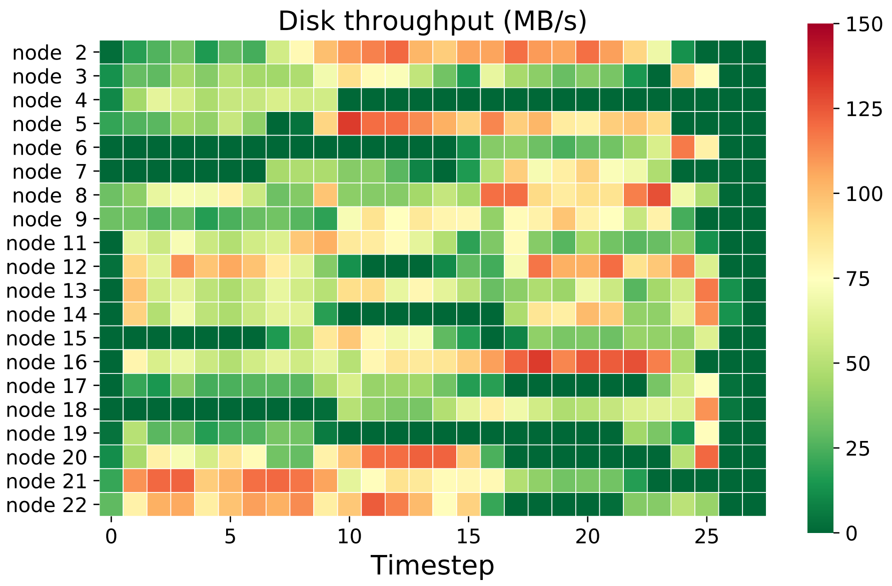


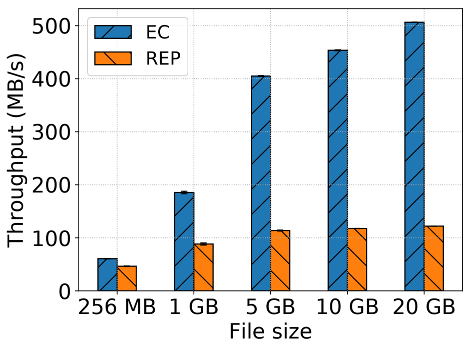
Figure 9.5: Disk throughput of the DNs during the concurrent read of 5 files by 5 clients each of 5 GB under EC.

disk throughputs are above 100 MB/s during 50% and 18% of the read workload time under REP and EC, respectively. Hence, this, in addition to the increase in the number of contributing nodes (active disks), explains the decrease in the performance difference from 4.95x to 2.08x between EC and REP.

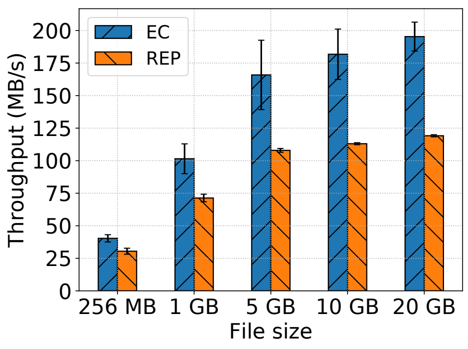
When 40 clients are reading 40 files, each has a size of 20 GB, we observe that HDFS suffers from obvious load imbalance under both REP and EC. Under REP, at least one disk was not active during 25% of the read workload (see Figure 9.4d). Under EC, some DNs still serve more clients compared to other DNs which in turn results in high contention on the disks and thus low disk throughputs; disk throughputs are above 100 MB/s during 40% and 2% under REP and EC, respectively. As a result, the aggregated disk throughout under EC is 1352 MB/s and REP is 1350 MB/s, which explains the 2% difference. In addition, this load imbalance results in high variation in the latency of the read workload in-between clients, therefore, clients which are served by nodes under heavy load need more time to read their files.

Reading the same file

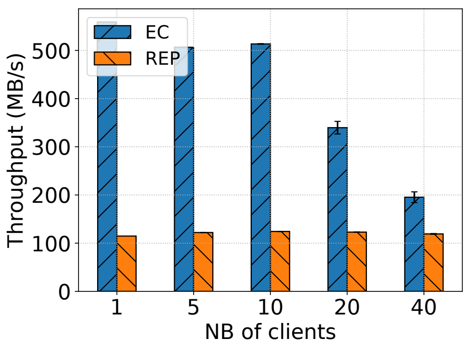
Figure 9.6a shows the average read throughput per client when 5 clients are reading the same file concurrently. With 20 GB file, the average read throughput is 506 MB/s under EC while it is 122 MB/s under REP, thus 4.14x faster under EC. To explain the difference, we inspect the amount of read data and sent data at the cluster level. Under EC, the cluster has an aggregated disk throughput of 537.6 MB/s and network throughput of 2840 MB/s: each block under EC is read once from the same nodes, cached and then sent to the 5 clients; hence, network throughput is 5.2x higher than the disk throughput. However, under REP, the cluster disk throughput is 322.5 MB/s and the network has a throughput of 652.8 MB/s, 1.7x higher than the disk. Here, a block is served by the



(a) Read by 5 clients



(b) Read by 40 clients



(c) Read 20 GB file

Figure 9.6: Reading the same file under REP and EC.

3 DNs at the same time: as a block is available on 3 DNs, the DNs receive requests – from different clients – to read the block, accordingly, the 3 DNs read the data from the disk, cache it and then sent it to 5 clients (one DN serves 1 or 2 clients). This explains why the total amount of read bytes is 52.1 GB (2.6x the size of the file) under REP and 20 GB under EC.

When increasing the number of concurrent clients to 40, we notice that the clients maintain their average throughput under REP (119.2 MB/s) while it drops to 195.3 MB/s under EC, as shown in Figure 9.6b. Under REP, each block is served by 3 different DNs, so a block will be served by a new set of DNs (3 DNs) while the previous block is still being served to clients (new block is requested once a client out of the 40 clients finishes fetching the current block). Consequently, disk throughput increases under REP from 322 MB/s with 5 clients to 367.6 MB/s and thus the network throughput increases to 5079 MB/s. On the other hand, 6 DNs are continuously serving the 40 clients: each DN is responsible for one data chunk. This saturates the network bandwidth of the DNs and cause variations in-between clients. Moreover, as the reading of a new block can start when all the chunks are received by at least one client, the same 6 DNs will be serving two blocks (belong to the same EC group) simultaneously and this will reduce the number of disk read requests for the new block and at the same time reduce the amount of data sent to clients through network and further increase the variation. Even worse, the performance degradation at the disk-level and the network-level (and the resulted

Table 9.2: Average aggregated cluster network and disk throughput (MB/s) in case of concurrent read of the same 20 GB file by the number of clients.

Number of clients	**Network throughput (MB/s)**		**Disk throughput (MB/s)**	
	EC	**REP**	**EC**	**REP**
5	2840.3	652.8	537.6	322.5
10	5836.8	1331.5	552.8	374.2
20	7009.7	2648.2	384.1	376.1
40	8048.7	5079.1	229.2	367.6

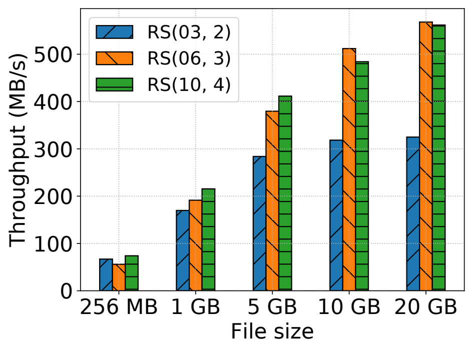
variation in the throughput in-between clients) is amplified when DNs are serving two successive EC groups. We observe that the number of disk requests drops by almost 50% (from 50 to 24) when increasing the number of clients from 5 to 40 clients. Consequently, the disk throughput drops from 537.6 MB/s to 229.2 MB/s – the disk throughputs of active DNs are less than 50 MB/s for almost 80% of the read workload time. This, in addition to the contention at the network due to the high concurrency, results in a network throughput of only 8048 MB/s and a variation of 5.7% in the throughput in-between clients.

Figure 9.6c depicts the read throughput of the same file by increasing the number of clients. REP maintains the same throughput per client, however, we can clearly see how the throughput under EC drops with increasing the number of concurrent clients. Table 9.2 shows the cluster-level network and disk throughput. REP maintains a stable disk throughput with increasing the number of clients and relative increase of network throughput with the number of clients. However, under EC, the disk throughput drops while the network throughput increases sub-linearly.

`Observation 5.` The goal of presenting data as EC groups is to reduce metadata overhead in the NN, but it results in a high imbalance in data distribution across DNs compared to replication. This, in turn, causes stragglers when high concurrent reads are performed to either distinct files or the same file, and therefore, it reduces the advance of parallel chunks reads. Specifically, DNs which are serving data continuously to multiple clients exhibit low disk throughput due to high contention on disk (distinct files) or high contention at the network (same file).

9.4.4 Impact of EC policies

Figure 9.7a shows the read throughput of one client with different EC schemes. With 20 GB file, $RS(6,3)$ achieves the highest throughput of 568 MB/s, followed by $RS(10,4)$ with 561.7 MB/s while $RS(3,2)$ achieves 324.8 MB/s. The average (max) disk throughputs are 53.8 (60.2) MB/s, 81.4 (102.6) MB/s, and 83.6 (102.9) MB/s for $RS(3,2)$, $RS(6,3)$, and $RS(10,4)$, respectively. $RS(3,2)$ leverage just 3 disks in parallel compared to 6 and 10, and thus, its limited throughput compared to $RS(6,3)$ and $RS(10,4)$. However, even though $RS(10,4)$ read from 10 disks in parallel while $RS(6,3)$ read from 6 disks, $RS(6,3)$ has slightly higher throughput. The reason is that more parallelism imposes higher overhead and introduces a higher probability of waiting for the slowest node to read and send its data. The same behavior is observed in caching systems [308].



(a) Variaous RS(X,Y)

Figure 9.7: The impact of EC policies on single client read.

`Observation 6.` In contrast to write, the amount of read data is the same with all the schemes. However, the degree of parallelism (which is related to the number of data blocks n) becomes the deciding factor; smaller n means less parallelism, hence less throughput, while large n might result in considerable overhead [308].

9.4.5 Implications

The advantage of striped block layout under EC is to enable parallel data read. However, the use of EC groups (to reduce metadata overhead) reduces this advantage under concurrent clients read. EC groups introduce imbalance in the data access across DNs which, in turn, causes stragglers. Therefore, mitigating stragglers under concurrent access is essential. While *late binding* can be employed (as in EC-Cache [230]), it comes with the cost of extra network transfer. Furthermore, *intelligent* data access is used in EC-Store [3] but this method requires historical analysis of data access. On the other hand, reads under EC benefit more from OS caches than replication as data chunks are always read from the same node, which might be beneficial for iterative applications. In conclusion, the previous experiments demonstrate the feasibility and the advantages of using EC for read operation in large-scale clusters.

9.5 Conclusion

Efficient data access in large-scale storage systems is an important problem with the current rate of data generation. In this chapter, we experimentally evaluate the performance of data read and write operations in HDFS under both replication and erasure coding. Our findings can be summarized as follows:

- Write operations under replication have higher throughput than under EC for single client. However, when increasing the number of concurrent clients, the throughput under EC approaches that under replication and even outperforms it under high concurrency. In addition, applying EC as an alternative to replication does not only reduce storage cost and disk overhead but also results in lower network traffic.

- When reading data, EC can leverage parallel reads from multiple disks and deliver a 4.95x higher throughput than replication for a single client. However, under high concurrency, the performance of EC is impacted by the imbalanced load across nodes, which is caused by the distribution of chunks and the design of EC group.

Briefly, our findings demonstrate that erasure coding is not only feasible but also outperforms replication in many scenarios. In the next chapter, we further study the impact of EC on read and write within the context of MapReduce applications where the clients (i.e., the tasks) reside inside the cluster and are processing the data.

Chapter 10

Characterizing the Performance of Data Analytics under Erasure Coding

Contents

Data processing applications have become first-class citizens in many industrial and scientific clusters [76, 50], in this chapter, we aim to answer a broad question: *How erasure coding performs compared to replication for data processing applications in data-intensive clusters?*

The pervious chapter suggested that read and write operations exhibit promising performances under erasure coding (EC) in data-intensive clusters. However, the performance of data processing under EC is still not clear as there is a clear tradeoff between parallel reads of input data and not being able to fully preserve data locality. According, we provide – to the best of our knowledge – the first study on the impact of EC on the task runtimes and application performances. We conduct experiments to thoroughly understand the performance of data-intensive applications under replication and EC. We use representative benchmarks on the Grid'5000 [105] testbed to evaluate how analytics workloads, data persistency, failures, backend storage devices, and network configurations impact their performances.

The remainder of this chapter is organized as follows. Section 10.1 presents the motivation behind using erasure coding for data processing. The experimental methodology is explained in Section 10.2. Section 10.3 and Section 10.4 present the different sets of experiments highlighting the impact of erasure coding on the performance of MapReduce applications and summarize our observations. Section 10.5 discusses guidelines and new ways to improve data analytics under EC. Finally, Section 10.6 concludes this chapter.

10.1 Motivation

Moving from the traditional replication to EC presents unique opportunities for data processing in terms of storage cost and disk accesses (i.e., less amount is written to disk). The benefits are more obvious when adopting high-speed "expensive" storage devices such as SSDs and DRAMs. More importantly, important progress has been made to mitigate and reduce the impact of the inherited limitations of EC such as encoding overhead and extra network traffic when reading the data inputs. We list them in detail below:

- **CPU overhead of EC:** EC used to have high CPU overhead when encoding and decoding data. Fortunately, thanks to Intel Intelligent Storage Acceleration Library (ISA-L) [131], EC operations are implemented and run at CPU speed (e.g., encoding throughput using only one single core is 5.3 GB/s for Intel Xeon Processor E5-2650 v4 [133]). Moreover, in disk-based storage systems, reading/writing the data is the dominant factor, not the encoding/decoding operations which become negligible compared to disk throughput [151, 3]. Therefore, performing EC operations on the critical path (online) has minimal impacts.

- **Network overhead of EC:** Accessing data under EC requires remote data access and generates network traffic. Therefore, achieving data locality under EC is not possible (apart from the fraction of the block that is read locally). On the other hand, the current advance in data center networks makes the gap between network bandwidth and storage I/O bandwidth rapidly narrowing [92], and therefore, the performance bottleneck is shifting from network to storage I/O [231]. Moreover, previous work has shown that disk-locality becomes irrelevant [16, 143]. This is acknowledged in practice as all PaaS offer of MapReduce (e.g., AWS Elastic MapReduce [25], Azure HDInsight [112], etc.) are implemented as disaggregated computation and storage design.

To demonstrate how (im)practical is EC for data processing, we conduct an extensive experimental study to understand the performance of data-intensive applications under replication and EC. In detail, we aim at answering three main questions:

1. **How different analytical workloads perform under EC?** The wide adoption of clouds and MapReduce introduces many types of data-intensive applications ranging from simple sort applications to the emerging machine learning applications [187]. In this study, we conduct experiments with three representative data-intensive applications: Sort, Wordcount and Kmeans applications. Moreover, we study different execution patterns by enabling and disabling the overlap between map execution and data shuffle.

2. **What is the impact of hardware on the performance of data processing under EC?** We conduct experiments with different hardware configurations: we used HDDs and DRAMs as backend storage devices for HDFS and vary the network bandwidth from 1 Gbps to 10 Gbps. In addition, we conduct experiments to imitate the limited memory capacity by persisting all the output data to disk.

3. **How EC recovery impact the performance of data processing under failures?** Failures are common in data-intensive clusters [98, 88, 106, 136, 96]. Previous works show that number of failures and their occurrence times impact the performance of data-intensive applications significantly [305, 70]. Thus, we conduct experiments while varying the number of failures and the injecting time.

10.2 Methodology Overview

We conducted a set of experiments to assess the impact of analytics workloads, data persistency, failures, the backend storage devices, and the network configuration on the performance of data-intensive applications when HDFS operates under replication (REP) and erasure coding (EC).

10.2.1 Platform

We have performed our experiments on top of Hadoop 3.0.0. We evaluate MapReduce applications in two scenarios: when overlapping map phase and shuffle stage in reduce phase – as in Hadoop and Flink – and when there is no overlapping between the two phases – as in Spark [310, 259]. It is important to note that the work (and the findings) we present here neither is limited to HDFS implementation nor specific to Hadoop and can be applied to other distributed file systems that implement a striped layout erasure coding policy. Moreover, our findings can be valid with other data analytics frameworks if they run on top of HDFS as the impact of EC in limited to reading the input data and writing the output data; not on how the actual computation and task scheduling are performed.

10.2.2 Testbed

Our experiments were conducted on the French scientific testbed Grid'5000 [105] at the site of Nantes. We used for our experiments the Econome cluster, which comprises 21 machines. Each machine is equipped with two Intel Xeon E5-2660 8-core processors, 64 GB of main memory, and one disk drive (HDD) at 7.2k RPM with 1 TB. The machines are connected by 10 Gbps Ethernet network, and run 64-bit Debian stretch Linux with Java 8 and Hadoop 3.0.0 installed. All the experiments have been conducted in isolation on the testbed, with no interference originated from other users.

In all the experiments, one node is dedicated to run the NameNode (NN) and the ResourceManager (RM), while the remaining 20 nodes serve as workers (i.e., DataNodes (DNs) and NodeManagers (NMs)). Network bandwidth of 1 Gbps links is emulated with the Linux Traffic-Control tool [179].

10.2.3 Hadoop configuration

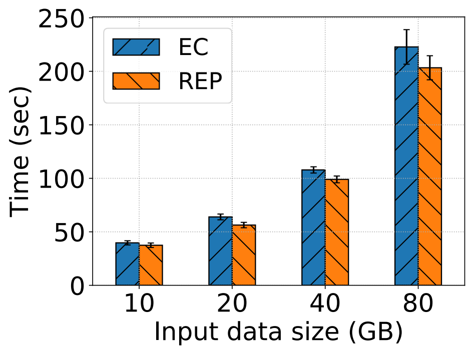
HDFS block size is set to 256 MB and the replication factor is set to 3. For EC, if not otherwise stated, we use the default EC policy in HDFS, i.e., $RS(6,3)$ scheme with a cell size of 1 MB. We disable speculative execution to have more control over the number of launched tasks. We configure YARN (the resource manager) to run 8 containers per node (i.e., one per CPU core). Therefore, 160 slots are available in our cluster, which is sufficient to process 40 GB of data in a single map wave.

10.2.4 Benchmarks

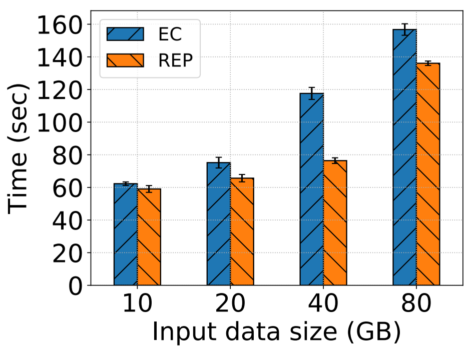
We evaluate the performance of MapReduce with two micro-benchmarks (i.e., Sort and Wordcount applications) and one iterative application (i.e., Kmeans). *Sort* application is considered shuffle intensive and generates an output equal in size to the input, which represents a considerable portion of scientific and production applications (e.g., traces collected at Cloudera show that, on average, 34% of jobs across five customers had output at least as large as their inputs [50]). *Wordcount* application is considered map intensive with small output size, which accounts for the majority of jobs in production data-intensive clusters (e.g., about 70% of the jobs in Facebook clusters [50]). Both Sort and Wordcount applications are available with Hadoop distribution.

In addition to the micro-benchmarks, we evaluated *Kmeans* application from the HiBench suite [124]. *Kmeans* is a basic Machine Learning application that is used to cluster multi-dimensional datasets. We used the provided synthetic dataset generator of HiBench to generate a dataset with 1200M samples of 20 dimensions each, which results in a total size of 222 GB. We set the number of clusters to 5 and set the maximum number of iterations to 10.

Each job is running alone, thus, it has all the resources of the cluster during the execution. We run each experiment 5 times and we report the average alongside the standard deviation. Moreover, when analyzing a single run, we present the results of the job with the median execution time. We cleared the caches between data generation and data processing, as well as between the runs.



(a) Sort application



(b) Wordcount application

Figure 10.1: Job execution time of Sort and Wordcount applications under EC and REP (non-overlapping shuffle).

10.2.5 Metrics

We used the *execution time* and the *amount of exchanged data* for MapReduce applications; Job execution time is the total time of the job from its start time to finish time (not including time waiting in the queue). Exchanged data is the amount of data that goes over the network between the DNs. It consists of non-local read for input data, the shuffled data, and the non-local write of output data. It is measured as the difference between the bytes that go through each machine network interface, for all the DNs, before and after each run. Also, we used the *coefficient of variation* metric (i.e., the standard deviation divided by the mean) to measure the variation in tasks' (map and reduce) runtimes.

During all the experiments, we collect the metrics related to CPU utilization, memory utilization, disk, and network I/O of the DataNodes using the python library `psutil` [225] version 5.4.8.

10.3 Data Processing under EC

A MapReduce job consists of two phases: (1) map phase: map tasks read the input data from HDFS and then write the intermediate data (i.e., the output of the map phase/input for the reduce phase) in the local disks after applying the map function. (2) reduce phase which in turn includes three stages: shuffle stage which can be performed in parallel with map phase, sort stage and finally reduce stage. Sort and reduce stages can only be performed after the map phase is completed. In the reduce stage, data are written to HDFS after applying the reduce function. However, we note here that writing data to HDFS is considered successful when the data is completely buffered in memory. Consequently, a job can be considered as finished *before* all the output data are persisted to disks. When running MapReduce applications, HDFS is accessed during the map phase and the reduce stage, whereas, intermediate data is written to the local file system of the DNs.

Table 10.1: Detailed execution times of Sort and Wordcount applications for 40 GB input size in second with the percentage of each phase.

		Job execution time	**Map Phase**	**Reduce Phase**
Sort	**EC**	113.6	70.6 (64.2%)	38.3 (35.8%)
	REP	103.9	43.4 (43.9%)	54.3 (56.1%)
Wordcount	**EC**	113.7	101.0 (90.8%)	8.2 (09.2%)
	REP	73.8	60.4 (85.0%)	8.4 (15.0%)

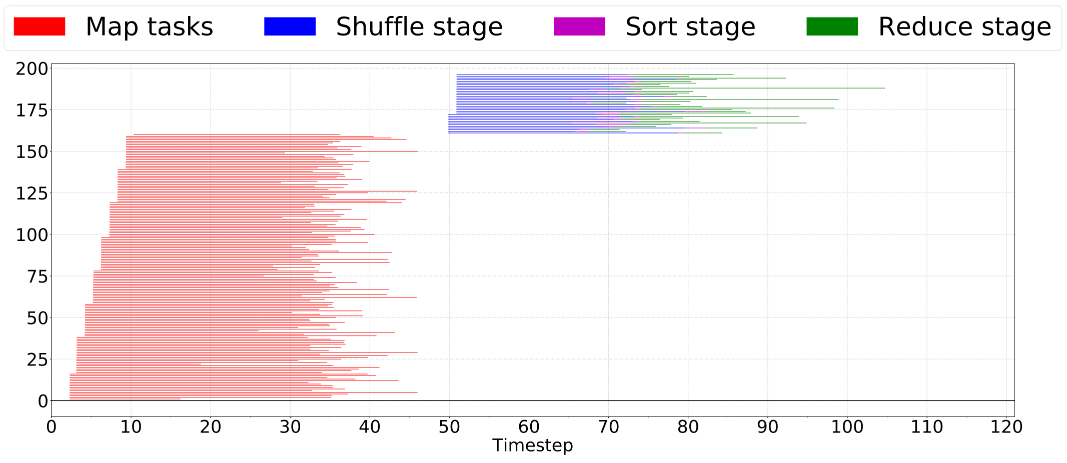
10.3.1 Hadoop when NO overlapping between the map and reduce phases

To facilitate the analysis of MapReduce jobs and focus on the differences regarding REP and EC (i.e., reading input data and writing output data) we start with the case when there is no overlapping between the map phase and the reduce phase (during the shuffle). The same approach is employed in Spark, while in MapReduce, the shuffle starts when a specific number of map tasks finish (5% by default). This allows the overlapping between the computation of map tasks and the transfer of intermediate data.

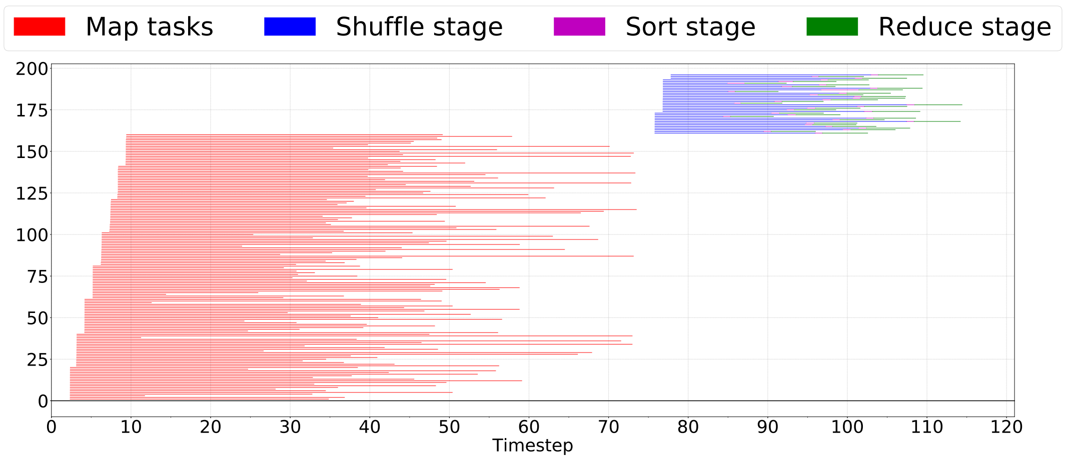
First, the job execution time of Sort application with increasing input sizes is depicted in Figure 10.1a. We can notice that REP slightly outperforms EC. For 40 GB input size (as stated before, we focus on the run with the median job execution time), job execution time under EC is 113.6s, thus 9% higher than that under REP (103.9s). This difference can be explained by the time taken by the map and reduce phases (as shown in Table 10.1), knowing that these two phases are not overlapping. Map phase finishes faster under REP by 38% (70.6s under EC and 43.4s under REP). On the other hand, reduce phase finishes faster under EC compared to REP by 29% (38.3s under EC and 54.3s under REP).

Runtimes distribution of map and reduce tasks. Figure 10.2 shows the timeline of task runtimes under both EC and REP. For simplicity, we start with analyzing the runtimes of reduce tasks and then map tasks. We can clearly see that the main contributor to the increase in the runtimes of reduce tasks under REP compared to EC is reduce stage, in particular, writing data to HDFS. While the times of shuffle and sort stages are almost the same under both EC and REP, REP needs more time to transfer the output data to the DNs: 53.3 GB under EC, among which 10.2 GB are written to disks, while 80 GB are transferred through network to DNs under REP, among which 32.8 GB are persisted to disks. The remaining are buffered in OS caches. On the other hand, we can see a high variation in the runtimes of map tasks under EC compared to REP. Map runtime varies by 33.3% under EC (from 7.9s to 69.2s) while it varies by 15.8% under REP (from 13.7s to 43.2s) and the average runtimes are 38.6s and 29.6s under EC and REP, respectively. Interestingly, we find that the minimum runtime of map tasks under EC is 7.9s while it is 13.7s under REP. Moreover, the runtimes of 25% of map tasks under EC are below the average of map runtimes under REP. Hence, the degradation in the runtimes of map tasks under EC is not due to data locality or network overhead, especially as the network is under-utilized during the whole map phase under EC (100 MB/s on average).

Zoom-in on map phase. Figure 10.3a shows the distribution of input data. We notice that the variation in data distribution is almost the same under both EC and REP (a standard deviation of 1.36 GB under EC and 1.13 GB under REP). Accordingly, and



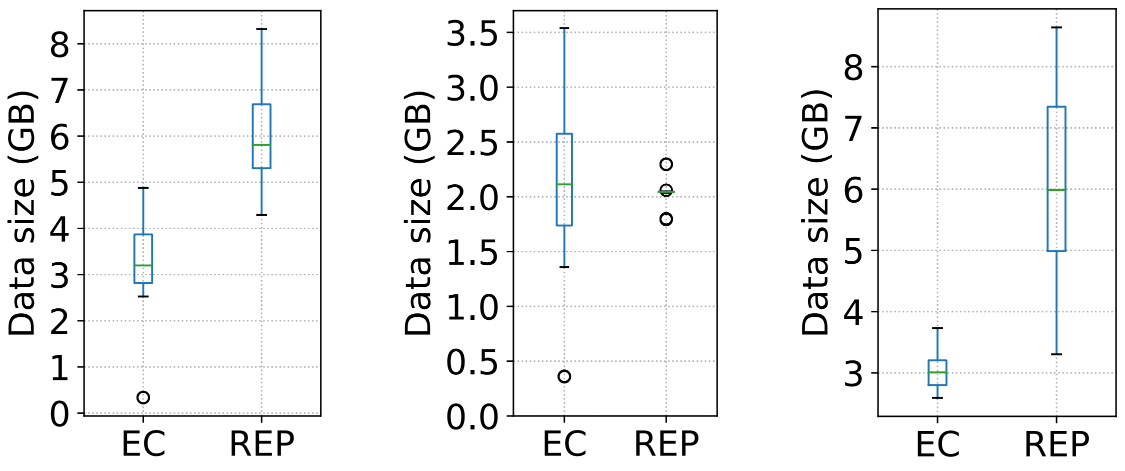
(a) REP



(b) EC

Figure 10.2: Tasks of Sort application under EC and REP (non-overlapping shuffle) for 40 GB input size. The first 160 tasks are map tasks, while the remaining 36 tasks are reduce tasks.

given that each node executes the same number of map tasks, we expect that the amount of data read by each node is the same: as we run 8 containers per node, and each map task handles 1 block of data (256 MB), therefore, ideally 2 GB of data should be read by each DN. However, as shown in Figure 10.3b, the amount of data read varies across nodes under both REP and EC. Under REP, we can see this with a couple of outliers that represent non-local reads (in our experiments, the achieved data locality is 94%); hence this contributes to the variation in the map runtimes under REP. On the other hand, we observe high variation (44.3%) in the data read across DNs under EC. This imbalance of data read under EC is related to the fact that HDFS block distribution algorithm does not distinguish between data and parity chunks under EC, thus, some DNs might end up with more parity chunks than others even though they have the same total number of chunks. As map tasks do not read parity chunks when there is no failure or data corruption, the imbalance in data read occurs and results in longer runtimes of map tasks. Nodes which are continuously serving map tasks running in other nodes



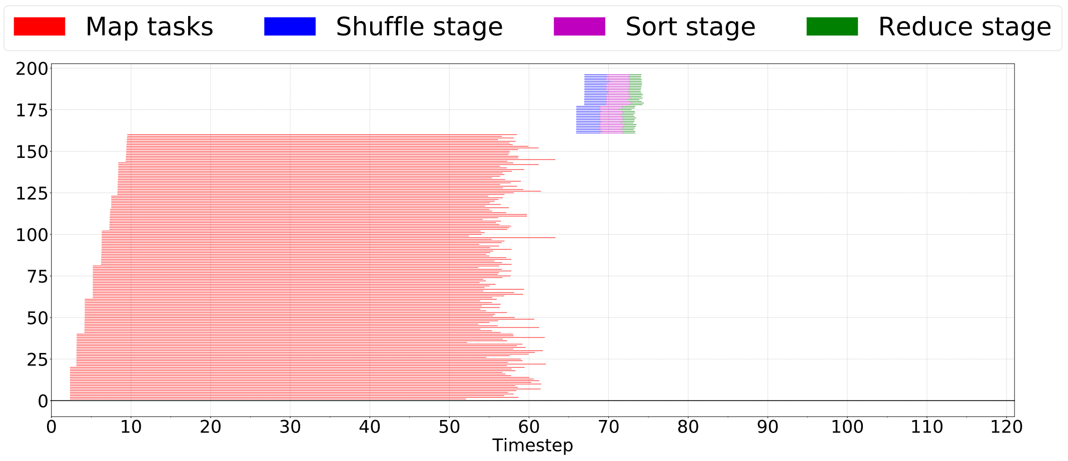
(a) Initial data distribution (NO) (b) Data read per machine (NO) (c) Write load distribution (O)

Figure 10.3: Initial data load on DNs, data read per node, and data written per node for non-overlapping (NO) and overlapping (O) cases.

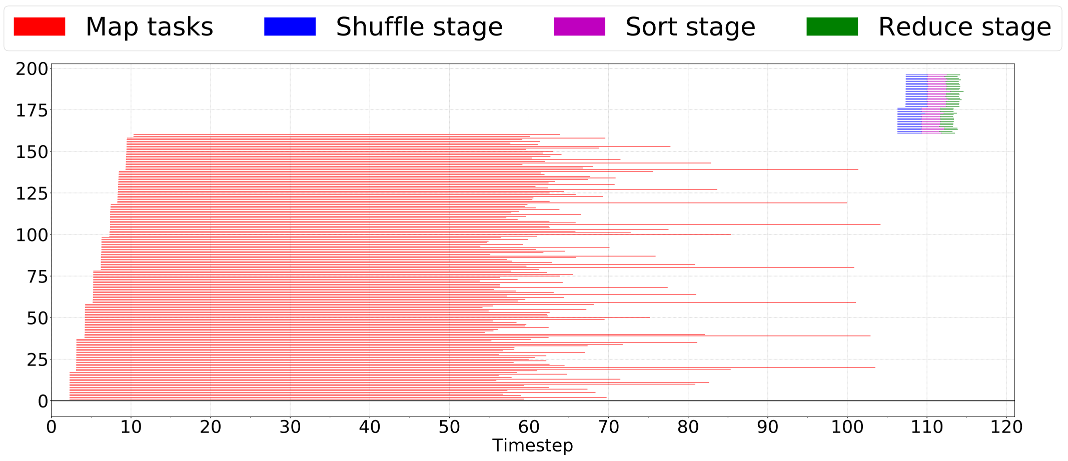
exhibit high CPU iowait time and therefore the runtimes of map tasks running within them increase. The median iowait time of the node with the largest data read (3.53 GB) is 82% while the median iowait time of the node with the lowest data read (0.35 GB) is 0.7%. Consequently, the average of map runtimes (map tasks running within the two aforementioned nodes) is 59s and 25s, respectively. As a result, the runtimes of map tasks which are severed by those nodes also increase. In conclusion, Hadoop exhibits a high read imbalance under EC which causes stragglers. This, in turn, prolongs the runtimes of map tasks compared to REP and causes high variation in map runtimes.

The same trend can be observed with Wordcount application (where the shuffle data and final output are relatively small compared to the input size); Load imbalance in data read across nodes under EC which results in longer (and high variation in) map runtimes (CPU iowait during the map phase is around 15% under EC and almost zero under REP). However, as the job execution time (shown in Figure 10.1b) is dominated by the map phase (as shown in Table 10.1 and Figure 10.4, the map phase accounts for around 90% of the execution time for Wordcount application), and given that map phase is quicker by 40% under REP (101s under EC and 60.4s under REP); Wordcount application finishes 35% slower under EC compared to REP (the job execution time is 113.7s and 73.8s under EC and REP, respectively). Note that reduce phase takes almost the same time under both EC and REP (8.2s and 8.4s, respectively) as the final output data is relatively small.

Finally, Figure 10.5a and Figure 10.5b show the amount of transferred data between the DNs when running Sort and Wordcount applications with different input sizes, respectively. When sorting 40 GB input size, 137 GB is transferred under EC, 8.7% more than that under REP (125 GB). However, for Wordcount application, 10x more data are transferred under EC. As shuffled and output data sizes are small compared to the input data, all the extra data under EC is attributed to the non-local read. However, for Sort application, the amount of non-local data read under EC is compensated when writing



(a) REP



(b) EC

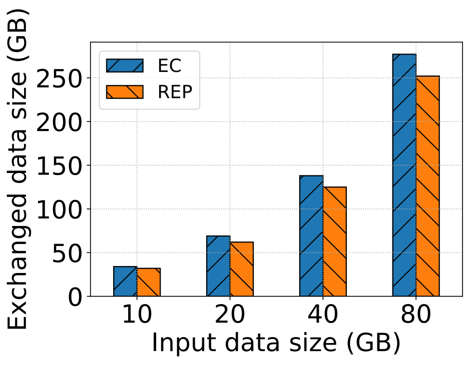
Figure 10.4: Tasks of Wordcount application under EC and REP (non-overlapping shuffle) for 40 GB input size. The first 160 tasks are map tasks, while the remaining 36 tasks are reduce tasks.

(i.e., replicating) the output data under REP.

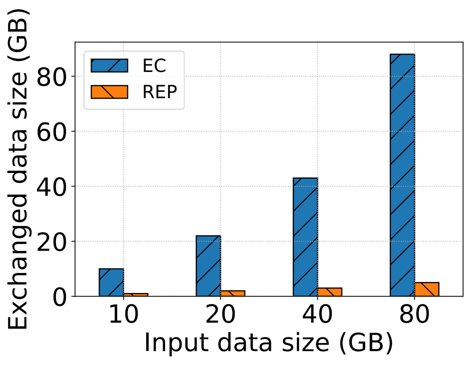
`Observation 7.` Though they have different functionalities, chunks (i.e., data and parity) are treated the same when distributed across DNs. This results in a high variation in the data reads amongst the different nodes in Hadoop cluster when running MapReduce applications. Data read imbalance can degrade the performance of MapReduce applications. The performance degradation related to the stragglers caused by hot-spots (nodes with large data reads).

10.3.2 The case of overlapping shuffle

Similarly to the previous section, we start by analyzing the execution of Sort application under both storage policies. It is expected that overlapping the shuffle and the map phase will result in better performance in both cases, especially under REP. However,

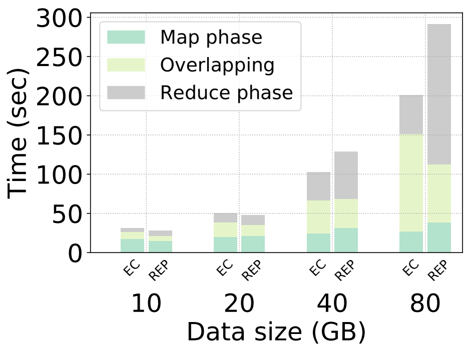


(a) Sort application

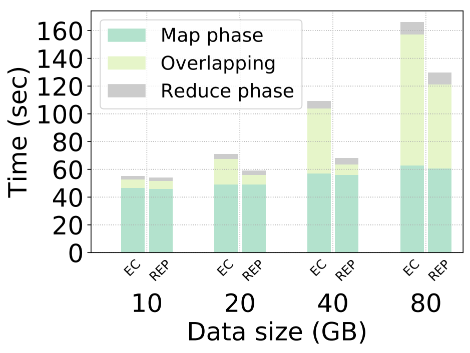


(b) Wordcount application

Figure 10.5: Amount of exchanged data between DNs during the job execution (non-overlapping shuffle).



(a) Sort application



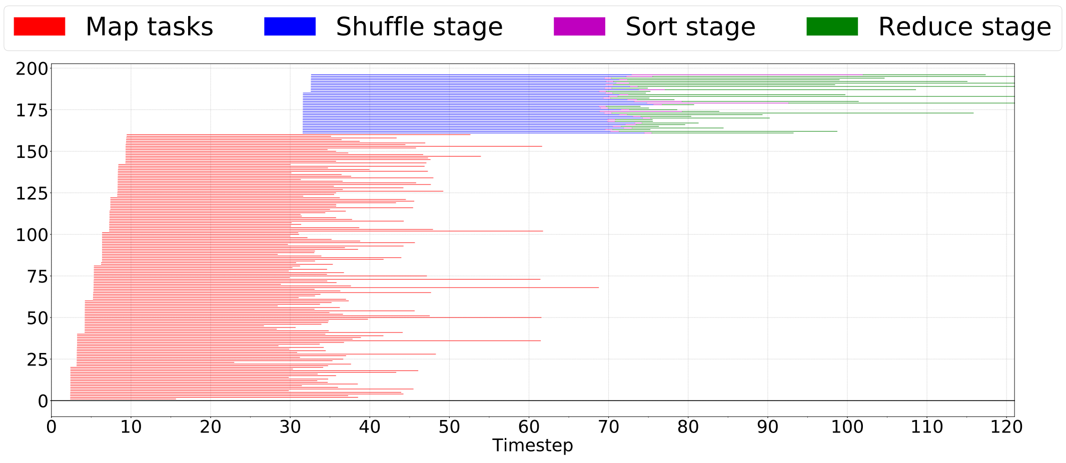
(b) Wordcount application

Figure 10.6: Job execution time of Sort and Wordcount applications under EC and REP.

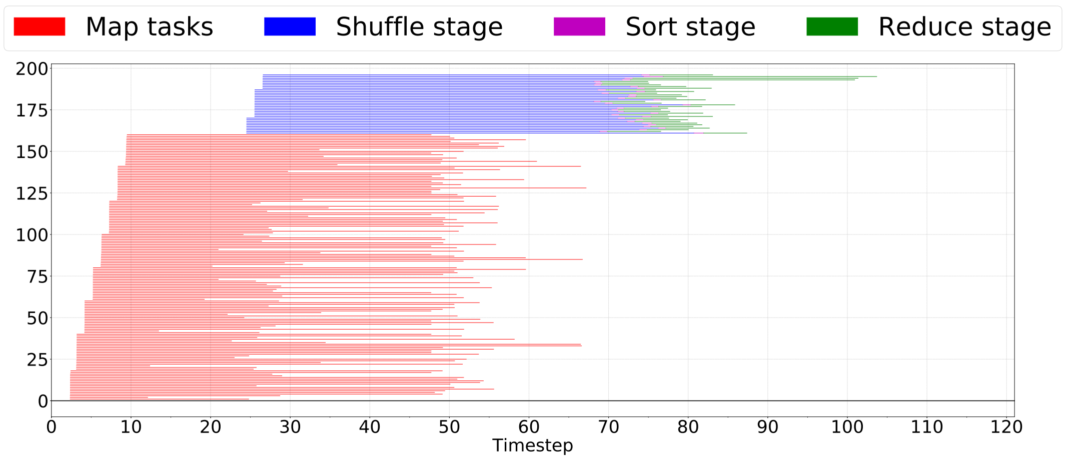
this is only true for small data inputs (non-overlapped run is slower by up to 30%). With large data inputs, overlapping results in a degradation in the performance of MapReduce applications under replication due to several stragglers (map and reduce) as explained below.

Job execution times: EC *vs.* REP. For Sort application with 40 GB input size, job execution time under EC is 103s while it is 129s under REP, thus 20% higher than that under EC as shown in Figure 10.6a. Moreover, for 80 GB input size, the improvement in the execution time under EC is increased to 31% (201s under EC and 291s under REP). The difference in job execution time can be explained by the time taken by the map and reduce phases. Map phase finishes faster under EC by 3.5% (66.6s under EC and 68.2s under REP). Moreover, the reduce phase is completed faster under EC by 24.5% (78.5s under EC and 97.8s under REP on average). Importantly, the reduce stage is 59% faster under EC (7.7s) compared to REP (19s).

Runtime distribution of map and reduce tasks. Figure 10.7 shows the timeline of task runtimes under both EC and REP. We can still see that the main contributor to the increase in the runtimes of reduce tasks under REP compared to EC is the reduce stage. However, different from the non-overlapping scenario, the runtimes of map tasks



(a) REP

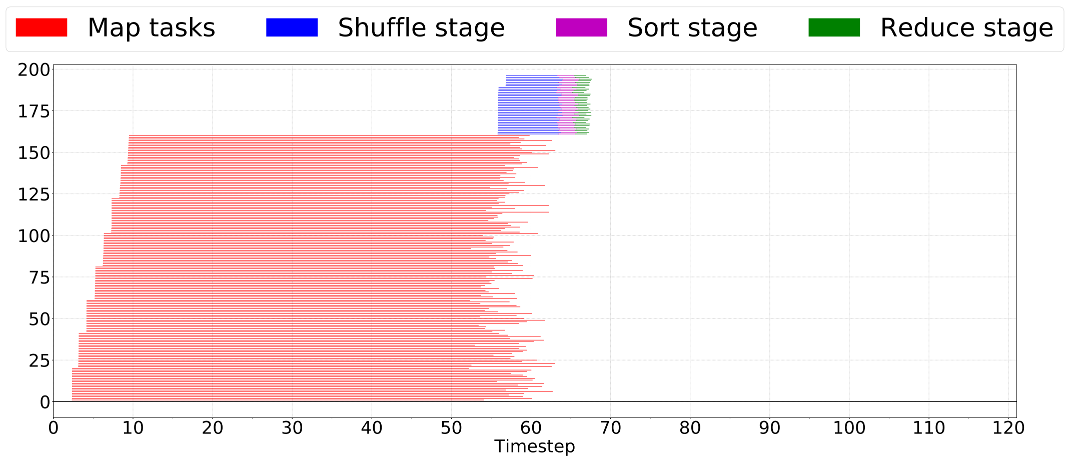


(b) EC

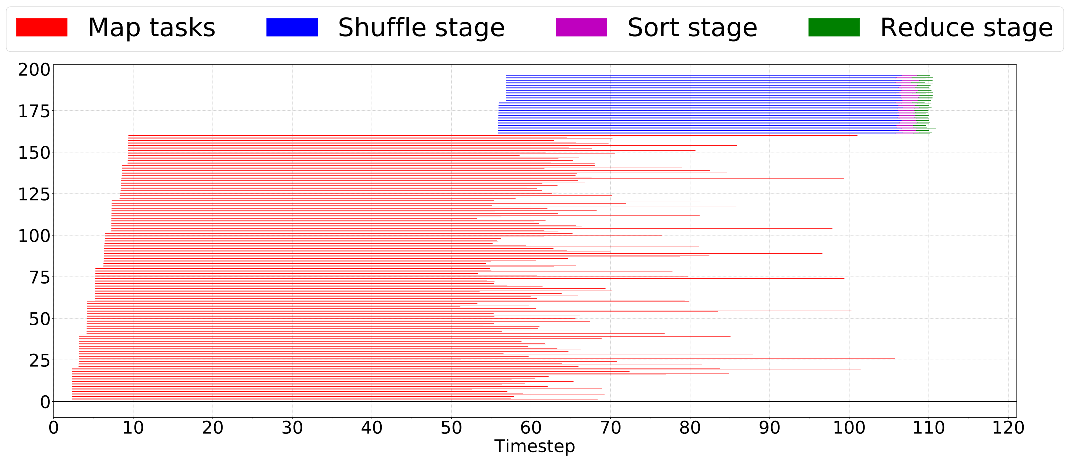
Figure 10.7: Tasks of Sort application under EC and REP for 40 GB input size. The first 160 tasks are map tasks, while the remaining 36 tasks are reduce tasks.

and reduce tasks exhibit high variation under both EC and REP. Map runtime varies by 33.3% under EC (from 9.1s to 62.8s) while it varies by 23.9% under REP (from 13.1s to 62.9s) and the average map runtimes are 37.5s and 31.8s under EC and REP, respectively. Reduce runtime varies by 12.2% under EC (from 47.9s to 76.4s) while it varies by 29.7% under REP (from 41.9s to 96.1s) and the average is 55.4s and 60.8s under EC and REP, respectively. Interestingly, we observe a high ratio of stragglers (heavy-tails) under REP: the runtimes of 5% of map tasks are at least 1.4x, and 1.3x longer than the average map runtimes under REP and EC, respectively; and the runtimes of 20% of reduce tasks are 1.3x longer than the average reduce runtimes under REP.

Zoom-in on map phase. Similar to the non-overlapping case, we still observe high variation in the data read across nodes which causes long iowait times and therefore increases the runtimes of the map tasks which are executed or served by nodes with large data read (as in the non-overlapping case). While the median iowait time of the node with the highest data read (2.88 GB) is 63.6% and the average map runtimes is 49.5s, the median iowait time of the node with lowest data read (0.36 GB) is 3.1% and



(a) REP

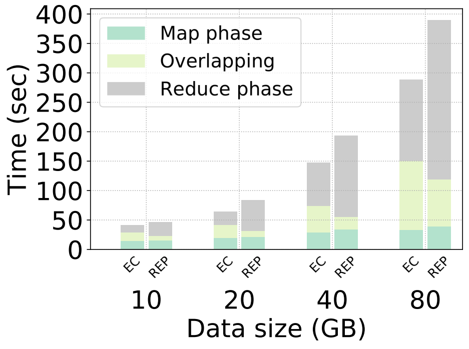


(b) EC

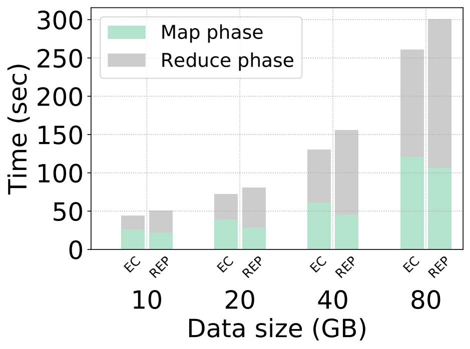
Figure 10.8: Tasks of Wordcount application under EC and REP for 40 GB input size. The first 160 tasks are map tasks, while the remaining 36 tasks are reduce tasks.

the average map runtimes is $21.1s$. Surprisingly, this waiting time is not much longer than the one observed in the non-overlapping scenario for the node with the largest data read, knowing that nodes are also writing data which are shuffled from other nodes. In conclusion, map tasks finish faster in nodes with low read data and therefore more reduce tasks are scheduled to them. This increases the waiting times and also increases the variation in the reduce stages. While this imbalance in reduce tasks distribution helps to reduce the variation in the map runtimes under EC, it prolongs the runtimes of some maps tasks under REP and more importantly, it prolongs and causes high variation in the runtimes of reduce tasks under REP. Figure 10.3c shows the write data across nodes, we can observe high variation in the write data under REP compared to EC.

Similarly, the same trend can be observed with Wordcount application; high variation in map tasks under EC that causes longer job execution time as shown in Figure 10.8 and Figure 10.6b.



(a) Overlapping shuffle



(b) NO overlapping between Map and Reduce phases

Figure 10.9: Job execution time of Sort application under EC and REP with disk persistence enabled.

`Observation 8.` Erasure coding can speed up the execution time of applications with large outputs (e.g., Sort application).

`Observation 9.` It is a bit counter-intuitive that jobs finish faster when disabling overlapped shuffle for large input size especially for Sort application which is shuffle intensive. The main reason behind that is resource allocation in YARN: Reduce tasks are launched when resources are available and after 5% of map tasks finished; this, on the one hand, may delay the launching of some map tasks as resources are occupied by early launched reduce tasks, and on the other hand, increases skew across nodes and cause stragglers under replication but not EC.

10.3.3 The impact of disk persistency

Usually, data-intensive clusters are shared by multiple applications and job outputs are synchronized to disks directly (not buffered in caches). Thus, job outputs are completely written to disk. To study the impact of disk persistency on the performance of MapReduce applications under EC and REP, we make sure that the data outputs are completely flushed to disk in the reduce stage (i.e., MapReduce jobs are considered successful when outputs are persisted to disk completely). We focus on Sort application since the size of output data is equal to the size of input data, thus, the impact of persisting the data to disk is clearer, in contrast to Wordcount application.

Results. Figure 10.9a shows the job execution times of Sort application in case of overlapping shuffle. For 40 GB input size, the job execution time under EC is 149.4s, thus, 25% faster than under REP (201.3s). As expected, the job execution time with disk persistency has increased compared to the previous scenario (Section 10.3.2). Obviously persisting output data does not impact reading input data, thus, the map phase has the same duration. The main increase in the execution time is attributed to the reduce stage. Reduce phase under REP takes 108.4s (97.8s previously), in which 89s are spent in the reduce stage (writing output data), while under EC, reduce phase takes 70.4s, of which 46.6s for the reduce stage, 1.9x faster than reduce stage under REP.

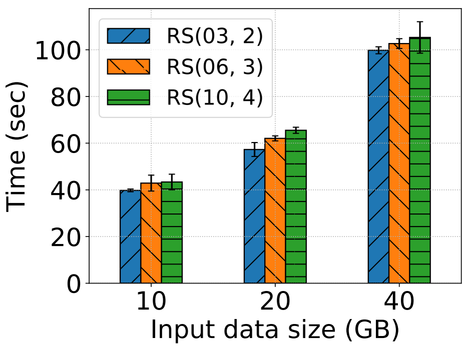


Figure 10.10: The impact of different RS schemes on Sort execution time.

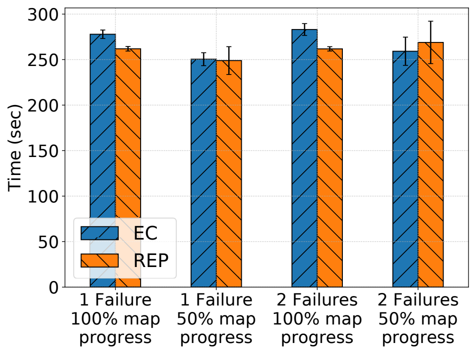
Figure 10.9b shows the job execution times of Sort application when there is no overlapping between map and reduce phase. Compared to previous results (see Figure 10.1a), we can notice that with disk persistency, job execution times under EC are now lower than those under REP. For example, for 40 GB input size, job execution time under EC is 130.4s, while it is 155.3s under REP, thus, 16% faster under EC. Here, the amount of data written to disk – to complete the job – is 120 GB (32.8 GB previously) under REP while it is 60 GB (10.2 GB previously) under EC. Consequently, reduce phase under REP takes 104.5s (54.3s previously), in which 70.6s is spent in the reduce stage, while under EC, reduce phase takes 68s, of which 24.8s for the reduce stage, 2.8x faster than reduce stage under REP.

`Observation 10.` When output data are completely persisted to disk, jobs under EC are clearly faster than those under REP, at least during the reduce stage. This situation (synchronizing output data to disks directly) is common in shared clusters as the available memory to buffer output data is usually limited [203].

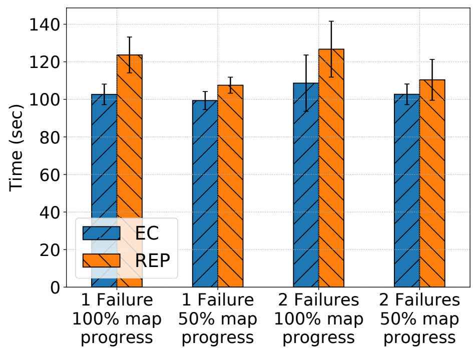
10.3.4 The impact of RS schemes

In this section, we present the impact of different RS schemes on jobs execution time of MapReduce applications. We note here that different schemes have different fault tolerance guarantee, therefore, they could not be considered as alternatives, however, we compare them from their performance point of view.

Results. Figure 10.10 shows job execution time of Sort application under different RS schemes. For all the input sizes, we can notice an increase in the job execution time while increasing the number of data chunks of the EC scheme. For example, for 40 GB input size, job execution time is 99.8s, 102.7s, and 105.3s under $RS(3, 2)$, $RS(6, 3)$, and $RS(10, 4)$, respectively. The small difference in job execution time between these schemes is mainly contributed by the map phase (In reduce phase, the same amount of data is transferred across nodes under the three EC schemes). First, more data is read locally when the number of data chunks is small. Second, as each map task involves n reads in parallel, increasing the number of data chunks (n) accelerates reading the inputs of map tasks. But increasing the number of chunks increases the number of I/O accesses and causes higher CPU iowait time and therefore increases the execution of map tasks (performing the map operations). The average CPU iowait time per node is 19.8s, 27.8s,



(a) With default timeouts



(b) With immediate failure detection

Figure 10.11: Sort execution time with 40 GB input size under failure.

and 28.6s under $RS(3,2)$, $RS(6,3)$, and $RS(10,4)$, respectively. Consequently, the average runtimes of map tasks are 35.4s, 37.3s, and 37.7s under $RS(3,2)$, $RS(6,3)$, and $RS(10,4)$, respectively. Finally, we observe that the three EC schemes exhibit almost the same data read skew, with slightly higher skew under $RS(10,4)$; this leads to maximum runtimes of map tasks of 60s, 60.9s, and 65.6s under $RS(3,2)$, $RS(6,3)$, and $RS(10,4)$, respectively.

`Observation 11.` While increasing the size $(n+k)$ of RS schemes can improve failure resiliency, it reduces local data accesses (map inputs) and results in higher disk accesses. Moreover, this increases the probability of data read imbalance (i.e., it introduces stragglers).

10.3.5 Performance under failure

A well-known motivation for replication and erasure coding is tolerating failures. That is, data are still available under failure and therefore data-intensive applications can complete their execution correctly (though with some overhead). In this section, we study the impact of node failure on MapReduce applications under both EC and REP. We simulate node failure by killing the NodeManager and DataNode processes on that node. The node that hosts the processes to be killed is chosen randomly in each run – we make sure that the affected node does not run the ApplicationMaster process. We fix the input size to 40 GB, and focus on Sort application with non-overlapping shuffle, for simplicity. We inject one and two node failures at 50% and 100% progress of the map phase.

Failure detection and handling. When the RM does not receive any heartbeat from the NM for a certain amount of time (10 minutes by default), it declares that node as *LOST*. Currently running tasks on that node are marked as *killed*. In addition, completed map tasks are also marked as *killed* since their outputs are stored locally on the failed machine, not in HDFS as reduce tasks. Recovery tasks, for *killed* ones, are then scheduled and executed on the earliest available resources. If a task – running in a healthy node – is reading data from the failed node (i.e., non-local map task under REP, map tasks under EC, and reduce tasks), it switches to other healthy nodes. In particular, non-local map tasks under REP continues reading the input block from another replica; map tasks under EC triggers *degraded read* (i.e., reconstruct the lost data chunk using the remaining data chunks and a parity chunk); and reduce tasks read the data from the recovery map tasks.

Results. Figure 10.11 shows the job execution time of Sort application when changing the number of failed nodes and the failure injection time with the default timeout. As expected, the job execution time increases under failure, under both EC and REP, compared to failure-free runs. This increase is mainly attributed to the time needed to detect the killed NM(s) and the time needed to execute recovery tasks. Moreover, the job execution time of Sort application is longer when increasing the number of failed nodes or the time to inject failures. This is clearly due to the increase in the number of recovery tasks. For example, when injecting the failure at 100% of map progress, the execution time is around 19s and 24s longer compared to injecting failure at 50% map progress under both EC and REP, respectively. This is due to the extra cost of executing recovery reduce tasks. This is consistent with a previous study [71].

Failure handling under EC and REP. To better understand the overhead of failures under EC and REP, we provide an in-depth analysis of failure handling. We make sure that failed nodes are detected directly and therefore eliminate the impact of failure detection on job execution time. Surprisingly, we find that the overhead of failures is lower under EC than under REP when failure is injected at 50% map progress. For example, when injecting two node failures, the job execution time increases by 3.2% and 4.7% under EC and REP, respectively. This is unexpected as more tasks are affected by failures under EC compared to REP (in addition to recover tasks, tasks with degraded reads). On the one hand, the total number of degraded reads is 78 degraded reads, among which 4 degraded reads are associated with the recovery tasks. However, as the main difference between normal map tasks and map tasks with degraded reads is the additional decoding operation to construct the lost chunk and given that this operation does not add any overhead, degraded reads incur almost zero overhead. To further explain: in case of a normal execution of map task, 6 data chunks will be read, cached and processed; while in case of a task with a degraded reads 5(4) data chunks and 1(2) parity chunk(s) will be read, cached, decoded and processed; hence no extra data is retrieved and no extra memory overhead as chunks are eventually copied to be processed. Hence, the average runtimes of map tasks with and without degraded read are almost the same (38s). On the other hand, recovery map tasks are faster under EC compared to REP: the average runtimes of recovery tasks under 2 failures are 6.3s and 7.5s under EC and REP, respectively. This is due to the contention-free parallel reads (recovery tasks are launched after most of the original tasks are complete) and the non-local execution of recovery tasks under REP (75% of recovery tasks are non-local, this is consistent with a previous study [306]).

`Observation 12.` Unlike EC with contiguous block layout which imposes high network and memory overhead and extra performance penalty under failures [173], degraded reads under EC with striped block layout introduces negligible overhead and therefore the performance of MapReduce applications under EC is comparable to that under REP (Even better than REP when recovery map tasks are non-locally executed).

Table 10.2: Performance measures for Kmeans application.

	EC	**REP**
Total execution time (s)	2077	2143
Data Read from disk (GB)	225	536
Data written to disk (GB)	396	743

10.3.6 Machine learning applications: Kmeans

A growing class of data-intensive applications is Machine Learning (ML) applications. ML applications are iterative by nature: input data is re-read in successive iterations, or read after being populated in later iterations.

In this section, we present the performance of Kmeans application under EC and REP. Kmeans application proceeds in iterations, each one is performed by a job and followed by a classification step. In each iteration, the complete data set is read (i.e., 222 GB) and the cluster centers are updated accordingly. The new centers (i.e., few kilobytes) are written back to HDFS during the reduce phase. The classification job is a map-only job that rewrites the input samples accompanied by the "cluster ID" it belongs to and a "weight" that represents the membership probability to that cluster. Therefore, it has an output of 264 GB.

Results. Figure 10.12 shows the execution time of each job under both EC and REP. The execution time of the first iteration is 204s and 282s under REP and EC, respectively. This 27% difference in the execution time is due to the longer iowait time under EC (iowait time is 19% under EC while it is 11% under REP) and due to the stragglers caused by hot-spots under EC (i.e., the average and the maximum map runtimes are 46.7s and 131.5s under EC while they are 32.2s and 40.3s under REP). For later iterations, the data is served mostly from memory. The whole data is requested from the same nodes but from caches under EC and thus the execution time is reduced by almost 40%. But, as data can be requested from 3 DNs under REP, there is a higher probability that some DNs are serving data from disk (even in the last iteration): at least 9 GB of data is read from disks in later iterations under REP. Hence, this explains the slight advantage of EC against REP, given that both are expected to perform the same as data is mostly served from memory (more details in Section 10.4). Finally, the execution time of the classification phase is 27% faster under EC compared to REP. This is expected as the output data under REP is double the one under EC: 792 GB under REP among which 743 GB is persisted to disk, and 396 GB under EC which is completely persisted to disk. As a result, Kmeans application runs slightly faster under EC (It takes 2143s under REP, while it is 2077s under EC). Table 10.2 summaries the differences under EC and REP.

`Observation 13.` Under EC, iterative applications can exploit caches efficiently. This reduces the disk accesses and improves the performance of later iterations. While subsequent jobs always read data from memory under EC, this is not the case under replication as multiple replicas of the same block could be eventually read.

`Observation 14.` Iterative applications have similar performance under both EC and REP. Caching input data after the first iteration shifts the bottleneck to the CPU for subsequent iterations, therefore, EC and REP show the same performance.

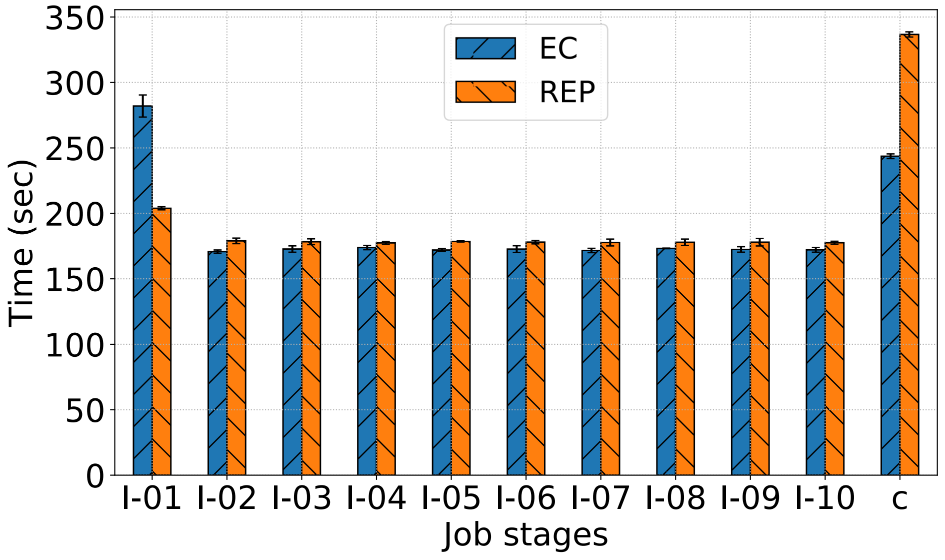


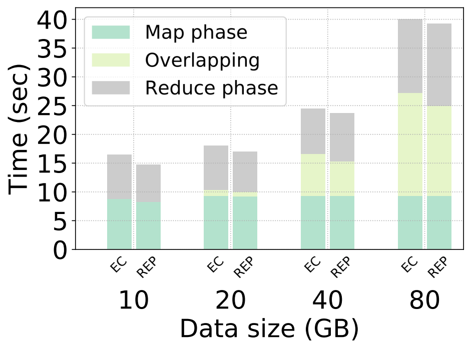
Figure 10.12: Job execution time of Kmeans application under both EC and REP. (I) jobs represent the iterations while (c) job is the classification step.

10.3.7 Implications

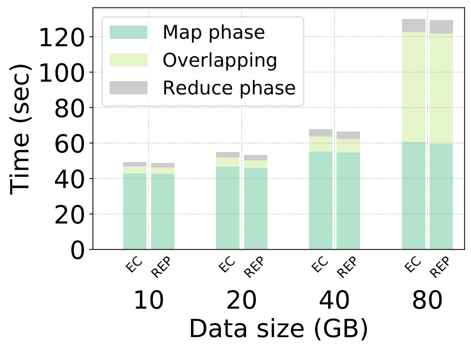
Despite the large amount of exchanged data over the network when reading the input data, EC is still a feasible solution for data-intensive applications, especially the newly emerging high-performance ones, such as Machine Learning and Deep Learning applications, which exhibit high complexity and require high-speed networks to exchange intermediate data. Furthermore, the extra network traffic induced when reading input data under EC is compensated with a reduction by half of the intra-cluster traffic and disk accesses when writing the output data. Analysis of the well-known CMU traces [55] shows that the size of output data is at least 53% of the size of input data for 96% of web mining applications. In addition, the total size of output data in three production clusters is 40% (120% if replicated) of the total size of input data. More importantly, EC can further speed up the performance of MapReduce applications when mitigating the map stragglers. This cannot be achieved by employing speculative execution but that would require to rethink data layout in EC and map task scheduling in Hadoop.

10.4 The Role of Hardware Configurations

The diversity of storage devices and the heterogeneity of networks are increasing in modern data-intensive clusters. Recently, different storage devices (i.e., HDDs, SSDs, DRAMs and NVRAM [230, 312, 193, 158]) and network configurations (i.e., high-speed networks with RDMA and InfiniBand [288, 182] as well as slow networks in geo-distributed environments [126]) have been explored to run data-intensive applications. Hereafter, we evaluate MapReduce applications with different storage and network configurations.



(a) Sort application



(b) Wordcount application

Figure 10.13: Job execution time of Sort and Wordcount applications when main memory is used as a backend storage for HDFS.

10.4.1 Main memory with 10 Gbps network

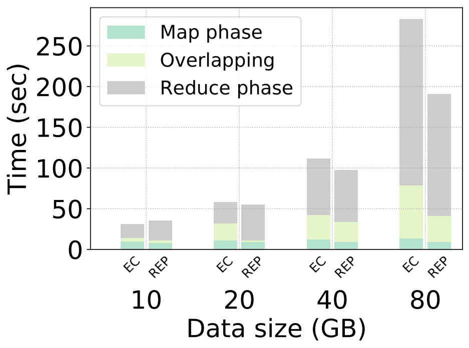
We start by evaluating the performance of MapReduce applications when the main memory is used as backend storage for HDFS. The job execution time of Sort application with increasing input sizes is shown in Figure 10.13a. As expected, compared to running on HDDs, the job execution time reduced significantly (e.g., under EC and with 40 GB input size, Sort application is 4x faster when HDFS is deployed on the main memory). This reduction stems for faster data read and write. For 40 GB input size, the job execution time is 24.4s under EC and 23.7s under REP. Both map and reduce phases finish faster under REP compared to EC: map phase is completed in 15.4s under EC and 13s under REP while reduce phase is completed in 14.8s under EC and 14.3s under REP. Moreover, similar to HDDs, we observe imbalance in the data read under EC. But since there is no waiting time imposed by the main memory, map tasks which are running or served by the nodes with large data reads experience small degradation due to the network contention on those nodes: some map tasks take 1.4x longer time compared to the average map runtimes. This results in a longer map phase, and also leads to a longer reduce phase, despite that the reduce stage is a bit faster under EC.

Wordcount application performs the same under both EC and REP as shown in Figure 10.13b. The job execution time is dominated by the map phase which is limited by the CPU utilization (CPU utilization is between 80% and 95%). Therefore, reading the input data has a lower impact on the map runtimes.

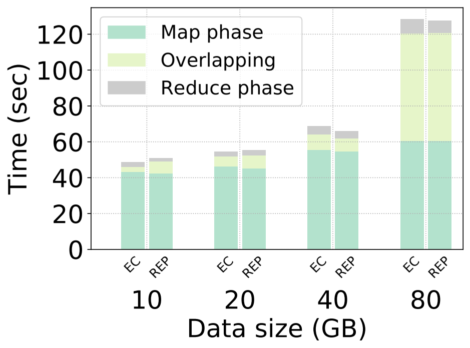
Notably, we also evaluate MapReduce applications when using SSDs as backend storage for HDFS. However, we did not present the results because they show similar trends to those of main memory.

`Observation 15.` Using high-speed storage devices eliminate the stragglers caused by disk contention, therefore, EC brings the same performance as replication. However, EC can be the favorable choice due to its lower storage overhead.

`Observation 16.` Hot-spots still cause stragglers under EC on memory-based HDFS, but those stragglers are caused by network contention.



(a) Sort application



(b) Wordcount application

Figure 10.14: Job execution time of Sort and Wordcount applications when HDFS data is stored in main memory and the network bandwidth is limited to 1 Gbps.

10.4.2 Main memory with 1 Gbps network

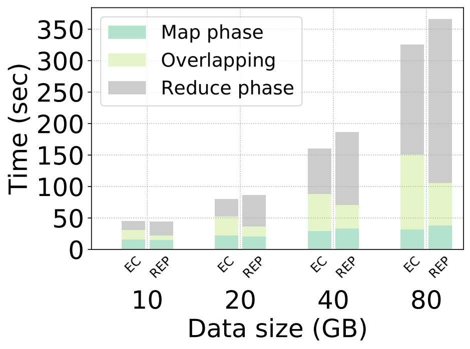
Figure 10.14a shows the job execution times of Sort application when using 1 Gbps network. Sort application has a shorter execution time under EC for 10 GB input size, while REP outperforms EC for larger input sizes. For example, for 80 GB input size, job execution time is 48.2% faster under REP compared to EC, while it is 14% faster for 40 GB input size. For 40 GB input size, map tasks have an average runtime of $4.8s$ under REP, while it is $19.6s$ under EC. On the other hand, reduce tasks under EC finish faster ($65.6s$) on average compared to reduce tasks ($74.8s$) under REP. Specifically, the reduce stage takes on average $27.3s$ under EC, while it takes $48.2s$ under REP. The main reason of the long reduce phase under EC, though the reduce stage is relatively short, is the long shuffle time caused by the imbalance in reduce tasks execution across nodes: some reducers finish 1.5x slower than the average.

Similar to the scenario when using 10 Gbps network, we observe that Wordcount application performs the same under both EC and REP as shown in Figure 10.14b.

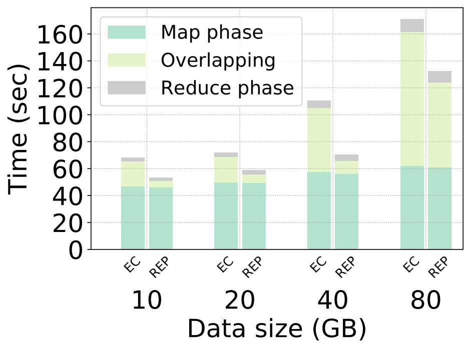
`Observation 17.` As the network becomes the bottleneck, the impact of data locality with replication is obvious for applications with light map computation (e.g., Sort application), but not for applications where map tasks are CPU intensive (e.g., Wordcount application). Therefore, caching the input data in memory is not always beneficial, this depends on the application.

10.4.3 HDD with 1 Gbps network

We study in this section the impact of 1 Gbps network on job execution time under both storage policies. Figure 10.15a depicts the job execution times of Sort application under both EC and REP while increasing input sizes. For 40 GB input size, the job finishes in $177s$ under EC while it takes $191s$ to finish under REP, and thus EC outperforms REP by 6.9%. Even though map tasks take more time on average under EC ($40.5s$) compared to REP ($31.5s$), reduce tasks finish faster under EC ($93s$) compared to under REP ($105s$). Surprisingly, compared to the case with 10 Gbps network, map tasks have



(a) Sort application



(b) Wordcount application

Figure 10.15: Job execution time of Sort and Wordcount applications when the network bandwidth is limited to 1 Gbps.

not been impacted by the slower network (less than 0.5% under REP and up to 7% under EC), but reduce tasks become 45.5% slower under EC and 50.7% slower under REP.

Finally, as shown in Figure 10.15b, we observe similar trends when running Wordcount application in the case of 1 Gbps and 10 Gbps network.

`Observation 18.` The performances of data-intensive applications on slow networks show similar trends as those of fast network (i.e.,10 Gbps). In particular, reading the input data under EC is slightly affected when the network bandwidth is reduced.

10.4.4 Implications

The need for lower response time for many data analytics workloads (e.g., ad-hoc queries) in addition to the continuous decrease of cost-per-bit of SSDs and memory, motivate the shift of analytics jobs to RAM, NVRAM and SSD clusters that host the complete dataset [301, 161] and not just intermediate or temporary data. Deploying EC in those data-intensive clusters does not only result in a better performance but also in lower storage cost. Importantly, EC is also a suitable candidate for Edge and Fog infrastructures which are featured with limited network bandwidth and storage capacity [121].

10.5 Discussion and General Guidelines

Our study sheds the light on some aspects that could be a potential research aspect for data analytics under EC.

Data and parity chunks distribution under EC. We have shown that chunk reads under EC are skewed. This skew impacts the performance of map tasks. Incorporating a chunk distribution strategy that considers data and parity chunks when reading data could result in a direct "noticeable" improvement in jobs execution times; by reducing the impact of stragglers caused by read imbalance. Note that as shown in Section 10.3, those stragglers prolong the job execution times by 30% and 40% for Sort and Wordcount applications, respectively.

EC-aware scheduler. Historically, all the schedulers in Hadoop take data locality into account. However, under EC the notion of locality is different from under replication. Developing scheduling algorithms that carefully consider EC could result in more optimized task placement. The current scheduler in Hadoop treats the task as local if it is running on a node that hosts any chunk of the block, even if it is a parity chunk. Moreover, tasks read always the data chunks even if a parity chunk is available locally. Hence, non-local tasks and local tasks that run on nodes with parity chunks behave exactly the same way with respect to network overhead. Importantly, interference-awareness should be a key design for task and job scheduling under EC.

Degraded reads, beyond failure. In addition to the low network and memory overhead, degraded reads under EC comes with "negligible" cost. This is not only beneficial to reduce the recovery time under failure but can be exploited to add more flexibility when scheduling map tasks by considering the $n + k$ chunks.

Deployment in cloud environment. Networks in the cloud – between VMs – are characterized by low bandwidth. Previous studies measured the throughput as 1 Gbps [308] and usually it varies as it is shared on best-effort way [33, 308, 83]. This results in a higher impact of stragglers under EC. Therefore, straggler mitigation strategies (e.g., late-binding [230]) could bridge the gap and render EC more efficient.

Geo-distributed deployment. Geo-distributed environments, as Fog and Edge [40, 261], are featured by heterogeneous network bandwidth [119, 63]. Performing data processing on geo-distributed data has been well studied. However, employing EC as a data storage policy is not yet explored. It has been shown that achieving data locality may not be always the best case, as sites have limited computations. Thus, moving data to other sites for processing could be more efficient [126]. Hence, storing the data encoded could provide more flexibility and more scheduling options to improve analytics jobs.

High-speed storage devices. Our experiments show that even though replication benefits more from data locality with high-speed storage devices such as SSD and memory (especially with low network bandwidth), this benefit depends on the type of workload.

In conclusion, could erasure codes be used as an alternative to replication? EC will gradually take considerable deployment space from replication as a cost-effective alternative method that provides the same, sometimes better, performance and fault-tolerance guarantees in data-intensive clusters. However, this will require a joint effort at EC level and data processing level to realize EC effectively in data-intensive clusters.

10.6 Conclusion

The demand for more efficient storage systems is growing as data to be processed is always increasing. To reduce the storage cost while preserving data reliability, erasure codes have been deployed in many storage systems. In this chapter, we study to which extent EC can be employed as an alternative to replication for data-intensive applications. Our findings demonstrate that EC is not only feasible but could be preferable as it outperforms replication in many scenarios. On the other hand, jobs under EC might be impacted by the block distribution in HDFS.

In the next chapter, we introduce an algorithm to mitigate some of these shortcomings. Specifically, we develop an EC-aware chunk placement algorithm and we show its impact on the performance of MapReduce applications.

Chapter 11

EC-aware Data Placement in HDFS: Improving the Performance of Data-Intensive Applications

Contents

Erasure coding seems to provide an attractive solution to enable scalable and efficient data processing in data-intensive clusters. However, the previous chapter showed that balancing the data load between DataNodes (DNs) is important for the performance of data-intensive applications. As the data load of a DN is attributed to the amount of data chunks reside in that node, in this chapter, we develop an EC-aware chunk placement algorithm that aims to balance the data chunk distribution among DNs. This, in turn, results in better job execution times as it reduces the variation (and thus the maximum) in task runtimes. We implement our algorithm into HDFS and we experimentally show that jobs' performances can be improved under EC-aware placement.

The remainder of this chapter is organized as follows. First, in Section 11.1, we motive the work by describing the impact of EC-awareness on jobs execution time. The EC-aware chunk placement algorithm is presented in Section 11.2. Experimental methodology is discussed in Section 11.3, followed by the experimental evaluation in Section 11.4. Finally, a discussion about the limitations and how they can be addressed are presented in Section 11.5 while Section 11.6 concludes this chapter.

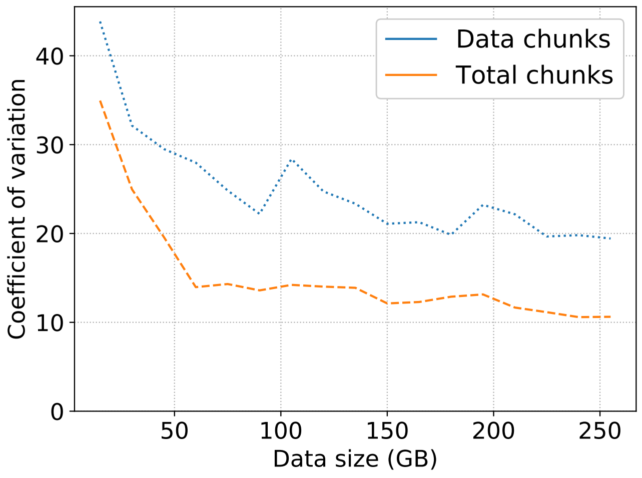


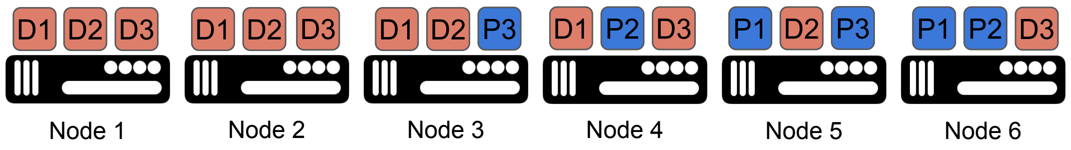
Figure 11.1: Data distribution balance across DNs: "Total chunks" considers all the chunks in the cluster while "Data chunks" considers only the data chunks.

11.1 Why EC-aware Placement?

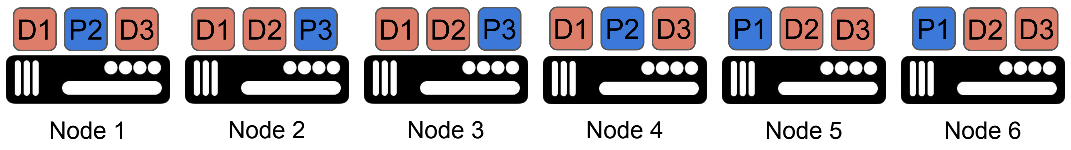
HDFS employs a random block distribution strategy to achieve best-effort balanced data distribution among the DNs at scale. However, for better data reliability, and to ensure rack fault-tolerance, one replica is written to a different rack.

Under replication, HDFS invokes the placement algorithm for each block, that is, if a file is composed of multiple blocks, the replicas of each block end up in different random nodes (supposing that the client writing the file resides outside the HDFS cluster). However, this is not the case for the files written under EC. Under EC with $RS(6,3)$, every EC group which is composed of 6 HDFS blocks – that belong to the same file – are placed together with their parity chunks on the same 9 nodes (as explained in Chapter 8). Thus, data under replications are scattered on more nodes compared to EC. For example, a 6-block file might have its blocks replicas distributed to 18 nodes under REP, while under EC, 9 nodes host the data chunks with their parities. This increases the possibility of having imbalance in data distribution among DNs under EC. Even worse, data placement under EC does not distinct data chunks from parity chunks, thus, DNs exhibit noticeable imbalance in the chunk distribution. Hence, as data processing applications read only data chunks (in failure-free setup), the job execution time is dominated by the tasks running and served by heavy loaded DNs.

We show that skew in data chunk distribution exists in HDFS cluster experimentally. We set EC scheme to $RS(6,3)$ and HDFS block size to 256 MB. We sequentially added 167 files to an HDFS cluster of 20 DNs. Each file has a size of 1.5 GB (one EC group). In total, we added a dataset of 250.5 GB (i.e., 1002 original blocks). Figure 11.1 shows the coefficient of variation of the chunk distribution among DNs for increasing dataset size. We can see that the distribution of data chunks exhibits higher variation than the distribution of the total chunks (data and parity chunks). Moreover, the node with the minimum number of data chunks hosts 204 chunks, while other nodes host up to 414 data chunks. In Section 11.4, we show that the difference in the amount of data chunks between nodes could reach 7x.



(a) A possible imbalanced data chunk distribution



(b) An EC-aware data chunk distribution

Figure 11.2: Two possible chunk distributions of 3 blocks on 6-nodes cluster under EC with $RS(4,2)$ scheme. D_i, P_i denotes a data chunk, a parity chunk of block i, respectively.

Motivating example

To show the impact of chunk placement on job execution time, we consider the following example. Suppose that we have a cluster of 6 nodes hosting 3 HDFS blocks with the $RS(4,2)$ scheme. Figure 11.2a shows a possible distribution of the chunks of these blocks. D_i denotes a data chunk of block i while P_i denotes a parity chunk. Even though all the nodes have the same number of chunks (3 chunks), they do not have the same number of data chunks. More precisely, Node 1 and Node 2 host 3 data chunks while Node 5 and Node 6 host only one data chunk.

To process these 3 blocks, a MapReduce job with 3 map tasks should be launched. Regardless where the map tasks are scheduled, all the data chunks are eventually read by the DNs. As in the current version of Hadoop, tasks (i.e., clients) always choose the nodes that host the data chunks to read the data block, if the data chunks are available (i.e., failure-free mode), the load of each DN depends on the number of data chunks that reside on that node.

If the map tasks are scheduled on nodes 1, 4, and 6. The first map task reads the first block from nodes 1, 2, 3, and 4. The second map task reads the second block from nodes 1, 2, 3, and 5. And finally, the third map task reads the third block from nodes 1, 2, 4, and 6. As a result, the first node becomes a straggler as it serves all tasks simultaneously (3 data chunks). Consequently, the map task running on this node might experience a delay in its runtime due to the long I/O wait time. Moreover, tasks served by this node are also delayed due to the high I/O contention in Node 1. This degrades the performance of MapReduce jobs (as we experimentally demonstrate in Chapter 10).

On the other hand, Figure 11.2b depicts an EC-aware distribution of these data and parity chunks. In this case, all the DNs contribute the same amount of data (i.e., two data chunks) during the processing, reducing the possibility of having stragglers, and thus, the potential degradation in job execution time. However, we note that the current implementation of EC in HDFS is oblivious to the distribution of data chunks, in essence, it is optimized to balance the total number of chunks, but not the number of data chunks, among DNs.

11.2 EC-aware Chunk Placement Algorithm

In an attempt to reduce the skew in data load under EC, we propose a greedy algorithm to distribute data chunks to the DNs evenly in a round-robin manner.

A placement request is issued for each EC group. Under $RS(n, k)$, an EC group is a collection of n (sequential) blocks that belong to the same file, however, for small files or the remaining blocks of a large file, an EC group could contain less than n blocks. In all cases, $n + k$ nodes should be returned to host the data and parity chunks of these blocks.

The algorithm maintains the number of data chunks and parity chunks hosted by each DNs. Initially, they are all set to zero. The algorithm creates an array with the same size as the number of DNs in the cluster and sets the number of data and parity chunks to zero.

Algorithm 4 shows how a write request to HDFS is handled. The algorithm takes as input the previously created and initialized array *datanodes*, the employed EC scheme (*ec_scheme*), and the number of blocks in the EC group (*nb_blocks*). As an output, the algorithm returns the nodes that should host the data chunks (*data_nodes*) and those for parity chunks (*parity_nodes*). For each request, the DNs are sorted in ascending order by the number of data chunks they host, that is, nodes with fewer data chunks are placed first. The first n DNs are chosen to host the new $n * nb\_blocks$ data chunks. After that, the remaining DNs are sorted again in ascending order by their total number of chunks (data + parity), and the first k DNs are chosen to host the $k * nb\_blocks$ parity chunks. This second step does not improve the distribution of data chunks, it only improves the balance of total data distribution among the DNs.

Algorithm 4: EC-aware chunk placement

```
Input  : datanodes, ec_scheme, nb_blocks
Output: data_nodes, parity_nodes
1  Sort datanodes by increasing number of hosted data chunks ;
2  $data\_nodes \leftarrow datanodes[0..ec\_scheme.n]$ ;
3  foreach $dn \in datanodes[0..ec\_scheme.n]$ do
4      $dn.nb\_data\_chunks \leftarrow dn.nb\_data\_chunks + nb\_blocks$ ;
5  end
6  Sort datanodes by increasing number of total hosted chunks ;
7  $parity\_nodes \leftarrow datanodes[0..ec\_scheme.k]$ ;
8  foreach $dn \in datanodes[0..ec\_scheme.k]$ do
9      $dn.nb\_parity\_chunks \leftarrow dn.nb\_parity\_chunks + nb\_blocks$ ;
10 end
11 return $data\_nodes, parity\_nodes$ ;
```

The time complexity of the algorithm is dominated by the time complexity of the two sort functions. Therefore, it is $O(|DN| * log_2(|DN|)$ where $|DN|$ is the number of DNs in the cluster. However, using a priority queue to maintain the order of the DNs can avoid the sort for every execution of the algorithm and it reduces its time complexity to $O(log_2(|DN|)$ (the time complexity for updating an element in a priority queue).

Applying this algorithm to the example mentioned in Section 11.1 results in 18 nodes hosting 300 data chunks and 2 nodes hosting 306 data chunks.

We have to note that the current algorithm only handles adding new blocks to a single rack in HDFS. Rack awareness, data deletion, data loss, data migration, and node decommission are not taken into account in the current version. However, these features do not impact our findings and can be easily adopted.

Implementation

We implemented our algorithm in HDFS as a sub-class of `BlockPlacementPolicyDefault` in the package `org.apache.hadoop.hdfs.server.blockmanagement`. It is composed of less than 200 LOC in Java. The algorithm takes a few milliseconds to run with medium-sized clusters (fewer than 1000 nodes).

11.3 Methodology Overview

We used the same methodology as in the two previous chapters (described in Section 9.2 and Section 10.2).

11.4 Evaluation

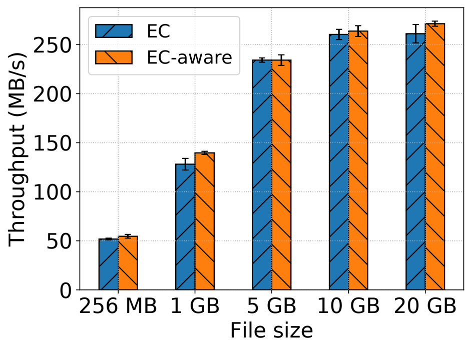
In the following experiments, we study how EC-aware chunk placement impacts the performance of data access and MapReduce applications.

11.4.1 HDFS data access

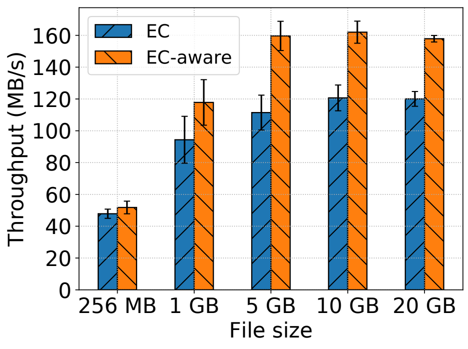
To understand the impact of EC-aware placement, we first evaluate its performance for the read and write operations in HDFS. For single read and write operations, there is no difference as the same number of machines are active in both cases, therefore, we focus on the concurrent access hereafter.

Figure 11.3a shows the write throughput of 10 concurrent clients with increasing file size. We can see a similar performance with a slight improvement under EC-aware placement. For example, 271.3 MB/s write throughput is achieved under EC-aware while it is 261.1 MB/s under default placement when writing 20 GB files. The impact of EC-aware placement on the write throughput is minimal as the written data are buffered in the DNs main memory first masking the impact of disks and thus the load imbalance.

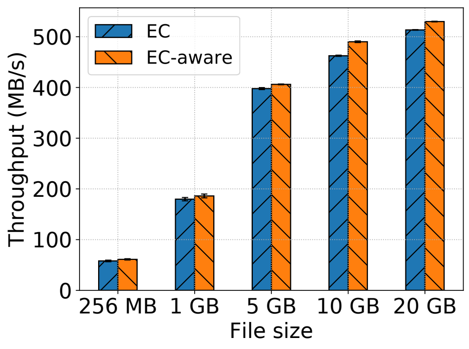
On the other hand, Figure 11.3b depicts the read throughput of 10 concurrent clients with increasing file size while reading distinct files. When reading 10 GB files, the average throughput per client increases from 120.6 MB/s under default placement to 161.9 MB/s under EC-aware placement with an improvement of 34%. Moreover, we observe a lower variation between the clients under EC-aware placement (4.2% compared to 6.7%). This is mainly due to the balance in the distribution of the data chunks across the DNs. Figure 11.3c shows the throughput when reading the same file. As discussed in Section 9.4, the data is read once from the disk and then served from memory. This explains the small difference in throughput between EC-aware and EC, however, this difference could be attributed to the contention of the memory. While the impact of EC-aware placement is clearer when reading from disk, however, EC-aware placement can still bring perfor-



(a) Write distinct files



(b) Read distinct files



(c) Read the same file

Figure 11.3: Concurrent data access with 10 clients under the default placement (EC) and the EC-aware placement.

mance benefits when data are in memory.

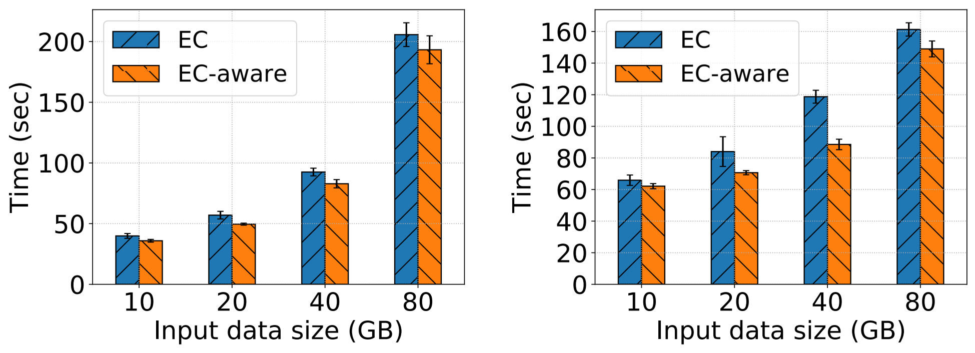
Discussion. EC-aware placement algorithm targets improving the performance of read operation under EC, and therefore, the performance benefit is clear under concurrent read. However, we also observe a slight improvement in the write throughput.

11.4.2 MapReduce applications

As our focus is on reading the input data under EC (i.e., the map phase), in this section, we study the non-overlapping shuffle scenario. When presenting results related to a specific run, we take the job that has the median execution time as a representative one.

Sort application

Figure 11.4a shows the job execution time with default and EC-aware chunk placement for Sort application. Up to 13% reduction in job execution time can be observed under EC-aware placement. For example, for 40 GB input size, job execution time is 91.33s under default placement while it is 82.02s under EC-aware placement on average, thus 10.4% faster. The reduction in job execution time is attributed to the map phase. Map phase accounts for 68.43% of the execution time under the default placement while it



(a) Sort execution time (b) Wordcount execution time

Figure 11.4: Job execution time of Sort and Wordcount applications under EC and EC-aware placements (non-overlapping shuffle).

accounts for 61.97% under EC-aware placement.

For detailed task execution, Figure 11.5 shows the timeline of task runtimes under both EC and EC-aware. While the times of shuffle and sort stages are almost the same under both EC and EC-aware, we can clearly see that the main difference is attributed to the maps tasks which have higher variation, and higher maximum runtime, under default placement than under EC-aware placement. Map tasks under default placement vary by 27.5% (from 8.2s to 56.3s) with an average of 38.2s while under EC-aware placement they vary by 24% (from 8.4s to 42.8s) with an average of 29.9s.

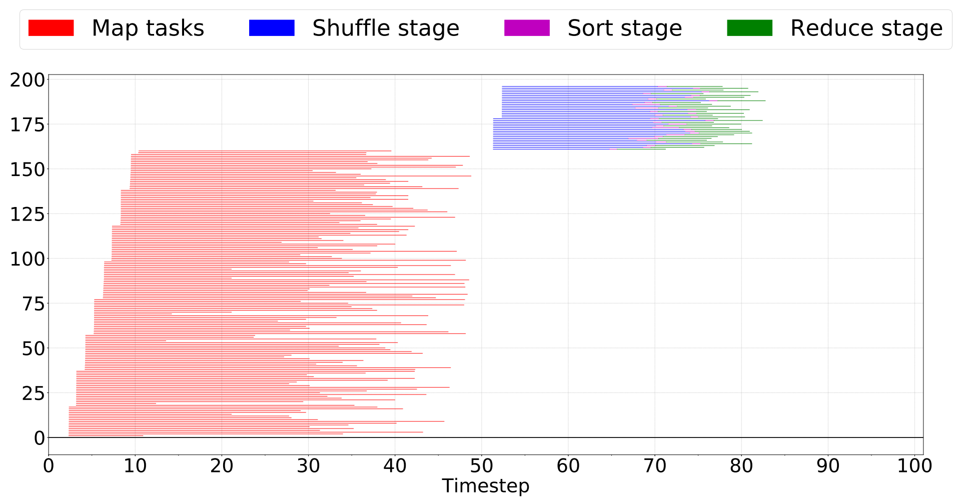
To understand the variation in tasks runtimes under EC, Figure 11.6 shows the distribution of amount of data read from disk by each DN under both placements. Under default placement, we observe a high variation of 36% of the amount of data contributed by each DN (as demonstrated in Chapter 8) with some nodes contributing up to 3.5 GB while others as low as 0.5 GB. On the other hand, EC-aware placement greatly reduces this variation to 8%, as the placement algorithm tries to balance the distribution of data chunks between the DNs as much as possible.

This variation of data read impact negatively the runtime of map tasks. As each map task reads its input block from 6 nodes, therefore, on average, each DN serves data to 48 tasks (supposing that it hosts 2 GB of data on average i.e., 8 blocks). However, if a DN serves 3.5 GB (i.e., 14 blocks), 84 tasks eventually contact this DN to obtain a data chunk (i.e., 1/6 of the block). This might cause high performance bottleneck on the disk of that DN, leading to a delayed execution of the map tasks reading from this DN.

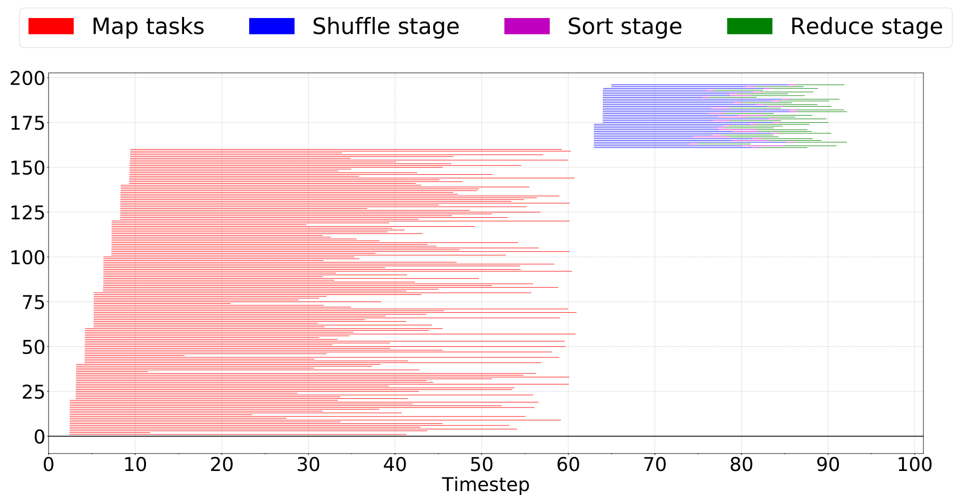
On the other hand, the average runtime of reduce stage is 11.3s under default placement, while it is 11.5s under EC-aware placement. This confirms that EC-aware placement does not adversely impact the write performance of reduce tasks.

Wordcount application

Figure 11.4b shows the job execution time under default and EC-aware chunk placements for Wordcount application. We can see that jobs under EC-aware finish faster and have less variation than those under default placement. For instance, up to 25% reduction in job execution time can be observed under EC-aware placement for 40 GB input size.



(a) EC-aware placement



(b) Default EC placement

Figure 11.5: Tasks of Sort application under default EC and EC-aware placements (non-overlapping shuffle) for 40 GB input size.

This is mainly because the map phase accounts for at least 90% of the job execution time in Wordcount application, compared to Sort application where the map phase accounts only for 65-69%.

Figure 11.7 shows the timeline of task runtimes under both default and EC-aware placements. Map tasks vary by 16% (from 50.04s to 95.31s) with an average of 60.29s under default placement while they vary by 10.3% (from 50.41s to 75.97s) with an average of 57.80s under EC-aware placement. This high variation in tasks' runtimes is also attributed to the variation in data load between the DNs (Similar to Sort application, we have observed the same variation of data load with Wordcount application). For instance, under the default placement, the slowest 10% of the map tasks read at least one of their data chunks from one of the two most overloaded DNs (i.e., serve more than 3 GB).

Contrary to Sort application, in Wordcount application, map phase accounts for the majority of job execution time (i.e., more than 90%), therefore, the straggler map tasks

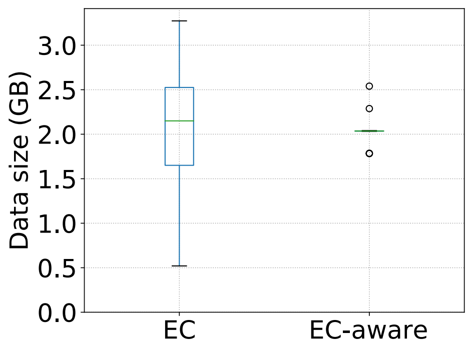


Figure 11.6: Disk read bytes during the map phase of Sort application under default EC and EC-aware placements (non-overlapping shuffle) for 40 GB input size.

have a high impact on the job execution time. EC-aware placement reduces the impact of stragglers as the maximum task runtime is $75.97s$, while under default placement, the slowest 5% of the map tasks have a runtime greater than $77.9s$ and up to $95.31s$.

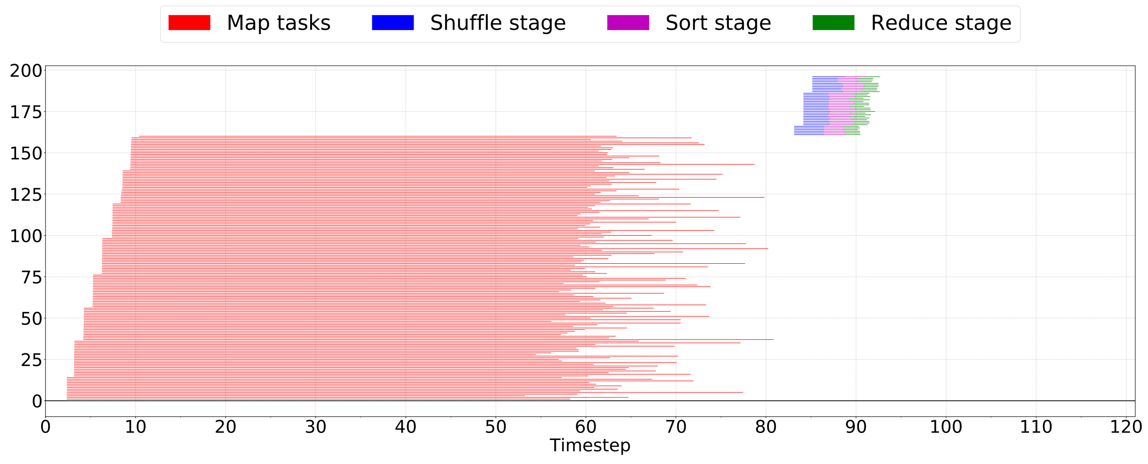
Discussion. Even though EC-aware placement can reduce the variation of map tasks' runtimes, we can still see some long-running tasks. We think this is due to the order of reading data under EC and the disk seek time. However, as perspective, we will investigate in the future some possible solutions such as data prefetching to reduce the impact of random data read from disk.

11.5 Limitations: Towards Dynamic EC-awareness

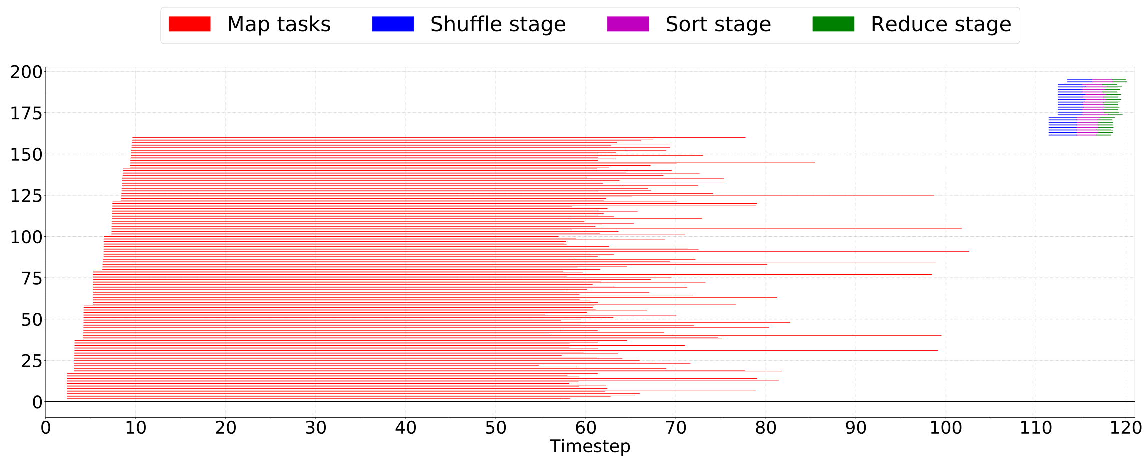
The current placement algorithm can ensure that all the DNs contribute equally to the total amount of data read during the map phase of an application if that application reads the whole data in the cluster. But it can not guarantee balanced data reads within a wave in case of multi-wave jobs, or in case of multiple applications are executed concurrently. To support these scenarios, an online scheduling policy should be applied. For instance, it can decide the set of map tasks to be launched in each wave or by each application in order to balance the data load between the nodes. We discuss that in more detail in the perspectives.

11.6 Conclusion

Erasure codes are increasingly deployed in many storage systems as cost-efficient alternative to replication. In the previous chapter, we have shown that the unawareness of EC in the current version of chunk placement in HDFS results in data skew which impacts the performance of MapReduce applications running under EC. In response, in this chapter, we propose a greedy algorithm that balances the distribution of data chunks between the DNs, which, in turn, reduces jobs execution time. For instance, up to 13% of reduction in job execution time is achieved with Sort application, while with Wordcount application, the reduction of job execution time attains 25%.



(a) EC-aware placement



(b) Default EC placement

Figure 11.7: Tasks of Wordcount application under default EC and EC-aware placements (non-overlapping shuffle) for 40 GB input size.

Part IV

Conclusion and Perspectives

Chapter 12

Conclusion and Perspectives

Contents

We are living in the era of Big Data where data is generated at an unprecedented pace from various sources all over the world. These data can be transformed into meaningful information that has a direct impact on our daily life. To process these Big Data, large-scale and distributed infrastructures are needed. Hence, clouds have been evolving as the de-facto solution for Big Data analytics as they provide an "infinite" pool of resources in a cost-effective manner. Moreover, distributed clouds are leveraged to provide data processing near the source of the data. However, efficient data processing in distributed clouds is facing two major challenges:

1. **Low and unpredictable service provisioning time:** Data processing services and applications in clouds are deployed in virtualized environments such as virtual machines and containers. A service is launched from an image that is large in size and should be locally-available at the compute host. Therefore, provisioning a service in distributed clouds might take a considerable time to complete as the service image might be transferred over the WAN from the image repository to the destination host. Unfortunately, existing works focus on service provisioning inside a single data center, and therefore, the proposed solutions are not adequate for geo-distributed cloud and Edge environments.

2. **High storage cost for data analytics workloads:** backend storage systems of data analytics rely on replication to achieve data availability. Moreover, analytics frameworks leverage replication to achieve data-aware tasks scheduling, thus, improving the performance of data-intensive applications by reducing remote data

access. With the current explosion of Big Data, replication becomes expensive, not only in terms of storage cost (that is particularly important with high-end storage devices such as DRAMs and SSDs) but also in terms of disk access [70] and network traffic [70]. Importantly, the storage overhead of replication always increases with the introduction of new data. And this, in turn, increases constantly the cost of data analytics.

In this thesis, we addressed the aforementioned challenges to achieve efficient Big Data processing in distributed clouds through several contributions that we describe next. Then, we discuss the perspectives that our research opens for Big Data processing in distributed clouds.

12.1 Achievements

12.1.1 Enabling efficient service provisioning in distributed clouds

Clouds provide a large-scale pool of resources and promote fast and agile service deployment. However, service provisioning is a complicated process; it requires complex collaboration between different cloud stacks and involves large network and disk overhead as services' images should be transferred over the network from the image repository to the host server. As clouds are going geographically-distributed, the wide area network between the cloud data centers poses new challenges on the provisioning process. Provisioning services in distributed clouds requires transferring services' images across the expensive and highly heterogeneous wide area network which results in longer provisioning time. In this thesis (Part II), we study data retrieval and placement techniques to improve service (i.e., virtual machine and container) provisioning in a distributed clouds setup.

Network-aware virtual machine image retrieval in geo-distributed clouds

We introduce Nitro, a novel VMI management system that is designed specifically for geographically-distributed clouds to achieve fast service (i.e., VM) provisioning. Different from existing VMI management systems, which ignore the network heterogeneity of WAN, Nitro incorporates two features to reduce the VMI transfer time across geo-distributed data centers. First, it makes use of deduplication to reduce the amount of data which is transferred due to the high similarities within an image and in-between images. Second, Nitro is equipped with a network-aware data transfer strategy to effectively exploit links with high bandwidth when acquiring data and thus expedites the provisioning time. We evaluate Nitro on Grid'5000 [105] testbed by emulating real network topology. Experimental results show that the network-aware data transfer strategy offers the optimal solution when acquiring VMIs while introducing minimal overhead. Moreover, Nitro outperforms state-of-the-art VMI storage system (OpenStack Swift) by up to 77%.

Network-aware container image placement in Edge

Container management in Edge is gaining more importance with the widespread of Edge-servers. To enable fast service provisioning in Edge, we propose to store the container images across these Edge-servers, in a way that the missing layers of an image could be retrieved from nearby Edge-servers. The main goal of this approach is to ensure predictable and reduced service provisioning time. To this end, we present KCBP and KCBP-WC, two container image placement algorithms which aim to reduce the maximum retrieval time of container images to any Edge-server. KCBP and KCBP-WC are based on k-Center optimization. However, KCBP-WC tries to avoid placing large layers of a container image on the same Edge-server. Through extensive simulation, using synthetic and real-world networks with a production container image dataset, we have shown that our proposed algorithm can reduce the maximum provisioning time by 1.1x to 4x compared to Random and Best-Fit based placements, respectively.

12.1.2 Characterizing and improving the performance of data-intensive applications under erasure codes

Data-intensive clusters are heavily relying on distributed storage systems to accommodate the unprecedented amount of data. Hadoop distributed file system (HDFS) [263] is the primary storage for data analytics frameworks such as Spark [20] and Hadoop [19]. Traditionally, HDFS operates under replication to ensure data availability and to allow locality-aware task execution of data-intensive applications. Recently, erasure coding (EC) has emerges as an alternative method to replication in storage systems due to the continuous reduction in its computation overhead. However, data blocks under EC are spread on multiple hosts resulting in remote data access, thus, locality-aware task execution under EC is not fully applicable. In this thesis (Part III), we first characterize and then improve the performance of data-intensive applications in data-intensive clusters under EC.

Experimental evaluation of erasure codes in data-intensive clusters

To understand the performance of data-intensive applications under EC, we conduct an in-depth experimental evaluation of Hadoop on top of the Grid'5000 [105] testbed. We use representative benchmarks to evaluate how data access pattern, concurrent data access, analytics workloads, data persistency, failures, the backend storage devices, and the network configuration impact the performance of MapReduce applications. While some of our results follow our intuition, others were unexpected. For example, disk and network contentions caused by chunks distribution and the unawareness of their functionalities are the main factors affecting the performance of Big Data applications under EC, not data locality.

Integrating EC-awareness in HDFS for higher performance

We observe that Hadoop task scheduler is not aware of the data layout under EC and can result in a noticeable skew in data accesses across servers when running data-intensive applications. This causes stragglers (i.e., some tasks exhibit a large deviation in their executions and take a longer time to complete compared to the average task runtime) and, in turn, prolongs the execution time of data-intensive applications. Accordingly, in an attempt to improve the performance of data-analytics jobs under erasure coding, we propose an EC-aware chunk placement algorithm that balances data accesses across servers by taking into account the semantics of data chunks when distributing them. Our experiments on top of Grid'5000 [105] testbed show that EC-aware placement can reduce the execution time of Sort and Wordcount applications by up to 25%. Our results pave the way and motivate the integration of EC-awareness on the scheduling level to cope with the dynamicity of the environment.

12.2 Perspectives

Our work opens a number of perspectives. In this section, we discuss the most promising ones. We separate these perspectives into two parts: the first part addresses the directions for service provisioning in distributed clouds, while the second one discusses the potential contributions regarding the role of erasure coding in Big Data processing.

12.2.1 Prospects related to service provisioning in distributed clouds

More elaborated network model

In our work on service image provisioning, we assume a complete graph network (as discussed in Section 6.7). That is, each pair of sites are connected with a dedicated link. Under this assumption, parallel retrieval of images to multiple sites produces no interference and it is equivalent to individual single retrievals. It would be interesting to add more constraints on the network model, e.g., putting limits on the uplink/downlink bandwidths of the sites. In that case, concurrent service provisionings to multiple sites have to share the available bandwidth to perform image transfers. To achieve optimal chunk scheduling under the new constraints, the scheduling algorithm of Nitro has to be revisited. Moreover, testing Nitro in real cloud deployment could be interesting to better understand its performance.

Dynamic placement of container images

In our work in Chapter 7, we address the problem of static placement of container images across Edge-servers. By static, we mean that the image dataset, the infrastructure (nodes and network), and the image access patterns (we suppose it is uniform) are known in advance and are not changing over the time. This static approach was essential to provide the base analysis of the problem and hence, to motivate the need for dynamic management.

New container images will be constantly added to the system, while old ones may be removed. On the other hand, Edge-servers may also leave and join the network. Moreover, image access patterns are highly skewed and bursty [17]. To cope with the dynamicity of the environment and optimize the desired objectives, dynamic placement and reconfiguration strategies are required. For instance, placing the complete images on the nodes that are always requesting these images or increasing the replication factor for popular layers may reduce the maximum retrieval time.

Furthermore, joint scheduling of containers and container images could be considered. For example, some use cases do not require the service to be provisioned on a specific Edge-server, but on any (set of) Edge-server(s). This allows choosing the node where the image could be retrieved in a minimal time for instance. Moreover, multi-objective optimization could be employed to optimize not only the maximal retrieval time but also the average retrieval time, total storage, consumed energy, and network traffic.

12.2.2 Prospects related to data analytics under erasure codes

EC-aware and access-aware task scheduling

In Chapter 11, we present how we integrated EC-awareness into HDFS by developing a new chunk placement algorithm. Experimentally, we show that jobs achieve better execution time under this algorithm. However, the current approach that we followed could be improved in two dimensions; First, the static block placement has limited potential in practice as it does not take the dynamic/changing environment into account (e.g., concurrent running jobs, multi-wave jobs, stragglers machines, etc.). This could be overcome by integrating EC-awareness at the scheduler level i.e., leveraging parity chunks for reading the input data. Second, the current placement algorithm can improve the *spatial* load balance between the DNs (i.e., reading the same amount of data from each DNs), however, the number of requests issued to each DN per unit of time is not considered. The skew in data requests over time could lead to a *temporal* load imbalance. To balance the temporal I/O load between DNs, a new access-aware I/O scheduling policy is needed. In addition, the EC-awareness could be also extended to handle heterogeneous environments and improve the speculative execution of tasks.

Enabling data processing under erasure coding in Edge

The current emerging applications in Edge (e.g., video processing) generate large amount of data that becomes unsustainable to be moved to the cloud for processing. These data should be stored across Edge-servers and then processed (collectively). Besides their limited storage capacities, Edge-servers are connected with heterogeneous networks that renders traditional single data center processing approaches suboptimal.

Given the storage reduction brought by EC and its low computation overhead, EC could be an ideal candidate for data processing in Edge. However, EC brings important "high" network overhead. In contrary to replication, where the majority of tasks can run locally, all the tasks under EC have to read most of their input data remotely. Even worse, the cost of data transfer when reading input data and its impact on the performance of data analytics jobs are amplified in Edge environment due to network heterogeneity. Accordingly, as a first step towards realizing EC for data processing in Edge, we empirically demonstrate the impact of network heterogeneity on the execution time of MapReduce applications in a Poster [60]. We found that map tasks under EC suffer from obvious performance degradation (the maximum map task runtime is 3.3x longer compared to the mean) when reading input data from remote nodes, as they have to wait for the last chunk to arrive. Thus, it is important to reduce the impact of network heterogeneity when reading the input data under EC. An interesting direction is to investigate decentralized scheduling policies which try to find to which nodes to retrieve the chunks in order to minimize the maximum retrieving time of the tasks's input data and to allow (data and/or parity) chunks to be proactively pushed to the corresponding tasks. Moreover, it will be interesting to study the impact of using EC to store intermediate data to avoid re-executing map tasks in case of failure.

Bibliography

[1] *5G – high-speed radio waves that can make your portfolio fly*. 2019. URL: https://www.dws.com/en-fr/insights/investment-topics/5g-high-speed-radio-waves-that-can-make-your-portfolio-fly (visited on 07/26/2019).

[2] *63 Fascinating Google Search Statistics*. 2018. URL: https://seotribunal.com/blog/google-stats-and-facts (visited on 07/26/2019).

[3] Michael Abebe, Khuzaima Daudjee, Brad Glasbergen, and Yuanfeng Tian. "EC-Store: Bridging the Gap between Storage and Latency in Distributed Erasure Coded Systems". In: *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 255–266.

[4] Russell L Ackoff. "From data to wisdom". In: *Journal of applied systems analysis* 16.1 (1989), pp. 3–9.

[5] Arif Ahmed and Guillaume Pierre. "Docker-pi: Docker Container Deployment in Fog Computing Infrastructures". In: *International Journal of Cloud Computing* (2019), pp. 1–20.

[6] Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. "Computational investigations of maximum flow algorithms". In: *European Journal of Operational Research* (1997).

[7] *Akamai's state of the internet connectivity report*. 2017. URL: https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-report.pdf (visited on 07/26/2019).

[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM)*. 2008, pp. 63–74.

[9] William Allcock. "GridFTP: Protocol Extensions to FTP for the Grid". In: *Global Grid Forum GFD-R-P.020* (2003).

[10] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. "The Globus Striped GridFTP Framework and Server". In: *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*. 2005.

[11] *Alluxio Homepage*. 2019. URL: https://www.alluxio.io (visited on 07/26/2019).

[12] *Amazon EC2 Pricing*. 2019. URL: https://aws.amazon.com/ec2/pricing/on-demand/ (visited on 07/26/2019).

[13] *Amazon Elastic Container Service*. 2019. URL: https://aws.amazon.com/ecs/ (visited on 07/26/2019).

[14] *Amazon EMR: Easily Run and Scale Apache Spark, Hadoop, HBase, Presto, Hive, and other Big Data Frameworks*. 2019. URL: https://aws.amazon.com/emr/ (visited on 07/26/2019).

[15] *Amazon S3 Glacier: Long-term, secure, durable object storage for data archiving*. 2019. URL: https://aws.amazon.com/glacier (visited on 07/26/2019).

[16] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Disk-locality in Datacenter Computing Considered Irrelevant". In: *Proceedings of the USENIX Workshop on Hot Topics in Operation Systems (HotOS)*. 2011.

[17] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. "Improving Docker Registry Design Based on Production Workload Analysis". In: *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. 2018, pp. 265–278.

[18] *Apache Flink*. 2019. URL: https://flink.apache.org (visited on 07/26/2019).

[19] *Apache Hadoop*. 2019. URL: http://hadoop.apache.org (visited on 07/26/2019).

[20] *Apache Spark*. 2019. URL: https://spark.apache.org (visited on 07/26/2019).

[21] Aaron Archer, Kevin Aydin, Mohammad Hossein Bateni, Vahab Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. "Cache-aware Load Balancing of Data Center Applications". In: *Proceedings of the VLDB Endowment (PVLDB)* 12.6 (2019), pp. 709–723.

[22] *AUFS - Another Union Filesystem*. 2018. URL: http://aufs.sourceforge.net (visited on 07/26/2019).

[23] *AWS Elastic Compute Cloud (EC2)*. 2019. URL: https://aws.amazon.com/ec2 (visited on 07/26/2019).

[24] *AWS Elastic Load Balancing (ELB)*. 2019. URL: https://aws.amazon.com/elasticloadbalancing/ (visited on 07/26/2019).

[25] *AWS Elastic MapReduce (EMR)*. 2019. URL: https://aws.amazon.com/emr/ (visited on 07/26/2019).

[26] *AWS Global Infrastructure*. 2018. URL: https://aws.amazon.com/about-aws/global-infrastructure (visited on 07/26/2019).

[27] *AWS: Regions and Availability Zones*. 2019. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html (visited on 07/26/2019).

[28] *AWS Simple Storage Service (S3)*. 2019. URL: https://aws.amazon.com/s3 (visited on 07/26/2019).

[29] *Azure Blob Storage*. 2019. URL: https://azure.microsoft.com/en-us/services/storage/blobs/ (visited on 07/26/2019).

[30] Victor Bahl. *Emergence of micro datacenter (cloudlets/edges) for mobile computing.* 2015. URL: https://www.microsoft.com/en-us/research/publication/emergence-of-micro-datacenter-cloudlets-edges-for-mobile-computing/ (visited on 07/26/2019).

[31] Ahmet Cihat Baktir, Atay Ozgovde, and Cem Ersoy. "How Can Edge Computing Benefit from Software-Defined Networking: A Survey, Use Cases and Future Directions". In: *IEEE Communications Surveys and Tutorials* PP (June 2017), pp. 1–1.

[32] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. "Pelican: A Building Block for Exascale Cold Data Storage". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI).* 2014, pp. 351–365.

[33] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. "Towards Predictable Datacenter Networks". In: *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM).* 2011, pp. 242–253.

[34] Gaurab Basu, Shripad Nadgowda, and Akshat Verma. "LVD: Lean Virtual Disks". In: *Proceedings of the 15th International Middleware Conference (Middleware).* 2014, pp. 25–36.

[35] Sobir Bazarbayev, Matti Hiltunen, Kaustubh Joshi, William H. Sanders, and Richard Schlichting. "Content-Based Scheduling of Virtual Machines (VMs) in the Cloud". In: *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS).* 2013, pp. 93–101.

[36] Olivier Beaumont, Thomas Lambert, Loris Marchal, and Bastien Thomas. "Data-Locality Aware Dynamic Schedulers for Independent Tasks with Replicated Inputs". In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 2018, pp. 1206–1213.

[37] Paolo Bellavista and Alessandro Zanni. "Feasibility of Fog Computing Deployment Based on Docker Containerization over RaspberryPi". In: *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN).* 2017, 16:1–16:10.

[38] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: *CoRR* abs/1407.3561 (2014). URL: http://arxiv.org/abs/1407.3561.

[39] André B. Bondi. "Characteristics of Scalability and Their Impact on Performance". In: *Proceedings of the 2nd International Workshop on Software and Performance (WOSP).* 2000, pp. 195–203.

[40] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the 1st Edition of the MCC Workshop on Mobile Cloud Computing (MCC).* 2012, pp. 13–16.

[41] Eric A Brewer. "Towards robust distributed systems. abstract". In: *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing (PODC).* 2000, p. 7.

[42] P Chris Broekema, Rob V Van Nieuwpoort, and Henri E Bal. “Exascale high performance computing in the square kilometer array”. In: *Proceedings of the workshop on High-Performance Computing for Astronomy Date*. ACM. 2012, pp. 9–16.

[43] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. “TAO: Facebook’s Distributed Data Store for the Social Graph”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2013, pp. 49–60.

[44] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. “InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services”. In: *Algorithms and Architectures for Parallel Processing*. 2010, pp. 13–31.

[45] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. “Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. 2011, pp. 143–157.

[46] Rodrigo N Calheiros, Rajiv Ranjan, and Rajkumar Buyya. “Virtual machine provisioning based on analytical performance and QoS in cloud computing environments”. In: *Proceedings of the International Conference on Parallel Processing (ICPP)*. 2011, pp. 295–304.

[47] Li Chen, Shuhao Liu, Baochun Li, and Bo Li. “Scheduling jobs across geo-distributed datacenters with max-min fairness”. In: *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. 2017, pp. 1–9.

[48] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. “RAID: High-performance, Reliable Secondary Storage”. In: *ACM Computing Surveys (CSUR)* 26.2 (1994), pp. 145–185.

[49] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. “Energy efficiency for large-scale mapreduce workloads with significant interactive analysis”. In: *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. ACM. 2012, pp. 43–56.

[50] Yanpei Chen, Sara Alspaugh, and Randy Katz. “Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads”. In: *Proceedings of the VLDB Endowment (PVLDB)* 5.12 (2012), pp. 1802–1813.

[51] Yanpei Chen, Sara Alspaugh, and Randy H Katz. *Design insights for MapReduce from diverse production workloads*. Tech. rep. CALIFORNIA UNIV BERKELEY, 2012.

[52] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. "Giza: Erasure Coding Objects across Global Data Centers". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2017, pp. 539–551.

[53] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. "Leveraging Endpoint Flexibility in Data-Intensive Clusters". In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. 2013, pp. 231–242.

[54] *Cloud Bigtable: A petabyte-scale, fully managed NoSQL database service for large analytical and operational workloads*. 2019. URL: https://cloud.google.com/bigtable/ (visited on 07/26/2019).

[55] *CMU Hadoop traces*. 2013. URL: https://www.pdl.cmu.edu/HLA/ (visited on 07/26/2019).

[56] Bram Cohen. *Incentives build robustness in bittorrent*. 2003. URL: http://www.bittorrent.org/bittorrentecon.pdf (visited on 07/26/2019).

[57] *Configure back ends for OpenStack Galnce*. 2019. URL: https://docs.openstack.org/mitaka/config-reference/image-service/backends.html (visited on 07/26/2019).

[58] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. "Spanner: Google's Globally Distributed Database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[59] Paul Covington, Jay Adams, and Emre Sargin. "Deep neural networks for youtube recommendations". In: *Proceedings of the 10th ACM conference on recommender systems*. ACM. 2016, pp. 191–198.

[60] Jad Darrous and Shadi Ibrahim. *Enabling Data Processing under Erasure Coding in the Fog*. In the 48th International Conference on Parallel Processing (ICPP); Available online at. 2019.

[61] Jad Darrous, Shadi Ibrahim, and Christian Perez. "Is it time to revisit Erasure Coding in Data-intensive clusters?" In: *Proceedings of the 27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2019, pp. 165–178.

[62] Jad Darrous, Shadi Ibrahim, Amelie Chi Zhou, and Christian Perez. "Nitro: Network-Aware Virtual Machine Image Management in Geo-Distributed Clouds". In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2018, pp. 553–562.

[63] Jad Darrous, Thomas Lambert, and Shadi Ibrahim. "On the Importance of container images placement for service provisioning in the Edge". In: *Proceedings of the 28th International Conference on Computer Communications and Networks (ICCCN)*. 2019, pp. 1–9.

[64] *Data Lake: A no-limits data lake to power intelligent action*. 2019. URL: https://azure.microsoft.com/en-us/solutions/data-lake/ (visited on 07/26/2019).

[65] *Data preservation: Preserving data for future generations*. 2019. URL: https://home.cern/science/computing/data-preservation (visited on 07/26/2019).

[66] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*. Vol. 6. 2004, pp. 10–10.

[67] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. 2007, pp. 205–220.

[68] Tharam Dillon, Chen Wu, and Elizabeth Chang. "Cloud computing: issues and challenges". In: *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*. 2010, pp. 27–33.

[69] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. "Network Coding for Distributed Storage Systems". In: *IEEE Transactions on Information Theory* 56.9 (Sept. 2010), pp. 4539–4551.

[70] Florin Dinu and T.S. Eugene Ng. "RCMP: Enabling Efficient Recomputation Based Failure Resilience for Big Data Analytics". In: *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS)*. 2014, pp. 962–971.

[71] Florin Dinu and T.S. Eugene Ng. "Understanding the Effects and Implications of Compute Node Related Failures in Hadoop". In: *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 2012, pp. 187–198.

[72] *Docker homepage*. 2018. URL: https://www.docker.com (visited on 07/26/2019).

[73] *Docker Hub*. 2019. URL: https://hub.docker.com (visited on 07/26/2019).

[74] *Docker Hub Image Index*. 2019. URL: https://hub.docker.com/search?type=image (visited on 07/26/2019).

[75] Lian Du, Tianyu Wo, Renyu Yang, and Chunming Hu. "Cider: A rapid docker container deployment system through sharing network storage". In: *Proceedings of the IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2017, pp. 332–339.

[76] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. "MapReduce for Data Intensive Scientific Analyses". In: *Proceedings of the IEEE Fourth International Conference on eScience*. 2008, pp. 277–284.

[77] Yehia Elkhatib, Barry Porter, Heverson B Ribeiro, Mohamed Faten Zhani, Junaid Qadir, and Etienne Rivière. "On Using Micro-Clouds to Deliver the Fog". In: *IEEE Internet Computing* 21.2 (2017), pp. 8–15.

[78] *Erasure Code Support in OpenStack Swift*. URL: https://docs.openstack.org/swift/pike/overview_erasure_code.html (visited on 07/26/2019).

[79] *Eucalyptus*. 2018. URL: https://www.eucalyptus.cloud (visited on 07/26/2019).

[80] *Executive Summary: Data Growth, Business Opportunities, and the IT Imperatives*. 2014. URL: https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm (visited on 07/26/2019).

[81] *Facebook puts 10,000 Blu-ray discs in low-power storage system*. 2014. URL: https://www.itworld.com/article/2831281/facebook-puts-10-000-blu-ray-discs-in-low-power-storage-system.html (visited on 07/26/2019).

[82] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. "DiskReduce: RAID for Data-intensive Scalable Computing". In: *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. 2009, pp. 6–10.

[83] Juntao Fang, Shenggang Wan, and Xubin He. "RAFI: Risk-Aware Failure Identification to Improve the RAS in Erasure-coded Data Centers". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2018, pp. 495–506.

[84] *Fast Data Transfer - FDT*. 2019. URL: https://fast-data-transfer.github.io (visited on 07/26/2019).

[85] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. "An updated performance comparison of virtual machines and linux containers". In: *Proceedings of the IEEE international symposium on performance analysis of systems and software (ISPASS)*. 2015, pp. 171–172.

[86] A. Fikes. *Colossus, successor to Google File System*. 2010. URL: http://web.archive.org/web/20160324185413/http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf (visited on 07/26/2019).

[87] *File Transfer Protocol (FTP)*. 2019. URL: https://en.wikipedia.org/wiki/File_Transfer_Protocol (visited on 07/26/2019).

[88] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. "Availability in Globally Distributed Storage Systems". In: *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)*. 2010, pp. 61–74.

[89] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[90] *G Suite*. 2019. URL: https://gsuite.google.com (visited on 07/26/2019).

[91] John Gantz and David Reinsel. *Extracting Value from Chaos*. 2011. URL: http://nfic-2016.ieeesiliconvalley.org/wp-content/uploads/sites/16/2016/05/NFIC-2016-IBM-Jeff-Welser.pdf (visited on 07/26/2019).

[92] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "Network Requirements for Resource Disaggregation". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 249–264.

[93] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. "Edge-centric Computing: Vision and Challenges". In: 45.5 (2015).

[94] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. 2002.

[95] Simson Garfinkel. *Architects of the information society: 35 years of the Laboratory for Computer Science at MIT*. MIT press, 1999.

[96] Peter Garraghan, Paul Townend, and Jie Xu. "An Empirical Failure-Analysis of a Large-Scale Cloud Computing Environment". In: *Proceedings of the 15th IEEE International Symposium on High-Assurance Systems Engineering*. 2014, pp. 113–120.

[97] Frank Gens. *Clouds and Beyond: Positioning for the Next 20 Years in Enterprise IT*. 2009. URL: https://www.slideshare.net/innoforum09/gens (visited on 07/26/2019).

[98] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*. 2003, pp. 29–43.

[99] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. "Firmament: Fast, Centralized Cluster Scheduling at Scale". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 99–115.

[100] *Google Cloud Platform: Regions and Zones*. 2019. URL: https://cloud.google.com/compute/docs/regions-zones/ (visited on 07/26/2019).

[101] *Google Datacenter Locations*. 2018. URL: https://www.google.com/about/datacenters/inside/locations/index.html (visited on 07/26/2019).

[102] *Google: 'EVERYTHING at Google runs in a container'*. 2014. URL: https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion (visited on 07/26/2019).

[103] *Google Kubernetes Engine*. 2019. URL: https://cloud.google.com/kubernetes-engine/ (visited on 07/26/2019).

[104] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, Dave Maltz, Parveen Patel, and Sudipta Sengupta. "VL2: A Scalable and Flexible Data Center Network". In: *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM)*. Vol. 39. 4. Recognized as one of "the most important research results published in CS in recent years". 2009, pp. 51–62.

[105] *Grid'5000*. URL: https://www.grid5000.fr (visited on 07/26/2019).

[106] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. "Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages". In: *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*. 2016, pp. 1–16.

[107] Kiryong Ha, Padmanabhan Pillai, Grace Lewis, Soumya Simanta, Sarah Clinch, Nigel Davies, and Mahadev Satyanarayanan. "The impact of mobile multimedia applications on data center consolidation". In: *Proceedings of the IEEE international conference on cloud engineering (IC2E)*. 2013, pp. 166–176.

[108] Andreas Haeberlen, Alan Mislove, and Peter Druschel. “Glacier: Highly durable, decentralized storage despite massive correlated failures”. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI)*. Vol. 2. 2005, pp. 143–158.

[109] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “Slacker: Fast Distribution with Lazy Docker Containers”. In: *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*. 2016, pp. 181–195.

[110] *HDFS Erasure Coding*. URL: https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html (visited on 07/26/2019).

[111] *HDFS-RAID wiki*. 2011. URL: https://wiki.apache.org/hadoop/HDFS-RAID (visited on 07/26/2019).

[112] *HDInsight: Easy, cost-effective, enterprise-grade service for open source analytics*. 2019. URL: https://azure.microsoft.com/en-us/services/hdinsight/ (visited on 07/26/2019).

[113] Anthony JG Hey, Stewart Tansley, Kristin M Tolle, et al. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Vol. 1. Microsoft research Redmond, WA, 2009.

[114] Cheol-Ho Hong and Blesson Varghese. “Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms”. In: *ACM Comput. Surv.* 52.5 (Sept. 2019), 97:1–97:37.

[115] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. “Achieving High Utilization with Software-driven WAN”. In: *Proceedings of the ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. 2013, pp. 15–26.

[116] *How many bytes for...* 2008. URL: https://searchstorage.techtarget.com/definition/How-many-bytes-for (visited on 07/26/2019).

[117] *How Much Data is Produced Every Day?* 2016. URL: https://www.northeastern.edu/levelblog/2016/05/13/how-much-data-produced-every-day (visited on 07/26/2019).

[118] *How Much Information? 2003*. 2003. URL: http://groups.ischool.berkeley.edu/archive/how-much-info-2003/execsum.htm (visited on 07/26/2019).

[119] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. “Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds”. In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2017, pp. 629–647.

[120] Wen-Lian Hsu and George L Nemhauser. “Easy and hard bottleneck location problems”. In: *Discrete Applied Mathematics* 1.3 (1979), pp. 209–215.

[121] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. "Survey on fog computing: architecture, key technologies, applications and open issues". In: *Journal of Network and Computer Applications* (2017).

[122] Zhiming Hu, Baochun Li, and Jun Luo. "Flutter: Scheduling Tasks Closer to Data Across Geo-Distributed Datacenters". In: *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*. 2016, pp. 1–9.

[123] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. "Erasure Coding in Windows Azure Storage". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2012, pp. 15–26.

[124] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis". In: *Proceedings of the 26th IEEE International Conference on Data Engineering Workshops (ICDEW 2010)*. 2010, pp. 41–51.

[125] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. "VideoEdge: Processing Camera Streams using Hierarchical Clusters". In: *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*. 2018, pp. 115–131.

[126] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. "Wide-area Analytics with Multiple Resources". In: *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*. 2018, 12:1–12:16.

[127] Shadi Ibrahim, Bingsheng He, and Hai Jin. "Towards Pay-as-you-consume Cloud Computing". In: *Proceedings of the IEEE International Conference on Services Computing (SCC)*. IEEE. 2011, pp. 370–377.

[128] Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Gabriel Antoniu, and Song Wu. "Maestro: Replica-Aware Map Scheduling for MapReduce". In: *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2012, pp. 435–442.

[129] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud". In: *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2010, pp. 17–24.

[130] *IDC: Expect 175 zettabytes of data worldwide by 2025*. 2018. URL: https://www.networkworld.com/article/3325397/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html (visited on 07/26/2019).

[131] *Intel Intelligent Storage Acceleration Library Homepage*. 2019. URL: https://software.intel.com/en-us/storage/ISA-L (visited on 07/26/2019).

[132] *IPFS github Homepage*. 2019. URL: https://github.com/ipfs/go-ipfs (visited on 07/26/2019).

[133] *ISA-L Performance report*. 2017. URL: https://01.org/sites/default/files/documentation/intel_isa-l_2.19_performance_report_0.pdf (visited on 07/26/2019).

[134] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: Fair Scheduling for Distributed Computing Clusters". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. 2009, pp. 261–276.

[135] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. "Evaluation of docker as edge computing platform". In: *Proceedings of the IEEE International Confernece on Open Systems (ICOS)*. 2015, pp. 130–135.

[136] Navendu Jain and Rahul Potharaju. "When the Network Crumbles: An Empirical Study of Cloud Network Failures and their Impact on Services". In: *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC)*. 2013, p. 15.

[137] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. "B4: Experience with a Globally-deployed Software Defined WAN". In: *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM)*. 2013, pp. 3–14.

[138] K. R. Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei. "An Empirical Analysis of Similarity in Virtual Machine Images". In: *Proceedings of the Middleware Industry Track Workshop (Middleware)*. 2011, p. 6.

[139] Hai Jin, Shadi Ibrahim, Tim Bell, Wei Gao, Dachuan Huang, and Song Wu. "Cloud Types and Services". In: *Handbook of Cloud Computing*. Springer, 2010, pp. 335–355.

[140] Hai Jin, Shadi Ibrahim, Tim Bell, Li Qi, Haijun Cao, Song Wu, and Xuanhua Shi. "Tools and Technologies for Building Clouds". In: *Cloud Computing*. Springer, 2010, pp. 3–20.

[141] Hai Jin, Shadi Ibrahim, Li Qi, Haijun Cao, Song Wu, and Xuanhua Shi. "The MapReduce Programming Model and Implementations". In: *Cloud Computing: Principles and Paradigms* (2011), pp. 373–390.

[142] Keren Jin and Ethan L. Miller. "The Effectiveness of Deduplication on Virtual Machine Disk Images". In: *Proceedings of the SYSTOR*. 2009, p. 7.

[143] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. "Occupy the Cloud: Distributed Computing for the 99%". In: *Proceedings of the Symposium on Cloud Computing (SoCC)*. 2017, pp. 445–451.

[144] *Just how big is Amazon's AWS business? (hint: it's absolutely massive)*. 2014. URL: https://www.geek.com/chips/just-how-big-is-amazons-aws-business-hint-its-absolutely-massive-1610221 (visited on 07/26/2019).

[145] *Just one autonomous car will use 4,000 GB of data/day*. 2016. URL: https://www.networkworld.com/article/3147892/one-autonomous-car-will-use-4000-gb-of-dataday.html (visited on 07/26/2019).

[146] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. "The Nature of Data Center Traffic: Measurements and Analysis". In: *Proceedings of the Internet Measurement Conference*. 2009, pp. 202–208.

[147] Jussi Kangasharju, James Roberts, and Keith W Ross. "Object replication strategies in content distribution networks". In: *Computer Communications* 25.4 (2002), pp. 376–383.

[148] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. "FID: A Faster Image Distribution System for Docker Platform". In: *Proceedings of the 2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 2017, pp. 191–198.

[149] Alexei Karve and Andrzej Kochut. "Redundancy Aware Virtual Disk Mobility for Cloud Computing". In: *Proceedings of the IEEE 6th International Conference on Cloud Computing (CLOUD)*. 2013, pp. 35–42.

[150] *Kernel Virtual Machine (KVM)*. 2019. URL: https://www.linux-kvm.org/page/Main_Page (visited on 07/26/2019).

[151] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads". In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. 2012, p. 20.

[152] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, Rajkumar Kettimuthu, P Sadayappan, Ian Foster, and Joel Saltz. "Using Overlays For Efficient Data Transfer Over Shared Wide-Area Networks". In: *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*. 2008, p. 47.

[153] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, Rajkumar Kettimuthu, P Sadayappan, and Joel Saltz. "A Dynamic Scheduling Approach for Coordinated Wide-Area Data Transfers using GridFTP". In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 2008, pp. 1–12.

[154] Nakku Kim, Jungwook Cho, and Euiseong Seo. "Energy-credit scheduler: An energy-aware virtual machine scheduler for cloud systems". In: *Future Generation Computer Systems (FGCS)* 32 (2014), pp. 128 –137.

[155] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. "OSv—Optimizing the Operating System for Virtual Machines". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2014, pp. 61–72.

[156] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. "Pixida: Optimizing Data Parallel Jobs in Wide-area Data Analytics". In: *Proceedings of the VLDB Endowment (PVLDB)* 9.2 (Oct. 2015), pp. 72–83.

[157] Andrzej Kochut and Alexei Karve. "Leveraging local image redundancy for efficient virtual machine provisioning". In: *Proceedings of the IEEE Network Operations and Management Symposium (NOMS)*. 2012, pp. 179–187.

[158] KR Krish, Ali Anwar, and Ali R Butt. “hatS: A Heterogeneity-Aware Tiered Storage for Hadoop”. In: *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2014, pp. 502–511.

[159] *Kubernetes homepage*. 2018. URL: https://kubernetes.io (visited on 07/26/2019).

[160] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. “OceanStore: An Architecture for Global-scale Persistent Storage”. In: *SIGPLAN Not.* 35.11 (2000), pp. 190–201.

[161] Pradeep Kumar and H. Howie Huang. “Falcon: Scaling IO Performance in Multi-SSD Volumes”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2017, pp. 41–53.

[162] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. “Singularity: Scientific containers for mobility of compute”. In: *PLOS ONE* 12 (May 2017), pp. 1–20.

[163] Tobias Kurze, Markus Klems, David Bermbach, Alexander Lenk, Stefan Tai, and Marcel Kunze. “Cloud federation”. In: *Cloud Computing* 2011 (2011), pp. 32–38.

[164] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. “SkewTune: Mitigating Skew in Mapreduce Applications”. In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2012, pp. 25–36.

[165] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. “Atlas: Baidu’s key-value storage system for cloud data”. In: *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 2015, pp. 1–14.

[166] Avinash Lakshman and Prashant Malik. “Cassandra: a Decentralized Structured Storage System”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[167] Doug Laney. “3D data management: Controlling data volume, velocity and variety”. In: *META group research note* 6.70 (2001), p. 1.

[168] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. “Identifying hot and cold data in main-memory databases”. In: *Proceedings of the 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 26–37.

[169] *Leveldb github Homepage*. 2019. URL: https://github.com/google/leveldb (visited on 07/26/2019).

[170] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. “Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks”. In: *Proceedings of the 5th annual Symposium on Cloud Computing (SoCC)*. 2014.

[171] Jun Li and Baochun Li. “On Data Parallelism of Erasure Coding in Distributed Storage Systems”. In: *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 45–56.

[172] Jun Li and Baochun Li. “Parallelism-Aware Locally Repairable Code for Distributed Storage Systems”. In: *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 87–98.

[173] Runhui Li, Patrick PC Lee, and Yuchong Hu. “Degraded-First Scheduling for MapReduce in Erasure-Coded Storage Clusters”. In: *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2014, pp. 419–430.

[174] *libtorrent Homepage*. 2019. URL: https://libtorrent.org (visited on 07/26/2019).

[175] Anthony Liguori and Eric Van Hensbergen. “Experiences with Content Addressable Storage and Virtual Disks”. In: *Proceedings of the 1st Conference on I/O Virtualization (WIOV)*. 2008, pp. 5–5.

[176] Ching-Chi Lin, Pangfeng Liu, and Jan-Jan Wu. “Energy-Aware Virtual Machine Dynamic Provision and Scheduling for Cloud Computing”. In: *Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD)*. 2011, pp. 736–737.

[177] Chuan Lin, Yuanguo Bi, Guangjie Han, Jucheng Yang, Hai Zhao, and Zheng Liu. “Scheduling for Time-Constrained Big-File Transfer Over Multiple Paths in Cloud Computing”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 2.1 (2018), pp. 25–40.

[178] *Linux scp command*. 2019. URL: https://www.computerhope.com/unix/scp.htm (visited on 07/26/2019).

[179] *Linux Traffic Control*. 2006. URL: https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html (visited on 07/26/2019).

[180] Haikun Liu, Bingsheng He, Xiaofei Liao, and Hai Jin. “Towards Declarative and Data-centric Virtual Machine Image Management in IaaS Clouds”. In: *IEEE Transactions on Cloud Computing* (2017).

[181] Wantao Liu, Brian Tieman, Rajkumar Kettimuthu, and Ian Foster. “A Data Transfer Framework for Large-Scale Science Experiments”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*. 2010, pp. 717–724.

[182] Xiaoyi Lu, Nusrat S Islam, Md Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K Panda. “High-Performance Design of Hadoop RPC with RDMA over InfiniBand”. In: *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*. 2013, pp. 641–650.

[183] Michael Luby, Roberto Padovani, Thomas J. Richardson, Lorenz Minder, and Pooja Aggarwal. “Liquid Cloud Storage”. In: *ACM Trans. Storage* 15.1 (2019), 2:1–2:49.

[184] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*. Vol. 16. Elsevier, 1977.

[185] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. “Unikernels: Library operating systems for the cloud”. In: *ACM SIGPLAN Notices* 48.4 (2013), pp. 461–472.

[186] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. "My VM is Lighter (and Safer) than your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 2017, pp. 218–233.

[187] *MapReduce and Hadoop Algorithms in Academic Papers*. 2011. URL: http://atbrox.com/2011/05/16/mapreduce-hadoop-algorithms-in-academic-papers-4th-update-may-2011/ (visited on 07/26/2019).

[188] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. "ClickOS and the Art of Network Function Virtualization". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014, pp. 459–473.

[189] Mark McLoughlin. *The QCOW2 Image Format*. URL: https://people.gnome.org/~markmc/qcow-image-format.html (visited on 07/26/2019).

[190] *Microsoft 365*. 2019. URL: https://www.microsoft.com/en-us/microsoft-365 (visited on 07/26/2019).

[191] *Microsoft Hyper-V*. 2019. URL: https://microsoft.fandom.com/wiki/Hyper-V (visited on 07/26/2019).

[192] *Microsoft VHD Image Format*. URL: https://blogs.technet.microsoft.com/fabricem_blogs/2005/10/16/virtual-hard-disk-image-format-specification (visited on 07/26/2019).

[193] Sangwhan Moon, Jaehwan Lee, and Yang Suk Kee. "Introducing SSDs to the Hadoop MapReduce Framework". In: *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD)*. 2014, pp. 272–279.

[194] Nitesh Mor. *Edge Computing: Scaling resources within multiple administrative domains*. 2019. URL: https://queue.acm.org/detail.cfm?id=3313377 (visited on 07/26/2019).

[195] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. "f4: Facebook's Warm BLOB Storage System". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 383–398.

[196] Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. "Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2006, pp. 6–6.

[197] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. "CoMICon: A Co-Operative Management System for Docker Container Images". In: *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. 2017, pp. 116–126.

[198] *Netflix Case Study*. 2016. URL: https://aws.amazon.com/solutions/case-studies/netflix (visited on 07/26/2019).

[199] Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P. C. Lee, and John C. S. Lui. “Live Deduplication Storage of Virtual Machine Images in an Open-source Cloud”. In: *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware (Middleware)*. 2011, pp. 81–100.

[200] Thuy Linh Nguyen, Ramon Nou, and Adrien Lebre. “YOLO: Speeding Up VM and Docker Boot Time by Reducing I/O Operations”. In: *Proceedings of the 25rd International European Conference on Parallel and Distributed Computing (Euro-Par)*. 2019, pp. 273–287.

[201] Salman Niazi, Mikael Ronström, Seif Haridi, and Jim Dowling. “Size Matters: Improving the Performance of Small Files in Hadoop”. In: *Proceedings of the 19th International Middleware Conference (Middleware)*. 2018, pp. 26–39.

[202] Bogdan Nicolae, Andrzej Kochut, and Alexei Karve. “Discovering and Leveraging Content Similarity to Optimize Collective On-Demand Data Access to IaaS Cloud Storage”. In: *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2015, pp. 211–220.

[203] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. “Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter”. In: *Proceedings of the 13th EuroSys Conference (EuroSys)*. 2018, p. 16.

[204] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. “The Eucalyptus open-source cloud-computing system”. In: *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE Computer Society. 2009, pp. 124–131.

[205] *Object replication in Swift Ocata*. 2019. URL: https://docs.openstack.org/ocata/admin-guide/objectstorage-replication.html (visited on 07/26/2019).

[206] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. “A Binary-compatible Unikernel”. In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM. 2019, pp. 59–73.

[207] *OpenFog Reference Architecture for Fog Computing*. 2017. URL: https://iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf (visited on 07/26/2019).

[208] *OpenNebula: Open Source Data Center Virtualization*. 2018. URL: https://www.opennebula.org (visited on 07/26/2019).

[209] *OpenStack*. 2018. URL: https://www.openstack.org (visited on 07/26/2019).

[210] *OpenStack Storage (Swift)*. 2018. URL: https://github.com/openstack/swift (visited on 07/26/2019).

[211] *opentracker Homepage*. 2019. URL: https://erdgeist.org/arts/software/opentracker/ (visited on 07/26/2019).

[212] *Overlay Filesystem*. 2018. URL: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt (visited on 07/26/2019).

[213] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. “The Quantcast File System”. In: *Proceedings of the VLDB Endowment (PVLDB)* 6.11 (2013), pp. 1092–1101.

[214] *Oz Github repository*. 2019. URL: https://github.com/clalancette/oz/wiki (visited on 07/26/2019).

[215] *Packer: Build Automated Machine Images*. 2019. URL: https://www.packer.io (visited on 07/26/2019).

[216] Lluis Pamies-Juarez, Filip Blagojević, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandić. “Opening the Chrysalis: On the Real Repair Performance of MSR Codes”. In: *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*. 2016, pp. 81–94.

[217] Ripon Patgiri and Arif Ahmed. “Big Data: The V’s of the Game Changer Paradigm”. In: *Proceedings of the IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2016, pp. 17–24.

[218] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. “A comparison of approaches to large-scale data analysis”. In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2009, pp. 165–178.

[219] Chunyi Peng, Minkyong Kim, Zhe Zhang, and Hui Lei. “VDN: Virtual Machine Image Distribution Network for Cloud Data Centers”. In: *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. 2012, pp. 181–189.

[220] Tien-Dat Phan, Shadi Ibrahim, Amelie Chi Zhou, Guillaume Aupy, and Gabriel Antoniu. “Energy-Driven Straggler Mitigation in MapReduce”. In: *Proceedings of the 23rd International European Conference on Parallel and Distributed Computing (Euro-Par)*. 2017, pp. 385–398.

[221] Ilia Pietri and Rizos Sakellariou. “Mapping virtual machines onto physical machines in cloud computing: A survey”. In: *ACM Computing Surveys (CSUR)* 49.3 (2016), p. 49.

[222] *Powered by Apache Hadoop*. 2018. URL: https://cwiki.apache.org/confluence/display/HADOOP2/PoweredBy (visited on 07/26/2019).

[223] *Powered by Flink*. 2019. URL: https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink (visited on 07/26/2019).

[224] *Powered by Spark*. 2019. URL: https://spark.apache.org/powered-by.html (visited on 07/26/2019).

[225] *psutil: Cross-platform lib for process and system monitoring in Python*. 2019. URL: https://github.com/giampaolo/psutil (visited on 07/26/2019).

[226] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. "Low Latency Geo-distributed Data Analytics". In: *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. 2015, pp. 421–434.

[227] Lili Qiu, Venkata N Padmanabhan, and Geoffrey M Voelker. "On the placement of Web server replicas". In: *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. Vol. 3. 2001, pp. 1587–1596.

[228] Flavien Quesnel, Adrien Lèbre, and Mario Südholt. "Cooperative and reactive scheduling in large-scale virtualized platforms with DVMS". In: *Concurrency and Computation: Practice and Experience* 25.12 (2013), pp. 1643–1655.

[229] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. "Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014, pp. 275–288.

[230] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. "EC-cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 401–417.

[231] K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. "Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage and Network-bandwidth". In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. 2015, pp. 81–94.

[232] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster". In: *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 2013.

[233] K.V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. "Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth". In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. 2015, pp. 81–94.

[234] K.V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. "A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers". In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. 2014, pp. 331–342.

[235] Kaveh Razavi, Ana Ion, and Thilo Kielmann. "Squirrel: Scatter Hoarding VM Image Contents on IaaS Compute Nodes". In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*. 2014, pp. 265–278.

[236] Kaveh Razavi and Thilo Kielmann. "Scalable Virtual Machine Deployment Using VM Image Caches". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 2013, 65:1–65:12.

[237] *Real-time Video Analytics: the killer app for edge computing*. 2018. URL: https://www.microsoft.com/en-us/research/publication/real-time-video-analytics-killer-app-edge-computing/ (visited on 07/26/2019).

[238] *Redis*. 2018. URL: https://redis.io (visited on 07/26/2019).

[239] Irving Reed and Golomb Solomon. "Polynomial codes over certain finite fields". In: *Journal of the Society of Industrial and Applied Mathematics* 8.2 (1960), 300–304.

[240] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. "VMTorrent: Scalable P2P Virtual Machine Streaming". In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. 2012, pp. 289–300.

[241] Darrell Reimer, Arun Thomas, Glenn Ammons, Todd Mummert, Bowen Alpern, and Vasanth Bala. "Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl". In: *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 2008, pp. 111–120.

[242] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. "Hopper: Decentralized speculation-aware cluster scheduling at scale". In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. ACM. 2015, pp. 379–392.

[243] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. "Data-Driven Stream Processing at the Edge". In: *Proceedings of the 1st IEEE International Conference on Fog and Edge Computing (ICFEC)*. 2017, pp. 31–40.

[244] *Report: AWS Market Share Is Triple Azure's*. 2017. URL: https://awsinsider.net/articles/2017/08/01/aws-market-share-3x-azure.aspx (visited on 07/26/2019).

[245] Luigi Rizzo. "Effective erasure codes for reliable computer communication protocols". In: *ACM SIGCOMM Computer Communication Review* 27.2 (1997), pp. 24–36.

[246] Borut Robič and Jurij Mihelič. "Solving the k-center problem efficiently with a dominating set algorithm". In: *Journal of computing and information technology* 13.3 (2005), pp. 225–234.

[247] Rodrigo Rodrigues and Barbara Liskov. "High Availability in DHTs: Erasure Coding vs. Replication". In: *Proceedings of the Peer-to-Peer Systems IV*. 2005.

[248] *rsync home page*. 2019. URL: https://rsync.samba.org (visited on 07/26/2019).

[249] Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. "Reconstructable Software Appliances with Kameleon". In: *SIGOPS Operating Systems Review* 49.1 (Jan. 2015), pp. 80–89.

[250] Philip Russom. *Big Data Analytics*. 2011. URL: https://vivomente.com/wp-content/uploads/2016/04/big-data-analytics-white-paper.pdf (visited on 07/26/2019).

[251] Hadi Salimi, Mahsa Najafzadeh, and Mohsen Sharifi. "Advantages, Challenges and Optimizations of Virtual Machine Scheduling in Cloud Computing Environments". In: *International Journal of Computer Theory and Engineering* 4.2 (2012), p. 189.

[252] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. “XORing Elephants: Novel Erasure Codes for Big Data”. In: *Proceedings of the VLDB Endowment (PVLDB)* 6.5 (2013), pp. 325–336.

[253] Mahadev Satyanarayanan, Victor Bahl, Ramón Caceres, and Nigel Davies. “The Case for VM-Based Cloudlets in Mobile Computing”. In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23.

[254] Roland Schwarzkopf, Matthias Schmidt, Mathias Rüdiger, and Bernd Freisleben. “Efficient Storage of Virtual Machine Images”. In: *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date (ScienceCloud)*. 2012, pp. 51–60.

[255] *Scientists will use a supercomputer to simulate the universe*. 2019. URL: https://futurism.com/the-byte/scientists-supercomputer-simulate-universe (visited on 07/26/2019).

[256] Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K Panda. “High-Performance and Resilient Key-Value Store with Online Erasure Coding for Big Data Workloads”. In: *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 527–537.

[257] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. “Containers and Virtual Machines at Scale: A Comparative Study”. In: *Proceedings of the 17th International Middleware Conference (Middleware)*. 2016, 1:1–1:13.

[258] Zhiming Shen, Zhe Zhang, Andrzej Kochut, Alexei Karve, Han Chen, Minkyong Kim, Hui Lei, and Nicholas Fuller. “VMAR: Optimizing I/O performance and resource utilization in the cloud”. In: *Proceedings of the ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. 2013, pp. 183–203.

[259] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. “Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics”. In: *Proceedings of the VLDB Endowment (PVLDB)* 8.13 (2015).

[260] Weiming Shi and Bo Hong. “Towards Profitable Virtual Machine Placement in the Data Center”. In: *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC)*. 2011, pp. 138–145.

[261] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. “Edge computing: Vision and challenges”. In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

[262] Weisong Shi and Schahram Dustdar. “The Promise of Edge Computing”. In: *Computer* 49.5 (2016), pp. 78–81.

[263] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. “The Hadoop Distributed File System”. In: *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. Vol. 10. 2010, pp. 1–10.

[264] Dimitris Skourtis, Lukas Rupprecht, Vasily Tarasov, and Nimrod Megiddo. “Carving perfect layers out of Docker images”. In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2019.

[265] James E Smith and Ravi Nair. “The architecture of virtual machines”. In: *Computer* 38.5 (2005), pp. 32–38.

[266] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. “Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors”. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. 2007, pp. 275–287.

[267] Michael Stonebraker. *What Does 'big Data' Mean?* 2012. URL: https://cacm.acm.org/blogs/blog-cacm/155468-what-does-big-data-mean/fulltext (visited on 07/26/2019).

[268] *Swarm mode overview*. 2019. URL: https://docs.docker.com/engine/swarm/ (visited on 07/26/2019).

[269] Chunqiang Tang. “FVD: A High-performance Virtual Machine Image Format for Cloud”. In: *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*. 2011, pp. 18–18.

[270] *tcconfig Homepage*. 2019. URL: http://tcconfig.readthedocs.io/en/latest/pages/introduction/index.html (visited on 07/26/2019).

[271] *The Internet Topology Zoo*. 2019. URL: https://www.topology-zoo.org (visited on 07/26/2019).

[272] *The NIST Definition of Cloud Computing*. 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (visited on 07/26/2019).

[273] *The OpenStack Marketplace*. 2019. URL: https://www.openstack.org/marketplace/public-clouds (visited on 07/26/2019).

[274] *The SKA project*. 2019. URL: https://www.skatelescope.org/the-ska-project (visited on 07/26/2019).

[275] *The world's most valuable resource is no longer oil, but data*. 2017. URL: https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data (visited on 07/26/2019).

[276] Radu Tudoran, Alexandru Costan, Rui Wang, Luc Bougé, and Gabriel Antoniu. “Bridging Data in the Clouds: An Environment-Aware System for Geographically Distributed Data Transfers”. In: *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2014, pp. 92–101.

[277] *Under the Hood: Scheduling MapReduce jobs more efficiently with Corona*. 2012. URL: https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920 (visited on 07/26/2019).

[278] *Unikernels: Rethinking Cloud Infrastructure*. 2019. URL: http://unikernel.org (visited on 07/26/2019).

[279] *Vagrant*. 2018. URL: https://www.vagrantup.com (visited on 07/26/2019).

[280] Blesson Varghese, Nan Wang, Dimitrios S Nikolopoulos, and Rajkumar Buyya. "Feasibility of fog computing". In: *arXiv preprint arXiv:1701.05451* (2017).

[281] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC)*. 2013, p. 5.

[282] *VirtualBox VDI Image Format*. URL: http://forums.virtualbox.org/viewtopic.php?t=8046 (visited on 07/26/2019).

[283] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. "CLARINET: WAN-Aware Optimization for Analytics Queries". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 435–450.

[284] *VMware Virtual Disk Format 1.1*. URL: https://www.vmware.com/support/developer/vddk/vmdk_50_technote.pdf (visited on 07/26/2019).

[285] Maria A Voinea, Alexandru Uta, and Alexandru Iosup. "POSUM: A Portfolio Scheduler for MapReduce Workloads". In: *Proceedings of the IEEE International Conference on Big Data (Big Data)*. 2018, pp. 351–357.

[286] *vOpenData Dashboard*. 2019. URL: http://dash.vopendata.org (visited on 07/26/2019).

[287] *vSphere Hypervisor*. 2019. URL: https://www.vmware.com/products/vsphere-hypervisor.html (visited on 07/26/2019).

[288] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. "Hadoop Acceleration Through Network Levitated Merge". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2011, 57:1–57:10.

[289] Romain Wartel, Tony Cass, Belmiro Moreira, Ewan Roche, Manuel Guijarro, Sebastien Goasguen, and Ulrich Schwickerath. "Image Distribution Mechanisms in Large Scale Cloud Providers". In: *Proceedings of the IEEE 2nd International Conference on Cloud Computing Technology and Science (CloudCom)*. 2010, pp. 112–117.

[290] Hakim Weatherspoon and John D Kubiatowicz. "Erasure coding vs. replication: A quantitative comparison". In: *Proceedings of the International Workshop on Peer-to-Peer Systems*. 2002, pp. 328–337.

[291] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: A Scalable, High-performance Distributed File System". In: *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)*. 2006, pp. 307–320.

[292] Dr. Jeff Welser. *Cognitive Computing: Augmenting Human Capability*. 2011. URL: https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf (visited on 07/26/2019).

[293] *What are Availability Zones in Azure?* 2019. URL: https://docs.microsoft.com/en-us/azure/availability-zones/az-overview (visited on 07/26/2019).

[294] *What is Edge Computing: The Network Edge Explained.* 2018. URL: https://www.cloudwards.net/what-is-edge-computing/#cloud (visited on 07/26/2019).

[295] Edd Wilder-James. *What is big data? An introduction to the big data landscape.* 2012. URL: https://www.oreilly.com/ideas/what-is-big-data (visited on 07/26/2019).

[296] *Windows Azure Regions.* 2018. URL: https://azure.microsoft.com/en-us/regions (visited on 07/26/2019).

[297] Song Wu, Yihong Wang, Wei Luo, Sheng Di, Haibao Chen, Xiaolin Xu, Ran Zheng, and Hai Jin. "ACStor: Optimizing Access Performance of Virtual Disk Images in Clouds". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 28.9 (2017), pp. 2414–2427.

[298] Yu Wu, Zhizhong Zhang, Chuan Wu, Chuanxiong Guo, Zongpeng Li, and Francis CM Lau. "Orchestrating Bulk Data Transfers across Geo-Distributed Datacenters". In: *IEEE Transactions on Cloud Computing* 5.1 (2015), pp. 112–125.

[299] *Xen Project hypervisor.* 2019. URL: https://xenproject.org/users/virtualization/ (visited on 07/26/2019).

[300] Xiaolin Xu, Hai Jin, Song Wu, and Yihong Wang. "Rethink the Storage of Virtual Machine Images in Clouds". In: *Future Generation Computer Systems (FGCS)* 50 (Oct. 2015).

[301] Gala Yadgar and Moshe Gabel. "Avoiding the Streetlight Effect: I/O Workload Analysis with SSDs in Mind". In: *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*. 2016, pp. 36–40.

[302] Xiaoyu Yang, Bassem Nasser, Mike Surridge, and Stuart Middleton. "A business-oriented cloud federation model for real-time applications". In: *Future Generation Computer Systems (FGCS)* 28.8 (2012), pp. 1158–1167.

[303] Xin Yao, Cho-Li Wang, and Mingzhe Zhang. "EC-Shuffle: Dynamic Erasure Coding Optimization for Efficient and Reliable Shuffle in Spark". In: *Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2019, pp. 41–51.

[304] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. "Fog Computing: Platform and Applications". In: *Proceedings of the 3rd IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. 2015, pp. 73–78.

[305] Orcun Yildiz, Shadi Ibrahim, and Gabriel Antoniu. "Enabling Fast Failure Recovery in Shared Hadoop Clusters: Towards Failure-Aware Scheduling". In: *Future Generation Computer Systems (FGCS)* 74 (2016), pp. 208–219.

[306] Orcun Yildiz, Shadi Ibrahim, Tran Anh Phuong, and Gabriel Antoniu. "Chronos: Failure-aware scheduling in shared Hadoop clusters". In: *Proceedings of the IEEE International Conference on Big Data (Big Data)*. 2015, pp. 313–318.

[307] Matt MT Yiu, Helen HW Chan, and Patrick PC Lee. “Erasure Coding for Small Objects in In-memory KV Storage”. In: *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR)*. 2017, p. 14.

[308] Yinghao Yu, Renfei Huang, Wei Wang, Jun Zhang, and Khaled Ben Letaief. “SP-cache: Load-balanced, Redundancy-free Cluster Caching with Selective Partition”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2018, pp. 1–13.

[309] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. “Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling”. In: *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*. 2010, pp. 265–278.

[310] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. Vol. 10. 10-10. 2010, p. 95.

[311] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodík, Matthai Philipose, Victor Bahl, and Michael Freedman. “Live Video Analytics at Scale with Approximation and Delay-Tolerance”. In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2017, pp. 377–392.

[312] Heng Zhang, Mingkai Dong, and Haibo Chen. “Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication”. In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*. 2016, pp. 167–180.

[313] Xiaoxue Zhang and Feng Xu. “Survey of Research on Big Data Storage”. In: *Proceedings of the 12th International Symposium on Distributed Computing and Applications to Business, Engineering Science*. 2013, pp. 76–80.

[314] Zhaoning Zhang, Ziyang Li, Kui Wu, Dongsheng Li, Huiba Li, Yuxing Peng, and Xicheng Lu. “VMThunder: Fast Provisioning of Large-Scale Virtual Machine Clusters”. In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 25.12 (Dec. 2014), pp. 3328–3338.

[315] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. *Does erasure coding have a role to play in my data center?* Tech. rep. Microsoft research, 2010.

[316] Zhe Zhang, Andrew Wang, Kai Zheng, Uma Maheswara G., and Vinayakumar B. *Introduction to HDFS Erasure Coding in Apache Hadoop*. 2015. URL: `https://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop` (visited on 07/26/2019).

[317] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. “Wharf: Sharing Docker Images in a Distributed File System”. In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2018, pp. 174–185.

[318] Amelie Chi Zhou, Shadi Ibrahim, and Bingsheng He. “On Achieving Efficient Data Transfer for Graph Processing in Geo-Distributed Datacenters”. In: *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 1397–1407.

[319] Amelie Chi Zhou, Tien-Dat Phan, Shadi Ibrahim, and Bingsheng He. “Energy-Efficient Speculative Execution Using Advanced Reservation for Heterogeneous Clusters”. In: *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*. 2018, 8:1–8:10.

[320] Wu Zhou, Peng Ning, Xiaolan Zhang, Glenn Ammons, Ruowen Wang, and Vasanth Bala. “Always Up-to-date: Scalable Offline Patching of VM Images in a Compute Cloud”. In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*. 2010, pp. 377–386.

[321] Benjamin Zhu, Kai Li, and Hugo Patterson. “Avoiding the Disk Bottleneck in the Data Domain Deduplication File System”. In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*. Vol. 8. 2008, pp. 1–14.