Thèse n° 7207

An Architecture for Load Balance in Computer Cluster Applications

Présentée le 4 septembre 2020

à la Faculté informatique et communications
Laboratoire de systèmes d'exploitation
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Laurent BINDSCHAEDLER

École
polytechnique
fédérale
de Lausanne

2020

The only way to do great work is to love what you do.
— Steve Jobs

To my family, for their love and support.

Acknowledgements

During my Ph.D. journey, I have met many amazing people to whom I am forever grateful.

First, I want to express my deepest gratitude to my Ph.D. advisor, Willy Zwaenepoel, who guided me tirelessly throughout the past five years. I am grateful to Willy for encouraging me to strive to become an independent researcher and for the resulting opportunities that this has afforded me. I was exceptionally fortunate to work with him and benefit from his experience and perspective.

Second, I am indebted to my committee members: Edouard Bugnion, Ashvin Goel, and Dushyanth Narayanan, for their work and time in reviewing this manuscript and for their insightful feedback. I would also like to thank Prof. Alain Wegmann for presiding over the oral exam and for his valuable advice.

Third, I am incredibly grateful to Ashvin Goel, whom I have worked with closely during the past three years. His knowledge and expertise have contributed significantly to improve the quality of my work. He has guided and supported me consistently throughout my Ph.D. studies, and I consider him both a mentor and a close friend. Thank you, Ashvin!

The work in this dissertation is the result of collaboration with many other people. Chapter 3 was joint work with Amitabha Roy and Jasmina Malicevic. Chapter 4 was developed in collaboration with Jasmina Malicevic and Nicolas Schiper. More broadly, numerous people from the Operating Systems Laboratory (LABOS) and other laboratories (NAL, DCSL, DSLAB, DATA, DCL, LDS, VLSC) at EPFL contributed to the refinement of the ideas presented in this thesis.

Several students and researchers in LABOS helped me shape my ideas through many research discussions. Calin Iorgulescu was a great office mate who spent many hours brainstorming with me in front of a whiteboard. Florin Dinu, Kristina Spirovska, Baptiste Lepers, Pamela Delgado, Mihai Dobrescu, Diego Didona, Oana Balmau, Maria Chavez, Andrzej Nowak, Ivo Mihailovic, Peter Peresini, and Maciej Kuzniar were always around to provide valuable feedback. Also, a heartfelt thank you to the students and summer interns that I had the privilege of supervising during my Ph.D.: Junyao Zhao, Diego Antognini, Vlad Haprian, and Mario Bucev.

Various members of the administrative staff at EPFL and the IC doctoral school have helped me navigate through the regulations and administrivia. In particular, I would like to express my most profound appreciation to Madeleine Robert, the laboratory's administrative assistant, who was always ready to help me out. Also, Carlos Perez, Yohan Moulin, Stéphane Ecuyer, and Laurent Desimone made sure IT problems were minimal and promptly resolved.

I owe a special thanks to my entrepreneurship partner, Jean-Philippe Martin-Flatin, who introduced me to many areas orthogonal to research such as marketing and business development and helped me develop skills that became very useful during my Ph.D. I will always "blame" Jean-Philippe for introducing me to Willy. I was also fortunate to meet many brilliant researchers during my undergraduate studies at EPFL: Jean-Pierre Hubaux (my M.Sc. advisor), Amin Shokrollahi, and Jeffrey Huang (who advised me for semester projects). Murtuza Jadliwala, Igor Bilogrevic, Julien Freudiger, Reza Shokri, Imad Aad, Masoud Alipour, and Hendrik Knoche were also great research collaborators.

I was very fortunate to have many great friends and colleagues during my Ph.D. that supported me, inspired me, and kept me entertained. Thank you, Thomas, Maria, David, Régis, Ivan, Alexandre, Julien, Michael, Giuseppe, Hong-An, Marios, Mia, Fabiano, Julien, Nikolche, Diego, Yann, Marc, Nicolas, and Sabrina. I would also like to thank Suzie, Charade, Jiji, and Mimi for listening to me all these years. Finally, a big thank you to the many members of the improvisational theatre groups at EPFL: Morgane, Robin (×3), Laurane, John, Wendy, Tania, Martin, Yann, Guillaume, Maieul, Chloé, Basile, Thibault (×2), Quentin, Arseniy, Naira, Anne-Sophie, and many more.

I am very thankful to my parents, Elisabeth and Christian, and my siblings, Vincent and Delphine, for their support and encouragement. I would also like to thank my grandfather, Jean Gaeng, for his assistance with the German translation of this dissertation, as well as my other grandparents for their unwavering support.

Finally, I wish to thank my wonderful girlfriend, Mofasshara, for her support and patience.

Lausanne, April 19, 2020 L. B.

Abstract

Amid a data revolution that is transforming industries around the globe, computing systems have undergone a paradigm shift where many applications are scaled out to run on multiple computers in a computing cluster. As the storage and processing capabilities of a single machine are unable to keep pace with the amount of data, companies turn to distributed solutions to organize, persist, and analyze this *Big Data*. These new software solutions divide datasets into partitions that are processed in parallel on separate machines.

A common problem in current cluster computing frameworks is load imbalance and limited parallelism due to skewed data distributions, processing times, and machine speeds. Load imbalance occurs when a few machines have more work and take longer to finish while the others remain idle, resulting in reduced overall performance and low resource utilization. The underlying cause for these issues is that data locality, where machines process the data stored locally, leads to tight coupling between a partition and the machine on which it is placed.

This dissertation proposes a novel *scatter* architecture for computer cluster applications that effectively addresses the load imbalance problem and improves resource utilization. Scatter systems abandon data locality in favor of load balance. Existing systems first target locality, bringing computation to the data, and only then attempt to minimize load imbalance. Instead, we disregard any locality concerns and focus solely on achieving load balance by scattering all data across machines and bringing data to the computation as needed.

The scatter architecture disaggregates compute and storage resources and pools all resources together across machines. We ensure storage load balance by spreading the data for each partition in small blocks across all storage devices and allow machines to retrieve data using an efficient, decentralized scheme. Scatter systems achieve compute load balance at runtime by dynamically adjusting the parallelism within a partition, either through work-sharing or by offloading background tasks to other machines.

We design and implement three cluster applications inspired by the scatter architecture: a graph processing system that scales out using secondary storage, a general-purpose analytics framework, and a filesystem substrate that mitigates load imbalance for existing distributed databases. We demonstrate that each application can gracefully handle significant load imbalance with minimal performance degradation and that these applications can perform up to an order of magnitude faster than existing systems.

Keywords: *Big Data, cluster computing, load imbalance, locality, disaggregation, resource pooling, graph processing, analytics, distributed database, scatter.*

Résumé

En pleine révolution des données qui transforme les industries à travers le globe, les systèmes informatiques subissent un changement de paradigme où plusieurs applications passent à l'échelle sur plusieurs ordinateurs dans des grappes de serveurs. Les capacités de stockage et de traitement d'une seule machine étant incapables de suivre la quantité de données, les entreprises se tournent vers des solutions distribuées pour organiser, conserver et analyser ces *mégadonnées*. Ces nouvelles solutions logicielles divisent un jeu de données en plusieurs partitions qui sont ensuite traitées en parallèle sur des machines distinctes.

Un problème fréquent dans les infrastructures à grappes actuelles est le déséquilibre de la charge et les limitations au parallélisme provoqués par une distribution de données, un temps de traitement ou des perfomances asymétriques. Il y a déséquilibre de la charge lorsqu'un petit nombre de machines a plus de travail et nécessite ainsi plus de temps pour le terminer pendant que les autres machines restent inactives, ce qui implique de mauvaises performances globales et une faible utilisation des ressources. La cause sous-jacente de ces problèmes est que la localité des données, impliquant que les machines traitent les données stockées localement, conduit à un couplage strict entre une partition et la machine sur laquelle elle est placée.

Cette thèse propose une nouvelle architecture, dite *à dispersion* (*scatter*), pour les applications de grappes qui résout efficacement le problème de déséquilibre de la charge et améliore l'utilisation des ressources. Les systèmes à dispersion abandonnent la localité des données au profit de l'équilibre de la charge. Alors que les systèmes existants ciblent d'abord la localité, apportant le calcul aux données et ne tentent par la suite que de minimiser le déséquilibre de la charge, nous ignorons toute considération d'ordre de la localité et nous nous concentrons uniquement sur l'amélioration de l'équilibre de la charge en dispersant les données sur toutes les machines et en les apportant ensuite au calcul en fonction des besoins.

L'architecture à dispersion désagrège les ressources de calcul et de stockage et regroupe toutes ces ressources à travers les machines. Nous garantissons l'équilibre du stockage en répartissant les données de chaque partition en petits blocs sur tous les périphériques de stockage et permettons aux machines de récupérer des données efficacement à l'aide d'une technique décentralisée. Les systèmes à dispersion atteignent l'équilibre de la charge de calcul lors de l'exécution en ajustant dynamiquement le parallélisme au sein d'une partition, soit par un partage du travail, soit en déchargeant certaines tâches d'arrière-plan sur d'autres machines.

Nous concevons et implémentons trois applications de grappes inspirées de l'architecture à dispersion : un système de traitement de graphes utilisant le stockage secondaire, un cadriciel pour l'analytique générale, ainsi qu'une couche de système de fichiers qui minimise le déséquilibre de la charge pour les bases de données distribuées existantes. Nous démontrons que chaque application peut gracieusement gérer un déséquilibre de la charge important avec une dégradation minimale des performances et que ces dernières peuvent être jusqu'à un ordre de grandeur plus rapides que les systèmes existants.

Mots-clés : *Mégadonnées, grappe de serveurs, déséquilibre de la charge, localité, désagrégation, mise en commun des resources, traitement de graphe, analytique, base de données distribuée, dispersion.*

Zusammenfassung

Innerhalb der Datenrevolution, welche die Industrien der ganzen Welt umgestalten, haben Computersysteme einen Paradigma-schub erfahren, durch welche etliche Anwendungen auf eine Anzahl von Computern ausgebreitet werden, um ein Rechnerverbund zu bilden. Da die Speicher- und Verarbeitungskapazität eines einzelnen Gerätes nicht imstande ist, mit dem Anfall von Daten zu halten wenden sich Unternehmen zu Lösungen um die *Massendaten* organisieren und analysieren zu können. Neue Software-Lösungen verteilen Dateien auf Rechnerverbunde, welche gleichzeitig parallel auf mehreren Computern verarbeiten.

Ein gemeinsames Problem des derzeitigen Rechnerverbundrahmens ist ein Ungleichgewicht und beschränkter Parallelismus, wofür verzerrte Daten, Verteilung, Verarbeitungszeit und die Gerätegeschwindigkeit verantwortlich sind. Lastungleichgewichte entstehen, wenn einige Geräte mehr Arbeit haben und mehr Zeit brauchen, während die anderen pausieren. Dies führt zu mangelhafter Leistung und schlechter Nutzung der Ressourcen. Die zugrunde liegende Ursache für diese Probleme ist, dass die Datenlokalität, in der Maschinen die lokal gespeicherten Daten verarbeiten, zu einer engen Kopplung zwischen einer Datenpartition und der Maschine führt, auf der sie platziert ist.

Diese Dissertation schlägt eine neue *Ausstreuungs*-Architektur für Anwendungen durch Rechnerverbunde vor, die das Problem des Ungleichgewichts der Datenladung angehen und die Verwendung von Ressourcen verbessern. Die Verteilung auf Datenpartitionen verlässt Datenlokalität zugunsten von Lastungleichgewicht. Während die vorhandenen Systeme zuerst auf die Lokalität abzielen, indem sie die Verarbeitung der Daten ausführen und erst dann das Lastungleichgewicht zu vermindern, wir ignorieren die Datenlokalität und fokussieren lediglich auf Lastungleichgewicht durch die Verteilung der Daten auf etliche Geräte und laden die Daten nach Bedarf.

Die Ausstreungs-Architektur zieht die Verarbeitungs- und Speicherressourcen auseinander und verteilt alle Ressourcen über sämtliche Computer. Wir ermöglichen dem Lastausgleich des Speichers, indem wir die Daten in kleinen Blöcken über alle Datenpartitionen verteilen und es den Maschinen ermöglichen, Daten mit hilfe eines effizienten, dezentralen Schemas abzurufen. Ausstreuungs-Systeme bringen Lastausgleich an Computer durch Einlösung des Parallelismus innerhalb einer Datenpartition, entweder durch Arbeitsteilung oder delegieren Hintergrundaufgaben an andere Computer.

Wir entwerfen und implementieren drei Rechnerverbund-Anwendungen aufgrund der Ausstreuungs-Architektur: ein Betriebssystem für Graphen unter Benutzung von Sekundärspeicher, einen Rahmen für Allzweck-Analysen, und ein Ablagesystem, welches das Ungleichgewicht vorhandener Datenbanken mildert. Wir zeigen auf, dass jede Anwendung imstande ist, bedeutendes Ungleichgewicht ohne Betriebsstörung zu bewältigen, sowie dass diese Anwendungen bis zu einer Grössenordnung schneller als vorhandene Systeme arbeiten können.

Schlüsselwörter: *Massendaten, Rechnerverbund, Lastungleichgewicht, Datenlokalität, Auseinanderlegen, Ressourcenzusammenlegung, Graphverarbeitung, Analysen, verteilte Datenbank, Ausstreuung.*

Contents

1 Introduction

The past decade has witnessed unprecedented growth in the amount of data produced worldwide. Data has begun to fundamentally transform many industries, from high-energy particle physics to social networks, including media, payments, finance, travel, gene sequencing, and the internet of things [115]. By 2025, 463 exabytes of data will be created every day globally [141]. As more human activities become data-driven, many organizations are forced to scale out storage and processing of this *Big Data* to distributed computing infrastructure.

Today, thousands of software utilities designed to store and process large-scale datasets on a cluster of many computers are available. General-purpose cluster computing frameworks such as Google MapReduce [73], and Apache Hadoop [93] have paved the way for a myriad of improved solutions, including Apache Spark [160], Naiad [119], Apache Storm [10], and Apache Flink [18]. In turn, specialized systems have become mainstream, allowing for easier development of machine learning [37, 116, 135], graph processing [1, 89, 111], and data mining [118, 161] applications. New data storage paradigms have also emerged, enabling horizontally-scalable, distributed databases such as Google's BigTable [63], Amazon's DynamoDB [75], and MongoDB [19]. Most of these systems aim to maximize throughput while accessing and processing a large amount of data and thus do not have tight latency requirements. They generally operate in a similar fashion, splitting large datasets into partitions across machines, and then querying or processing each partition in parallel [19, 73, 111].

A frequent performance issue in cluster computing systems is load imbalance, where different machines take different amounts of time to finish their assigned tasks, wasting resources and limiting parallelism as the other machines remain idle [43]. Consequently, improving load balance is a key concern for developers and cluster operators as more balanced systems generally benefit from higher performance, faster job completion times, and better overall resource utilization[1]. Load imbalance takes various forms and has many possibles causes, including skewed data partitioning [106, 140], variance in generated intermediate state [77],

[1] In high-availability systems, load balancing also plays an important role to increase reliability by distributing the load across redundant components and ensuring proper failover in the event a component fails. This particular application area is not a primary concern in this thesis.

data-dependent processing times or filtering [88], irregular memory accesses [144], hardware heterogeneity or failures [40, 162], and interference from background tasks [51].

Achieving load balance in existing systems is hard because of data locality. Many current cluster computing systems statically partition data and assign work upfront before starting execution. They generally do not change assignments or increase the degree of parallelism within a partition once the input is split and partitions are assigned to machines. Several solutions have been proposed to improve load balance statically before execution starts, for example, using estimates of the necessary resources per partition [98, 152], sampling the input, or over-decomposition [129]. Unfortunately, these approaches introduce overhead, can be imprecise, and require significant programmer involvement. Other proposals attempt to correct load imbalance at runtime using speculative execution [74] or dynamic rebalancing of data across partitions [36, 101, 102]. These techniques also introduce overheads and unnecessarily increase the load on already overloaded machines [53, 54].

This dissertation proposes *scatter computing*, a novel architecture for distributed systems that successfully addresses load imbalance and improves resource utilization. This architecture builds on the observation that data locality, where each partition is stored and processed entirely on a machine, is often of little help in high-throughput cluster environments. Scatter systems disaggregate compute from storage and pool compute and storage resources on all machines together to spread the load of all partitions evenly without locality considerations. In the scatter architecture, all machines are collectively responsible for storing and processing the data in every partition. As a result, machines with a "heavy" load can leverage additional resources on other machines, and the system achieves better load balance.

Figure 1.1 illustrates the high-level architecture of a scatter system. Scatter systems scale storage by pooling all storage devices together and spreading data uniformly in fine-grained blocks (typically 1 MB) across all machines. They scale compute by dynamically adjusting the degree of parallelism within a partition, leveraging spare CPU resources on other machines to process a part of the partition. In the scatter architecture, any data block for any partition can, in principle, be processed by any machine.

The scatter architecture separates partitioning from work assignment. A partition does not belong to a machine; rather it is potentially shared among multiple machines *at runtime*. The system provides a second, storage-level sharding mechanism that redistributes the data in each partition in a balanced manner. Data blocks in a partition are accessed from different machines using a fast, decentralized scheme. This scheme makes it efficient to exploit parallelism at the data block-level rather than at the partition-level. The scatter architecture leverages this two-layer sharding mechanism and decentralized block access to achieve high performance compared to single-layer partitioning based on over-decomposition, which creates a large number of fine-grained partitions [129]. In scatter systems, partitioning remains very simple as partitions need not be balanced and can remain relatively large, avoiding significant scheduling, serialization, and processing overhead [150].

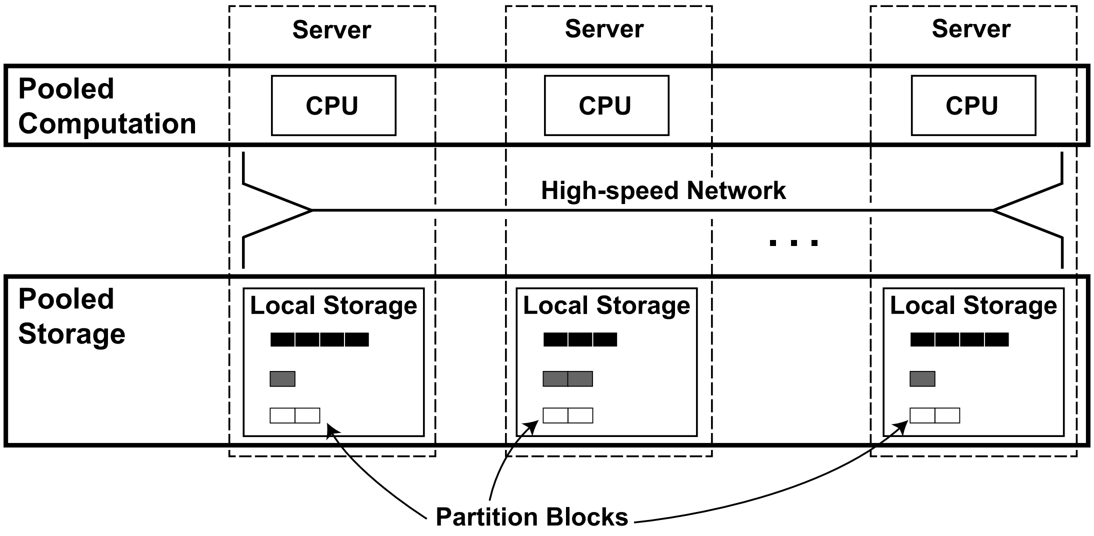


Figure 1.1 – The scatter architecture. Computation and storage are disaggregated, either logically (as shown in the figure) or physically by using separate, dedicated machines. The processing and storage resources of all machines are then pooled together. Each partition (represented in black/gray/white) is striped in small blocks and uniformly distributed across all storage devices.

The scatter architecture presents several benefits compared to current systems. First, it improves overall resource utilization in the presence of load imbalance by better balancing CPU usage and I/O load across machines. Second, it makes systems more adaptive to changing load conditions, as they can dynamically shift resources to help process "heavy" partitions. Third, it removes the need for complex partitioning as it is no longer necessary to optimize data placement, load balance, and locality. Finally, scatter systems achieve performance that is often an order of magnitude better than existing systems in the presence of load imbalance.

We implement the scatter architecture in three different distributed computing applications: graph processing using secondary storage, large-scale general-purpose analytics, and distributed LSM-based databases. We show how this new architecture improves scalability, load balance, and utilization, and present the specific design decisions in each case. For each application, we evaluate and compare the performance with best-in-class systems, and demonstrate that our implementations provide significant performance improvements.

Thesis statement *By disaggregating compute and storage and spreading the load evenly across pooled resources, forgoing data locality, we achieve load balance in cluster computing applications, specifically graph processing, general-purpose analytics, and distributed databases.*

In the remainder of this chapter, we survey the load imbalance problem in more detail and motivate the need for new solutions, present the scatter architecture and three use cases, and summarize the main results of this dissertation.

1.1 Background & Motivation

In distributed systems, load imbalance is a problem wherein the utilization of resources (compute, storage, network) across machines is unevenly distributed. Since all machines work together in parallel, the presence of load imbalance is undesirable because machines with higher resource usage will likely become a bottleneck. Load imbalance, therefore, decreases performance for the entire system and leads to reduced resource utilization.

Figure 1.2 illustrates the load imbalance problem for iterative graph processing using the widely-used GraphX [90] system that runs on top of Apache Spark [160]. We execute a Breadth-First Search (BFS) algorithm on the Twitter graph [2] using 32 machines and measure the total runtime for each machine. Runtimes exhibit high variance: while the mean runtime is 322s, the shortest is 217s, and the longest is 948s. The result is significant resource under-utilization as the application only terminates when the last machine is done, and therefore 31 machines remain largely idle for ~66% of the time. In this experiment, load imbalance is caused primarily by two factors: data imbalance due to imperfect partitioning of the graph and compute imbalance due to the characteristics of the BFS algorithm, which triggers computation on different vertices in each iteration.

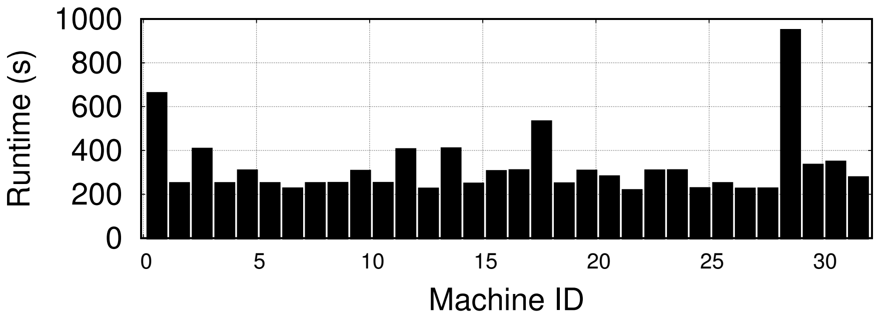


Figure 1.2 – Illustrating the load imbalance problem. Individual runtimes for each of 32 machines executing a parallel BFS computation on the Twitter graph using GraphX.

We next survey why load imbalance appears in distributed systems, what existing systems do to address it, and the drawbacks of these solutions.

Skew induces resource imbalance Skew can take many forms [113]. Many real-world datasets exhibit skew in the distribution of their features [106, 140]. For example, more people have names starting with B than Z, stars and politicians have millions of followers on social networks, half of the world's wealth is now in the hands of 1% of the population, etc. Even when the distribution is uniform, data-dependent processing times, filtering, and irregular memory accesses are hard to predict and can introduce skew in the processing of data records [88, 144]. Finally, processing the same data records may take different times on different machines due to, e.g., heterogeneous hardware, hardware failure, accessing data from disk rather than main memory, or interference from background tasks [40, 51, 162].

Balanced partitioning is challenging To achieve scalability, distributed systems divide the data into partitions (sometimes also called shards, fragments, or vnodes) and place these partitions on different machines. Partitioning aims to balance the load to maximize parallelism and to achieve locality to minimize data movement. Real-world data often exhibit complex shape or skewed data distribution, making it difficult to partition in a balanced way [106, 140]. For example, graphs are notoriously hard to partition due to skewed vertex degree distribution. Balanced graph partitioning is not only NP-hard but cannot even be approximated within a finite factor [45]. Even partitioning tuples and documents can prove challenging, and sampling may be necessary to determine the data distribution. As a result, partitioning is usually an expensive processing step. In multistage or iterative applications, the amount of intermediate state generated may differ by several orders of magnitude across machines [77], requiring repartitioning. Uneven data distribution can also lead to storage capacity problems if the data does not fit on a single machine or cause network link congestion as machines exchange data [47, 71]. In summary, since partitioning is done upfront, it is challenging to guarantee load balance during execution, especially for long-running distributed applications.

Heuristics are imprecise and expensive Once partitions are computed, a scheduler or resource manager [58, 60, 152, 153] assigns them to worker processes running on the machines that are part of the distributed system. Workers then process the data, copying it locally, if necessary, or service requests for their partitions. Unfortunately, optimally scheduling tasks to machines in a cluster subject to various constraints, such as load balance, locality, priority, and fast job completion time, is NP-hard [151]. Cluster schedulers must, therefore, rely on heuristics to estimate task resource usage and balance load across machines, which is often imprecise and expensive. Recent research suggests tuning the partitioning function to create a large number of fine-grained partitions to facilitate balanced task placement [129]. Although over-decomposition is a well-known technique to help mitigate the impact of stragglers, it introduces significant overheads to process all these tiny tasks [150]. Finally, a common scheduling technique that only addresses slow machines relies on speculatively executing copies of slow tasks on different machines [74].

"One partition = one worker" inevitably leads to hotspots In general, schedulers attempt to minimize network traffic by executing workers on the same machines as the partitions they are assigned to maximize data locality. A key limitation in many existing distributed systems is the fact that partitions are indivisible units of work to be processed by a single worker. Imbalanced partitions cause *hotspots*, where one or a few machines do most of the processing work or service a large fraction of requests. Some distributed systems attempt to rebalance partitions by migrating data to machines with less load. In distributed databases, this is typically achieved by changing the sharding key [36, 101]. In cluster computing frameworks, recent work proposes to identify slow tasks and split them across multiple machines [102]. Finally, in systems employing replication such as distributed databases, it is also possible to improve load balance by addressing a portion of requests for a popular shard to its replicas. However, this comes at the expense of synchronization overheads to keep the replicas synchronized.

All these solutions can successfully mitigate imbalance in some cases but cause significant data movement to/from a single machine, which can overwhelm resources if that machine is already overloaded, and often come too late to be effective [53].

In summary, existing solutions to overcome load imbalance in distributed systems fail to properly address its root cause: tight coupling between a data partition and the machine responsible for processing it. This coupling stems from data locality, where distributed systems always strive to have computation execute close to the data, ideally on the same machine. Partitioning, sampling, and heuristics proactively seek to avoid load imbalance but operate in a model constrained by data locality. Task splitting and partition rebalancing attempt to correct load imbalance at runtime by moving data and/or compute on less loaded machines. In fine, these reactive approaches fight against locality but do nothing to change the model.

Although data locality may be beneficial in some systems, e.g., real-time or transaction processing, it is often harmful to distributed systems that deal with Big Data as it increases the risk of creating load imbalance. Indeed, for many of these systems, data accesses are large, and throughput, not latency, is the key performance metric. Therefore, locality should no longer be a primary concern for the design of these systems.

A fundamental assumption allowing us to depart from data locality is that the network is not a critical bottleneck in the system. In that case, data can be accessed from a remote device at the same rate as from a local device. This assumption holds for clusters of modest size, in which machines, even with state-of-the-art SSDs, are connected by a high-speed commodity network. Recent work on datacenter networks suggests that this assumption also holds on a larger scale [46, 91, 124, 126].

1.2 Contributions

This dissertation introduces the scatter architecture, a novel architecture that enables distributed systems to achieve load balance across machines. The insight behind our approach is that addressing load imbalance in high-throughput systems requires abandoning data locality.

Therefore, scatter systems do not attempt to achieve locality and disaggregate compute and storage resources. The scatter architecture goes beyond disaggregation by pooling the storage and compute resources of all machines together and moving to a collaborative paradigm where storage and compute loads, respectively, are each spread evenly across pooled storage and pooled computation. We reverse the data locality paradigm in existing systems that executes computation close to the data. In essence, in scatter systems, the computation can run anywhere and only needs to pull the necessary data from remote, pooled storage. We take a proactive approach towards storage load balance by placing data evenly everywhere, and a reactive approach towards compute load balance by running computation where there are spare processing resources. This design enables efficient, balanced data storage and processing in the presence of arbitrary load imbalance.

Machines in scatter systems share a common distributed storage layer that pools together all their storage devices. This storage layer distributes all data uniformly in fine-grained blocks (typically 1 MB) across all storage devices, disregarding locality concerns. We leverage our storage layer to design an efficient, decentralized data access scheme based on oversubscription. This storage design removes single-machine storage capacity bottlenecks and ensures I/O load balance and high storage utilization even in the presence of imbalance across machines. It also addresses storage load peaks on single machines caused by temporary I/O bursts as each machine can efficiently access the available aggregate storage bandwidth.

All machines in a scatter architecture share responsibility for processing the data in every partition through dynamic work sharing. Scatter systems balance the processing by dynamically adjusting the parallelism within a single partition, allowing machines with spare compute resources to assist an overloaded machine by processing a part of its partition. Scatter systems solve synchronization issues due to concurrent data accesses to the same partition in an application-specific manner. In this thesis, we introduce a generic work-sharing solution for unordered data based on a bag abstraction that supports concurrently accessing the data in a partition in a coordination-free manner. We also demonstrate how to leverage specific characteristics of distributed databases by offloading background tasks to another machine.

We design three cluster computing applications based on the scatter architecture. These applications include both new systems designed from scratch and existing systems augmented with scatter capabilities. We now provide an overview of each application.

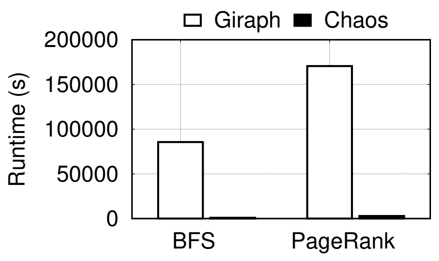
Graph processing Executing graph algorithms is one of the most popular and studied cluster computing applications in the past decade for which many specialized systems have been built [1, 64, 89, 90, 108, 111]. Graphs are a natural way to encode information and relationships, and the challenges involved in their processing have garnered much interest in the systems community due to their unpredictable access patterns, growing scale, and the power-law vertex degree distribution of many real-world graphs. We build Chaos, a scale-out graph processing system using secondary storage, and demonstrate that the system scales to trillions of edges using only a few machines.

General-purpose batch analytics We propose Hurricane, a cluster computing system to implement large-scale analytics algorithms. Cluster computing systems often perform poorly on applications that exhibit significant load imbalance [17, 102, 120]. Hurricane offers similar capabilities and expressiveness as Hadoop [93] and Spark [160], while ensuring skew resilience and high resource utilization. Hurricane includes primitives for filtering, transforming, sorting, counting, sketches [84], and supports multistage batch applications such as word counting and click log [54], as well as iterative-style applications such as sparse matrix-vector multiplication and graph processing.

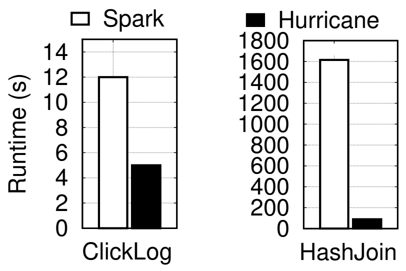
Distributed LSM-based databases Distributed databases [19, 20, 21, 22, 23, 44, 63, 69, 75, 154] organize large amounts of data in a structured or semi-structured manner, and provide the ability to update and retrieve information. Distributed databases often face throughput and latency issues due to load imbalance across machines and background task interference. Unlike the previous two applications where a new system was built from scratch, this example shows how existing distributed systems can use the scatter architecture. We build Hailstorm, a filesystem designed based on the scatter architecture, and ran mostly unmodified databases on this filesystem layer that improves throughput, latency, and utilization and interface it with MongoDB [19] over MongoRocks [24] and TiDB [25], two existing distributed databases built using Log-Structured Merge-tree (LSM) storage engines.

1.3 Main Results

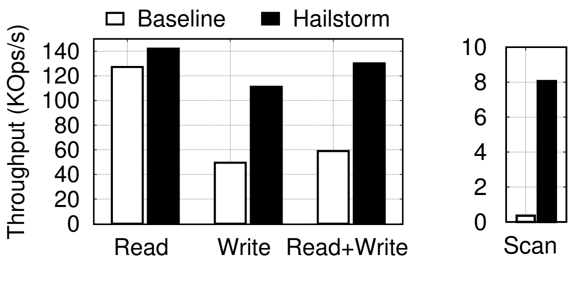
We have implemented the scatter architecture in three open-source systems, which we compare against state-of-the-art systems. Our evaluation focuses mainly on performance, especially for applications and datasets, showing significant load imbalance. We show that scatter systems generally exhibit performance close to a perfectly balanced workload despite data partitioning skew, compute imbalance, and interference from background operations. We also demonstrate that scatter systems incur a negligible overhead in the absence of load imbalance.



(a) Chaos (scatter) vs Giraph. Runtime for BFS and PageRank on RMAT-27 graph using 32 machines.



(b) Hurricane (scatter) vs Spark. Runtime for ClickLog on 320 MB input, and HashJoin on a 3.2 GB table using 32 machines.



(c) MongoDB with Hailstorm (scatter) vs MongoDB Baseline. Average throughput for four Zipfian workloads, each with an input of 100 GB using 8 machines.

Figure 1.3 – Summary of main results.

Figure 1.3 presents a summary of the main results in this thesis. For graph processing, we compare Chaos with Apache Giraph [1], an open-source graph processing framework inspired by Pregel [111], for the BFS and PageRank algorithms on an RMAT-27 graph [62] whose degree distribution follows a power-law. For general analytics, we compare Hurricane with Spark [160], a popular general-purpose cluster computing framework, using two typical applications on skewed inputs. Finally, for distributed databases, we augment the widely-used MongoDB [19] with Hailstorm and compare it with a pure MongoDB baseline using a Zipfian request distribution and four workloads: read-only, write-only, read+write, and scans. Overall, scatter systems provide significantly better performance than the state-of-the-art systems because they achieve better load balance, as shown in the next chapters.

We also demonstrate that scatter systems can process very large datasets using relatively few resources. Using Chaos, we execute the BFS algorithm on a synthetic RMAT [62] graph with 16 trillion edges and 1 trillion vertices in 10 hours on a cluster of 20 commodity servers. Chaos currently holds the third position for BFS in the Graph500 benchmark ranking for capacity [3], a ranking consisting mainly of supercomputers.

1.4 Publications

The contributions and results presented in Sections 1.2 and 1.3 were first introduced in the following publications:

- Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424. ACM, 2015[2].

- Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock You Like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the 13th EuroSys Conference*, EuroSys '18, pages 1–15. ACM, 2018.

- Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 301–316. ACM, 2020.

[2]The Chaos graph-processing system was designed and implemented in conjunction with Amitabha Roy. Very-large-scale extensions to Chaos (Ragnarok) were designed and implemented solely by the author.

1.5 Outline

This dissertation is organized as follows.

Chapter 2 introduces the scatter architecture, its key ideas, and components.

Chapter 3 proposes Chaos, a scatter system that enables processing graphs containing trillions of edges using secondary storage on small commodity clusters.

Chapter 4 presents Hurricane, a general-purpose cluster computing framework based on the scatter architecture that automatically handles load imbalance throughout execution.

Chapter 5 demonstrates the benefits of scatter computing for existing distributed LSM-based databases by introducing Hailstorm, a filesystem substrate that improves throughput and latency for skewed queries.

Chapter 6 surveys and compares the relevant literature.

Finally, we conclude and discuss possible research directions for future work in **Chapter 7**.

2 The Scatter Architecture

In this chapter, we describe the scatter architecture that helps distributed systems achieve load balance. The scatter architecture has two properties:

1. **Compute-Storage Disaggregation.** Separating processing from storage resources (logically or physically) allows each resource to scale independently. This property is necessary to address load imbalance properly because it enables the system to size each resource individually based on the requirements of a given task rather than adding entire machines.

2. **Resource Pooling and Load Scattering.** Combining the resources of all machines and dividing partitions into fine-grained units of data and work makes it possible to spread data evenly across pooled storage and to spread its processing evenly across pooled computation. In this collaborative paradigm, each machine is responsible for storing and processing all tasks. This property is sufficient to ensure good load balance because it allows the system to reassign spare resources available on a machine to process or store a "heavier" partition on another machine.

The scatter architecture forgoes data locality in an attempt to improve load balance. By dissociating compute from storage, we move away from the traditional model that executes computation close to the data. However, disaggregation alone does not guarantee load balance because it is still possible that most (if not all) data for a partition be placed on the same storage machine. We remove any locality considerations from the system by pooling resources and spreading the load evenly across them.

We first present the design of the scatter storage architecture. We then discuss the compute architecture. Finally, we discuss fault tolerance, and the scalability limitations of the scatter architecture.

2.1 Storage Architecture

2.1.1 Overview

Storage is not only the first place where imbalance generally appears due to partitioning skew, but it is also often slower than compute, and therefore likely to become a bottleneck in distributed systems. Scatter systems have three requirements for storage: I/O load balance, uniform data placement, and high storage utilization.

The scatter storage design relies on the following techniques to achieve these three goals:

1. **Storage pooling**. Scatter systems pool together storage devices to mitigate storage hotspots and relieve machines with taxed storage resources by sharing the I/O load across multiple machines.
2. **Uniform, fine-grained sharding**. Scatter systems split all data into fine-granularity blocks (typically 1 MB) and distribute these blocks uniformly across all storage devices, disregarding data locality. This approach removes single-machine storage capacity issues and spreads the I/O load uniformly.
3. **Decentralized access and oversubscription**. Scatter systems allow data access to any block of data from any machine. They leverage this decentralized access to avoid having storage devices un- or under-utilized by relying on a batch sampling technique where each storage device always has outstanding requests to service.

Figure 2.1 presents the scatter storage architecture. In the rest of this chapter, we describe each technique and explain how they help improve storage load balance and utilization. Next, we demonstrate how our storage architecture makes it possible to implement a fine-grained data sharing abstraction to support dynamic work sharing.

2.1.2 Storage Pooling

Figure 2.1 illustrates the architecture of a scatter storage service that pools storage from all machines using a client-server approach. Each compute node in the system runs a storage client, which exposes a filesystem interface that can be used to store and access data on the storage nodes. Each client divides files into blocks and places them on storage nodes. Each storage node runs a storage server that mediates access to the local storage devices and services client requests. Applications run as processes on the compute nodes and use their co-located client to perform storage operations.

This client-server architecture effectively decouples logical from physical storage, as clients store application data on remote storage servers. This decoupling enables logical disaggregation between compute and storage resources on a machine. Physical disaggregation is also possible by using separate, dedicated machines for computation and storage.

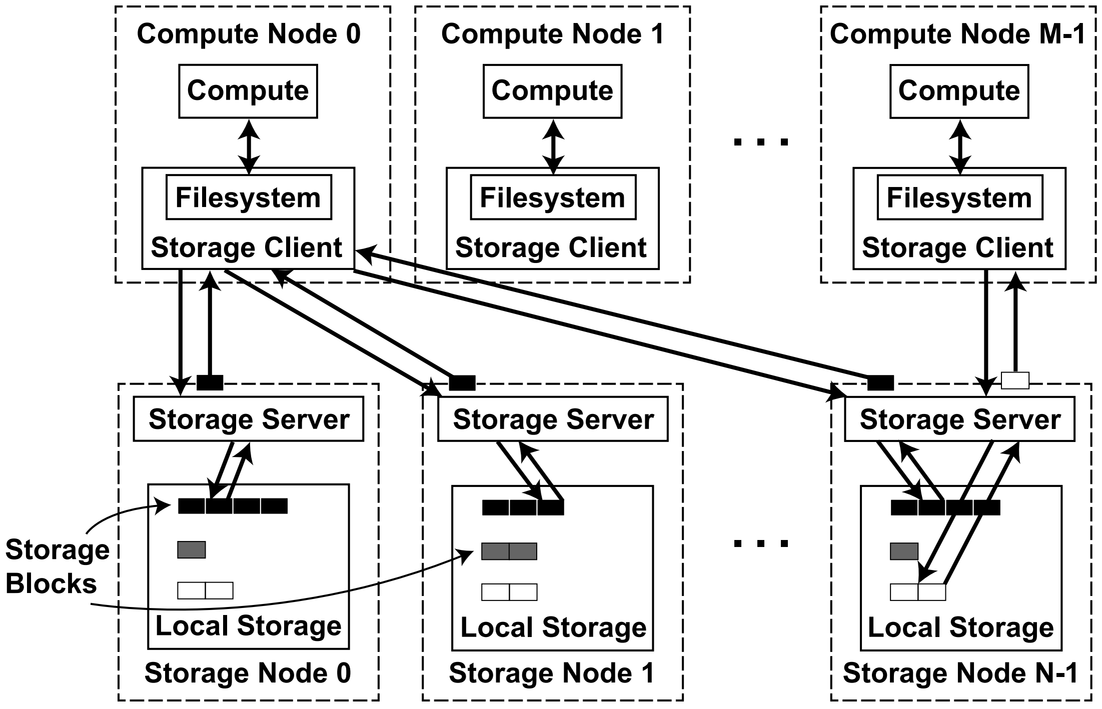


Figure 2.1 – The scatter storage architecture.

Our distributed storage service helps improve I/O load balance and avoids storage bottlenecks by pooling all storage devices in the cluster. As a result, clients can access multiple storage devices to spread the I/O load and benefit from fast data access. Another important consequence of this design is that storage disaggregation can help improve availability. Indeed, in the event of a compute node crash, we can easily spawn a new application process on a different machine to handle requests or process the data in a partition.

Clients identify files by a universally unique identifier (UUID) and store them in a flat namespace across servers. By default, clients can only see and modify their files, not the files created by other clients. Clients also keep all other metadata (file path, size, timestamps, permission, etc.) locally, since a file is typically accessed by a single client at a time. When file sharing across clients is necessary, sharing UUIDs and metadata is sufficient to provide access. Implementing synchronization for shared files is left to the application developer.

2.1.3 Uniform, Fine-Grained Sharding

As shown in Figure 2.1, a storage client splits all files into small blocks (typically 1 MB) and spreads blocks uniformly in a pseudorandom cyclic order across all storage servers, disregarding any locality concerns. This scheme alleviates the need for complex data placement and makes it easy to locate any block of data within a file. Given a file F, block size B, the number of servers N, and a pseudorandom mapping function $loc_F : \{0...N-1\} \rightarrow \{0...N-1\}$, the byte at offset I in F is on server $loc_F(\lfloor \frac{I}{B} \rfloor \mod N)$ in block number $\lfloor \frac{I}{BN} \rfloor$.

The scatter storage architecture is designed to support dynamically spreading the computation load within a single partition by adaptively adjusting the degree of parallelism. This allows multiple workers to collectively process the same partition by each processing a subset of the data. Adjusting the parallelism at runtime requires careful data placement, or else storage can become a bottleneck. For instance, placing all data on the machine that was assigned the partition will likely cause that machine to experience additional pressure due to data redistribution. Fine-grained storage-level sharding automatically distributes file blocks uniformly across all storage servers, ensuring that data placement is uniform, which, in turn, helps balance I/O load evenly. Uniform storage block placement also has an essential consequence for capacity: small and large partitions can cohabit without requiring additional provisioning or expensive rebalancing.

Since blocks are the unit of data storage and access, choosing a suitable block size is essential. In general, the size of a block is chosen to be large enough so that access to storage appears sequential, but small enough so that they can serve as units of distribution to achieve random uniform distribution across servers. Blocks are also the smallest amount of work that can be dynamically redistributed. Therefore, to achieve load balance, they need to be relatively small.

When servers use magnetic disks for storage, we find that block sizes of 1 to 16 MB incur minimal impact from random accesses to the disk. When using SSDs, we typically opt for a block size of 1 MB. Our sensitivity analysis indicates that block sizes ranging from 100 KB to 4 MB provide similar performance for most workloads. 1 MB provides a good balance between performance and remote access latency, incurs minimal impact from random accesses to the disk. Finally, if data is accessed at a finer-granularity than 1 MB and response latency is important, we find that block sizes of 32 KB or 64 KB are an excellent choice, even though they incur additional overhead from the network and the larger number of blocks causes more transitions to kernel mode.

2.1.4 Decentralized Access and Oversubscription

The scatter storage service has a fully decentralized data plane that allows fast data access from any storage client without relying on a centralized entity that maintains a directory of files, metadata, and block locations. Since data placement is deterministic, clients can independently request any block in a file from any server, and they can add, replace, or remove blocks at any server. Therefore, clients locate blocks on their own and communicate directly with the corresponding servers. This decentralized approach reduces latency as it avoids the need for a centralized data directory.

In the absence of data locality and with storage blocks spread uniformly across all machines, using a centralized directory that would place data blocks and coordinate access based on the load metrics of each machine is both unnecessary and undesirable. Not only would it increase block access latency, but it would also introduce a single point of failure, and it could quickly become a bottleneck due to a large number of fine-granularity storage blocks to manage.

However, this decentralized data plane can lead to load imbalance at storage servers in the absence of any coordination between clients accessing different servers. Since block requests generally involve random servers due to pseudorandom block placement, several clients may address their request to the same server, leaving other servers idle.

Scatter storage uses oversubscription to ensure high storage utilization. To keep all servers busy with high probability, we rely on a batch sampling technique inspired by [117, 132] where each client keeps multiple requests to different servers outstanding. Figure 2.1 illustrates this for the storage client in compute node 0. The number of such outstanding requests, called the batch factor K, is chosen to be the smallest number that with high probability keeps all servers busy all the time. The proper batch factor is derived as follows.

If a client has K requests outstanding, then only some fraction are being processed by the storage sub-system. The other requests are in transit. To ensure there are K outstanding requests at the servers, the clients use a larger *request window* ΦK. Φ is necessary to account for network delays and message processing (serialization, etc.). This amplification factor Φ can easily be computed by repeated application of Little's law [107]:

$$K = \lambda D_{storage}$$
$$\Phi K = \lambda(D_{storage} + D_{network})$$

where λ is the throughput of the server in terms of requests per unit time, $D_{storage}$ is the time for storage to service one request and $D_{network}$ is the application-level round trip time. Solving we have the required amplification Φ:

$$\Phi = 1 + \frac{D_{network}}{D_{storage}} \tag{2.1}$$

With this choice of Φ, we end up with K outstanding requests from each of M clients distributed at random across the N storage servers.

We can derive the utilization of a particular server as follows. The probability that a server is un-utilized is equal to the probability that no client picks it for any of its K requests:

$$\left(\frac{C_K^{N-1}}{C_K^N}\right)^M = (1 - \frac{K}{N})^M$$

The utilization of the server is therefore the probability that at least one client picks it, a function of the number of servers N and the batch-factor K:

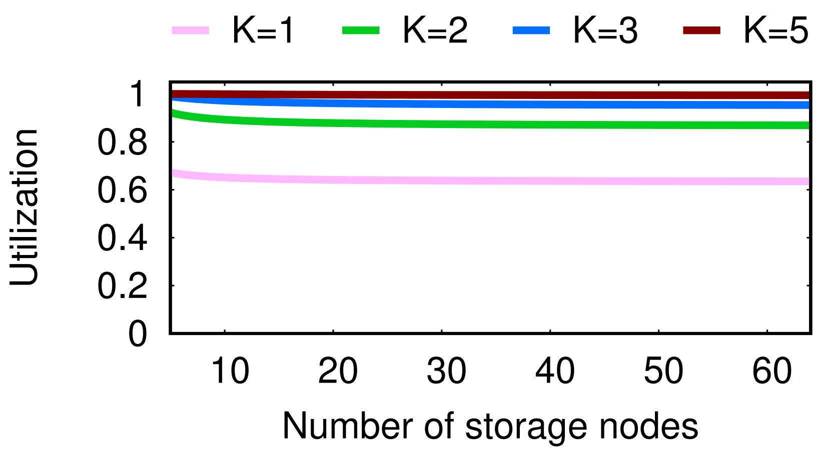


Figure 2.2 – Theoretical utilization for different number of storage nodes as a function of the batch factor K.

$$U(M,N,K) = 1-(1-\frac{K}{N})^M \tag{2.2}$$

Figure 2.2 shows the utilization as a function of the number of storage servers and for various values of K. For a fixed value of K, the utilization reduces with an increasing number of servers due to a greater probability of servers being left idle but *is asymptotic to a lower bound.* The lower bound is simply:

$$\lim_{M\to\infty, N\to\infty} U(M,N,K) = 1-\frac{1}{e^K} \tag{2.3}$$

Therefore, it suffices to pick a value for K large enough to approach 100% utilization regardless of the number of storage servers. For example, using $K=5$ means that the utilization cannot drop below 99.3%.

Clients ensure that they each have at most ΦK concurrent requests in progress. This also serves as a simple flow control scheme to avoid overwhelming storage nodes.

Prefetching The above description assumes that at any point in time, a client knows which next blocks to access and can issue more outstanding requests. However, that may not always be the case for each file and application, and as a result, storage utilization may be lower than expected. Scatter applications are, therefore, particularly well-suited for sequential access workloads where the client can automatically prefetch several blocks.

2.1.5 Fine-Grained Data Sharing with Bags

To dynamically adjust the degree of parallelism within a partition, separate processes running on different compute nodes must be able to access disjoint subsets of the data in that partition concurrently. In Subsection 2.1.2, we argued that data sharing between clients generally requires programmers to implement custom solutions. In this section, we describe how our storage architecture makes it easy to support a *bag* abstraction that enables fine-grained data sharing across multiple clients in a coordination-free manner.

A bag is an unordered collection of *chunks*. Chunks contain data records used by the application and are similar to blocks. However, unlike blocks, a single data record cannot span multiple chunks, i.e., each chunk contains an integral number of data records. Bags allow concurrent accesses, but only support a subset of operations:

- `insert(chunk)`: Inserts (append) a chunk into the bag.
- `remove()`: Removes (read) a chunk from the bag and returns it to the caller. The removed chunk is chosen at random. If the bag has no more blocks, this operation fails, indicating that the contents of the bag have been fully consumed.

The bag abstraction guarantees that each chunk in the bag is returned *exactly once*. Ensuring that chunks are not lost and that the same chunk is not processed multiple times is usually required to guarantee correct semantics for the application when sharing a bag across multiple compute nodes.

Bags are implemented as files in our distributed storage service but do not rely on the loc_F mapping to find specific blocks. This is because bags are unordered and, therefore, inserts and remove can be performed at any storage server. However, clients still use the pseudorandom mapping loc_F as a cyclic permutation of servers to determine the next server to pick for each operation to maintain I/O load balance and high utilization. The bag abstraction provides a prefetching-friendly workload and, therefore, works well with oversubscription (§2.1.4) because applications generally remove chunks from a bag until it becomes empty.

A chunk insert request simply appends the chunk to the file associated with the bag by sending a request to the next server in the cyclic permutation. The append operation is atomic, ensuring that concurrent inserts are performed correctly. Insert operations are performed in a FIFO order. Similarly, a remove operation is implemented by reading a block from the file sequentially on the next server in the permutation, which increments the file pointer and ensures that the same block is never returned again. An end-of-file indicates that all blocks have been removed from this server. To determine that a bag is empty, a client must get an end-of-file from each server.

2.2 Compute Architecture

2.2.1 Overview

Once storage load is balanced in a distributed system (§2.1), CPU is the next likely resource to suffer from imbalance and potentially become a bottleneck. Scatter systems adopt a collaborative paradigm where all compute nodes share responsibility for processing the data in every partition. These systems improve the CPU load balance by dynamically adjusting the parallelism within a single partition.

In this dissertation, we explore two ways to adjust parallelism dynamically:

1. **Dynamic work sharing** enables machines with spare processing resources to assist an overloaded machine by processing a part of its partition.
2. **Background task offloading** aims to alleviate some CPU pressure on overloaded machines by allowing them to outsource non-critical background tasks to less utilized machines.

Both approaches would not make sense in a system with data locality. Executing computation on machines that do not have the necessary data locally would first require moving that data to the computation. In the scatter architecture, it does not matter *where* the data is, and therefore it is possible to execute the computation anywhere. Workers running on any compute node can simply pull the necessary data at a fine granularity from pooled storage.

In the following section, we describe each technique in more detail.

2.2.2 Dynamic Work Sharing

Ideally, we would like to leverage parallelism to speed-up the processing of a heavy partition by having multiple compute nodes process a part of the partition in parallel. The scatter storage architecture supports fine-grained data sharing within a single partition, in turn allowing fine-grained dynamic work sharing across compute nodes by having storage clients on each compute node pull blocks from the same partition. Since the data for every partition is spread in blocks across storage nodes, we need not worry about overloading an already overloaded machine by asking it to relinquish part of its partition and migrate the corresponding data. Instead, the overloaded compute node can share the data for its partition by transmitting the UUIDs of its associated files (§2.1.2) to clients running on separate compute nodes, allowing other worker processes to access the data and share in the processing work directly. However, this potentially requires clients accessing data in the same partition concurrently to coordinate. Each application must, therefore, implement its solution to this problem.

The bag abstraction described in Subsection 2.1.5 is one such example of a solution that supports concurrent accesses to the same dataset by multiple clients in a coordination-free manner. Based on CPU load metrics, systems can easily adjust the number of worker processes processing a single bag to achieve compute load balance. Upon detecting a CPU bottleneck on a compute node, the system simply spawns a new worker process on a different compute node to assist processing the data in the bag. Over time, this simple scheme redistributes compute resources more evenly across compute nodes. Spawning a new worker is not always free, as there may be overhead associated with combining multiple partial results for the same partition in a single consistent output. We explore a simple solution to this problem based on programming-model support for partial output merging and heuristics that balance the cost of merging outputs with the benefits of increased parallelism.

We first use the bag abstraction in Chapter 3 to support work stealing in large-scale graph processing. In Chapter 4, we further explore this abstraction and use it as a first-class citizen in the Hurricane cluster computing framework.

2.2.3 Offloading Background Tasks

In addition to dynamic work sharing, or when assisting a compute node with part of its task is not supported or possible, CPU load can be balanced by offloading background tasks from an overloaded node.

Background tasks are not directly related to a system's primary operations but are in general necessary for its correct functioning. Examples of background tasks include, e.g., asynchronous data replication [148], logging, flushing in-memory buffers, garbage collection, or compaction in Log-Structured Merge-tree-based databases [133].

Some background tasks can be particularly expensive and consume significant resources to run. Whenever possible, scatter applications should strive to offload "heavy" background tasks to compute nodes with low resource utilization. In doing so, compute nodes experiencing high load from processing their main task or servicing requests can reclaim some much-needed resources.

In this dissertation, we explore background task offloading in Chapter 5, where we design a mechanism to offload compaction tasks in Log-Structured Merge-tree-based distributed databases.

2.3 Fault Tolerance

We briefly discuss fault tolerance concerns for the scatter architecture. In the event of a crash, we primarily rely upon the failure recovery mechanisms implemented in the application. We present the details of these mechanisms, which are application-specific, in the following chapters. Similarly, applications should implement their own replication (if needed) as it is an application-level concern that cannot easily be implemented in a generic way and with insufficient visibility.

A consequence of our architecture is that all persistent state is placed on storage nodes, while clients and processes generally only contain soft state. This allows applications to implement simpler recovery mechanisms in case of compute crashes. In some cases, this architecture also enables easier failover if a process or machine crashes by starting a new process elsewhere that accesses the same data.

When pooling storage, a single disk failure or machine crash may cause all partitions to become unavailable since the data for each shard is spread uniformly. We mitigate single-disk failures using standard techniques to ensure redundancy, e.g., RAID [136]. We also support optional primary-backup replication at the block level to further protect data durability and filesystem availability. File metadata is persisted locally and replicated.

2.4 Scalability Limitations

Since scatter systems spread data across all storage nodes, without considerations for locality, an underlying assumption in our design is that machine-to-machine network bandwidth exceeds the bandwidth of a storage device and that network switch bandwidth exceeds the aggregate bandwidth of all storage devices in the cluster. Under this assumption, the network is never the bottleneck.

This requirement is met by many workloads and deployments today. Recent work has shown that for many analytics workloads, the network is not the bottleneck, and its effect is mostly irrelevant to overall performance [55]. This is because much less data is sent over the network than is accessed from disk [130]. Our storage system is designed to optimize the latter bottleneck. While our approach increases network communication, network interface speeds today are easily able to keep up with storage bandwidth. A 10 GigE interface can easily support modern disks as well as fast SSDs, and 40 GigE networks are becoming more common. Thus, we expect that network endpoints will not be a bottleneck in our deployments. Similarly, high-bisection bandwidth is available at rack scale today, and many clusters are deployed at these scales. For example, in 2011, Cloudera reported a median cluster size of 30 and a mean of 200 nodes [9]. Similarly, many Hadoop clusters have 100-200 nodes [11]. For larger installations, data-center scale full bisection bandwidth networks are being actively researched and deployed [46, 91, 124, 126].

Although not explored in-depth in this work, scatter systems could define the granularity at which resources are pooled based on the availability of high bisection bandwidth. For example, if sufficient bandwidth is available within a rack but not across racks, the system could only pool resources within each rack, while using a standard architecture across racks.

3 Graph Analytics with Chaos[1]

In this chapter, we consider a first distributed system based on the scatter architecture: Chaos, a system that scales out graph processing from secondary storage.

3.1 Introduction

Processing large graphs is an application area that has attracted significant interest in the research community [64, 89, 90, 94, 99, 103, 111, 122, 123, 125, 137, 144, 155, 163].

Triggered by the availability of graph-structured data in domains ranging from social networks to national security, researchers are exploring ways to mine useful information from such graphs. A serious impediment to this effort is the fact that many graph algorithms exhibit irregular access patterns [109]. As a consequence, most graph processing systems require that the graphs fit entirely in memory, necessitating either a supercomputer or a very large cluster [89, 90, 122, 123].

Systems such as GraphChi [103], GridGraph [163] and X-Stream [144] have demonstrated that it is possible to process graphs with edges in the order of billions on a single machine, by relying on secondary storage. This approach considerably reduces the entry barrier to processing large graphs. Such problems no longer require the resources of very large clusters or supercomputers. Unfortunately, the amount of storage that can be attached to a single machine is limited, while graphs of interest continue to grow [137]. Furthermore, the performance of a graph processing system based on secondary storage attached to a single machine is limited by its bandwidth to secondary storage [112].

We investigate how to scale out graph processing systems based on secondary storage to multiple machines, with the dual goals of increasing the size of graphs they can handle to

[1]This chapter is based on the following publication: Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424. ACM, 2015. The author of this dissertation designed and implemented this system in collaboration with Amitabha Roy, and led the experimental evaluation jointly with Jasmina Malicevic. After publication, the author developed the very-large-scale extensions described in Section 3.11, and maintains the software to this day.

the order of a trillion edges and improving load balance to enhance their performance, by accessing secondary storage on different machines in parallel.

The conventional approach for scaling graph processing to multiple machines is first to partition the graph statically, and then to place each partition on a separate machine, where the graph computation for that partition takes place. Partitioning aims to achieve load balance to maximize parallelism and locality to minimize network communication. Achieving high-quality partitions that meet these two goals can be time-consuming, especially for out-of-core graphs. Optimal partitioning is NP-hard [85], and even approximate algorithms may take considerable running time. Also, static partitioning cannot cope with later changes to the graph structure or variations in access patterns throughout the computation.

Chaos takes a fundamentally different approach to scale out graph processing on secondary storage. This approach is grounded on the principles of the scatter architecture and, therefore, parts with locality as a first principle. First, rather than performing an elaborate partitioning step to achieve load balance and locality, Chaos performs a very simple initial partitioning to achieve sequential storage access. It does this by using a variant of the streaming partitions introduced by X-Stream [144]. Second, rather than locating the data for each partition on a single machine, Chaos spreads all graph data (vertices, edges, and intermediate data, known as updates) uniformly randomly over all secondary storage devices to balance I/O load across machines. Data is stored in large enough chunks to maintain sequential storage access. This approach assumes that network bandwidth is sufficient and, therefore, network is not a bottleneck. Third, since different streaming partitions can have very different numbers of edges and updates, and, therefore, require very different amounts of work, Chaos increases the degree of parallelism by allowing more than one machine to work on the same streaming partition, using a form of work stealing [56] for balancing the compute load between machines.

We evaluate Chaos on a cluster of 32 machines with ample secondary storage and connected by a high-speed network. We can scale up the problem size by a factor of 32, going from 1 to 32 machines, with on average only a 1.61× increase in runtime. Similarly, for a given graph size, we achieve speedups of 10 to 22 on 32 machines. The aggregated storage also lets us handle a graph with 16 trillion edges. This result represents a new milestone for graph processing systems on small commodity clusters. In terms of capacity it rivals those from the high performance computing community [4] and very large organizations [5] that place the graph on supercomputers or in main memory on large clusters. Therefore, Chaos enables the processing of very large graphs on rather modest hardware.

We also examine the conditions under which good scaling occurs. We find that sufficient network bandwidth is critical, as it underlies the assumption that locality has little effect. When sufficient network bandwidth is available, performance improves more or less linearly with available storage bandwidth. The number of cores per processor has little or no effect, as long as enough cores are available to sustain high network bandwidth.

The main contributions of this work are:

- We build the first efficient scale-out graph processing system from secondary storage (§3.2).
- We use a very cheap partitioning scheme to achieve sequential access to secondary storage rather than expensive partitioning for locality and load balance (§3.3).
- We allow multiple machines to work on the same partition to achieve compute load balance through randomized work stealing (§3.5).
- We forgo locality in storage access, and we achieve I/O load balance by uniformly spreading and accessing data at random (§3.6).
- We demonstrate that Chaos achieves high capacity and performance, as well as good load balance on a cluster of 32 machines (§3.9).

3.2 Programming Model

Chaos adopts an edge-centric and somewhat simplified GAS (Gather-Apply-Scatter) model [64, 89, 144].

The state of the computation is stored in the value field of each vertex. The computation takes the form of a loop, each iteration consisting of a scatter[2], gather and apply phase. During the scatter phase, updates are sent over edges. During the gather phase, updates arriving at a vertex are collected in that vertex's accumulator. During the apply phase these accumulators are applied to produce a new vertex value. The precise nature of the computation in each of these phases is specified by three user-defined functions, `Gather`, `Apply`, and `Scatter`, which are called by the Chaos runtime as necessary.

Listing 3.1 provides pseudo-code for the overall computation. During the scatter phase, for each edge, the `Scatter` function is called, taking as argument the vertex value of the source vertex of the edge, and returning the value of an update sent to the destination vertex of the edge. During the gather phase, for each update, the `Gather` function is called, updating the accumulator value of the destination vertex of the update using the value supplied with the update. Finally, during the apply phase, for each vertex, the `Apply` function is called, applying the value of the accumulator to compute the new vertex value.

```
while not done
  // Scatter
  for all e in Edges
    u = new update
    u.dst = e.dst
    u.value = Scatter(e.src.value)
```

[2]The scatter phase, one of the three phases in the GAS model, should not be confused with the scatter architecture introduced in this dissertation.

```
  // Gather
  for all u in Updates
    u.dst.accum = Gather(u.dst.accum, u.value)
  // Apply
  for all v in Vertices
    Apply(v.value, v.accum)
```

Listing 3.1 – GAS Sequential Computation Model.

Listing 3.2 shows, for example, how the PageRank algorithm [134] is implemented in the GAS model. When executing on multiple machines, vertices may be replicated to achieve parallelism. For each replicated vertex there is a master. Edges or updates are never replicated. During the scatter phase, parallelism is achieved by processing edges (and producing updates) on different machines. The update phase is distributed by each replica of a vertex gathering a subset of the updates for that vertex in its local accumulator. The apply phase then consists of applying all these accumulators to the vertex value (see Listing 3.3).

```
// Scatter
function Scatter(value val)
  return val.rank / val.degree

// Gather
function Gather(accum a, value val)
  return a + val

// Apply
function Apply(value val, accum a)
  val.rank = 0.15 + 0.85 * a
```

Listing 3.2 – PageRank using Chaos.

```
// Apply
for all v in Vertices
  for all replicas v' of v
    Apply(v.value, v'.accum)
```

Listing 3.3 – `Apply` in Distributed Computation Model.

The edge-centric nature of the programming model is evidenced by the iteration over edges and updates in the scatter and gather phases, unlike the vertex-centric model [111], in which the scatter and gather loops iterate over vertices. This model is inherited from X-Stream, and has been demonstrated to provide superior performance for graph processing from secondary storage [144]. The GAS model was introduced by PowerGraph, and naturally expresses distributed graph processing, in which vertices may be replicated [89]. Finally, Chaos follows the simplifications of the GAS model introduced by PowerLyra [64], scattering updates only over outgoing edges and gathering updates only for incoming edges.

As in other uses of the GAS model, Chaos expects the final result of multiple applications of any of the user-supplied functions `Scatter`, `Gather` and `Apply` to be independent of the order in which they are applied in the scatter, gather and apply loops respectively. Chaos takes advantage of this order-independence to achieve an efficient solution. In practice, all our algorithms satisfy this requirement, and so we do not find it to be a limitation.

3.3 Streaming Partitions

Chaos uses a variation of X-Stream's [144] streaming partitions to achieve efficient sequential secondary storage access. A streaming partition of a graph consists of a set of vertices that fits in memory, all of their outgoing edges and all of their incoming updates.

Executing the scatter and gather phases one streaming partition at a time allows sequential access to the edges and updates while keeping all (random) accesses to the vertices in memory.

In X-Stream the size of the vertex set of a streaming partition is – allowing for various auxiliary data structures – equal to the size of main memory. This choice optimizes sequential access to edges and updates while keeping all accesses to the vertex set in memory. In a distributed setting, other considerations play a role in the proper choice for the size of the vertex set. Main memory size remains an upper bound to guarantee in-memory access to the vertex set, and large sizes facilitate sequential access to edges and updates. However, smaller sizes are desirable, as they lead to easier load balancing.

Therefore, we choose the number of partitions to be the smallest multiple of the number of machines such that the vertex set of each partition fits into memory. We simply partition the vertex set in ranges of consecutive vertex identifiers. Edges are partitioned such that an edge belongs to the partition of its source vertex.

This partitioning is the only pre-processing done in Chaos. It requires one pass over the edge set and a negligible amount of computation per edge. Furthermore, it can easily be parallelized by splitting the input edge list evenly across machines. This low-cost pre-processing stands in stark contrast to the elaborate partitioning algorithms that are typically used in distributed graph processing systems. These complex partitioning strategies aim for static load balance and locality [89]. Chaos dispenses with locality entirely and achieves load balance at runtime.

3.4 Design Overview

Chaos follows the scatter architecture described in Chapter 2 and consists of a computation subsystem and a storage subsystem. The two subsystems are logically separated and communicate using a client-server model. In the following, we assume that each machine runs both a compute node and a storage node, but it is also possible to run them on separate, dedicated machines.

The storage subsystem consists of a storage node on each machine. The storage node runs a storage server that supplies vertices, edges, and updates of different partitions to the computation subsystem. The vertices, edges, and updates of a partition are uniformly randomly spread over the pooled storage servers.

The computation subsystem consists of a compute node on each machine, running a computation engine. Each computation engine bundles a storage client that performs all I/O on the storage subsystem on its behalf. The computation engines collectively implement the GAS model. Unlike the conceptual model described in Section 3.2, the actual implementation of the model in Chaos has only two phases per iteration, a scatter and a gather phase. The apply phase is incorporated into the gather phase, for reasons of efficiency. There is a barrier after each scatter phase and after each gather phase. The `Apply` function is executed as needed during the gather phase and does not imply any global synchronization.

The Chaos design supports dynamic work sharing (§2.2.2) through work stealing, allowing multiple computation engines to work on a single partition at the same time, to achieve computational load balance. When this is the case, each engine must read a disjoint set of edges (during the scatter phase) or updates (during the gather phase). This responsibility rests with the storage servers. This division of labor allows multiple computation engines to work on the same partition without synchronization between them.

Chaos ensures that all storage devices are kept busy all the time, thereby achieving maximum utilization of the bottleneck resource, namely the bandwidth of the storage devices.

3.5 Computation Subsystem

The number of streaming partitions is a multiple k of the number of computation engines. Therefore, each computation engine is initially assigned k partitions. This engine is the master for all vertices of those partitions, or, for short, the master of those partitions.

We start by describing the computation in the absence of work stealing. This aspect of Chaos is similar to X-Stream, but is repeated here for completeness. Later, we show how Chaos implements work stealing between computation engines. The complete pseudo-code description of the computation engine (including stealing) is shown in Listing 3.4.

```
// Scatter for partition P
function exec_scatter(P)
  for each unprocessed e in Edges(P)
    u = new update
    u.dst = e.dst
    u.value = Scatter(e.src.value)
    add u to Updates(partition(u.dst))

// Gather for partition P
function exec_gather(P)
```

```
  for each unprocessed u in Updates(P)
    u.dst.accum = Gather(u.dst.accum, u.value)

/////// Chaos compute engine

// Pre-processing
for each input edge e
  add e to Edges(partition(e.src))

// Main loop
while not done

  // Scatter phase
  for each of my partitions P
    load Vertices(P)
    exec_scatter(P)

  // When done with my partitions, steal from others
  for every partition P_Stolen not belonging to me
    if need_help(Master(P_Stolen))
      load Vertices(P_Stolen)
      exec_scatter(P_Stolen)
  global_barrier()

  // Gather Phase
  for each of my partitions P
    load Vertices(P)
    exec_gather(P)

    // Apply Phase
    for all stealers s
      accumulators = get_accums(s)
      for all v in Vertices(P)
        Apply(v.value, accumulators(v))
    delete Updates(P)

  // When done with my partitions, steal from others
  for every partition P_Stolen not belonging to me
    if need_help(Master(P_Stolen))
      load Vertices(P_Stolen)
      exec_gather(P_Stolen)
      wait for get_accums(P_Stolen)
  global_barrier()
```

Listing 3.4 – Chaos Computation Engine.

3.5.1 Scatter Phase

Each computation engine works on its assigned partitions, one at a time, moving from one of its assigned partition to the next (lines 23–33) without any global synchronization between compute nodes. The vertex set of the partition is read into memory, and then the edge set is streamed into a large main memory buffer. As edges are processed, updates may be produced. These updates are binned according to the partition of their target vertex and buffered in memory. When a buffer is full, it is written to storage. Multiple buffers are used, both for reading edges and writing updates, to overlap computation and I/O.

3.5.2 Gather Phase

Each computation engine works on its assigned partitions, one at a time, moving from one of its assigned partition to the next (lines 35–45) without any global synchronization between compute nodes. The vertex set of the partition is read into memory, and then the update set is streamed into a large main memory buffer. As updates are processed, the accumulator of the destination vertex is updated. Multiple buffers are used for reading updates to overlap computation and I/O.

3.5.3 Work Stealing

The number of edges or updates to be processed may differ significantly between partitions, and therefore between computation engines, causing CPU load imbalance. Chaos uses work stealing to even out the load, as described next.

When computation engine i completes the work for its assigned partitions (lines 23–26 for scatter and lines 35–38 for gather), it goes through every partition p (for which it is not the master) and sends a proposal to help out with p to its master j (line 30 for scatter and line 49 for gather). Depending on how far along j is with that partition, it accepts or rejects the proposal, and sends a response to i accordingly. In the case of a negative answer, engine i continues to iterate through the other partitions, each time proposing to help. It does so until it receives a positive response or until it has determined that no help is needed for any of the partitions. In the latter case, its work for the current scatter or gather phase is finished, and it waits at a barrier (line 33 for scatter and line 53 for gather).

When engine i receives a positive response to help out with partition p, it reads the vertex set of that partition from storage into its memory and starts working on it. When two or more engines work on the same partition, it is essential that they work on a disjoint set of edges (during scatter) or updates (during gather). Chaos puts this responsibility with the storage system: it makes sure that in a particular iteration, an edge or an update is processed only once, independent of how many computation engines work on the partition to which that edge or update belongs. This is easy to do in the storage system and avoids the need for synchronization between the computation engines involved in stealing.

For stealing during scatter, a computation engine proceeds exactly as it does for its partitions. Using the user-supplied `Scatter` function, the stealer produces updates into in-memory buffers and streams them to storage when the buffers become full.

Stealing during gather is more involved. As before, a computation engine reads updates from storage and uses the user-supplied `Gather` function to update the accumulator of the destination vertex of the update. There are now, however, multiple instances of the accumulator for this vertex, and their values need to be combined before completing the gather phase. To this end, the master of the partition keeps track of which other computation engines have stolen work from it for this partition. When the master completes its part of the gather for this partition, it sends a request to all those computation engines and waits for an answer. When a stealer completes its part of the gather, it waits to receive a request for its accumulator from the master, and eventually sends it to the master (line 52). The master then uses the user-supplied `Apply` function to compute the new vertex values from these different accumulators, and writes the vertex set back to storage.

The order in which the master and the stealers complete their work is unpredictable. When a stealer completes its work before the master, it waits until the master requests its accumulator values before it does anything else (line 52). When the master completes its work before one or more of the stealers, it waits until those stealers return their accumulator (line 42). On the plus side, this approach guarantees that all accumulators are in memory at the time the master performs the apply. On the minus side, there may be some amount of time during which a computation engine remains idle. An alternative would have been for an engine that has completed its work on a partition to write its accumulators to storage, from where the master could later retrieve them. This strategy would allow an engine to start work on another partition immediately. The idle time in our approach is, however, very short, because all computation engines that work on the same partition read from the same set of updates, and therefore all finish within a very short time of one another. Therefore, we prefer this efficient and straightforward in-memory approach over more complicated ones, such as writing the accumulators to storage or interrupting the master to incorporate the accumulators from stealers.

3.5.4 To Steal or Not To Steal

Stealing is helpful if the cost, the time for the stealer to read in the vertex set, is smaller than the benefit, the reduction in processing time for the edges or updates still to be processed at the time the stealer joins in the work. Since Chaos is I/O-bound, this decrease in processing time can be estimated by the decrease in I/O time caused by the stealer.

This estimate is made by considering the following quantities: B is the bandwidth to storage seen by each computation engine, D is the amount of edge or update data remaining to be read for processing the partition, H is the number of computation engines currently working on the partition (including the master), and V is the size of the vertex state of the partition.

If the master declines the stealing proposal, the remaining time to process this partition is $\frac{D}{BH}$. If the master accepts the proposal, then $\frac{V}{B}$ time is required to read the vertex set of size V. Since we assume that bandwidth is limited by the storage servers and not by the network, an additional helper increases the bandwidth from BH to $B(H+1)$, and decreases the remaining processing time from $\frac{D}{BH}$ to $\frac{D}{B(H+1)}$. The master accepts the proposal if and only if:

$$\frac{V}{B} + \frac{D}{B(H+1)} < \frac{D}{BH} \tag{3.1}$$

$$\Longrightarrow V + \frac{D}{(H+1)} < \frac{D}{H} \tag{3.2}$$

The master knows the size of the vertex set V, and keeps track of the number of stealers H. It estimates the value of D by sampling the amount of edge or update data still to be processed on one storage server and multiplying it by the number of storage servers. Since the data is evenly spread across storage servers, this estimate is accurate and makes the decision process local to the master. This stealing criterion is incorporated in `need_help()` on lines 30 and 49 of the pseudo-code in Listing 3.4.

3.6 Storage Subsystem

For an out-of-core graph processing system such as Chaos, computation is only one half of the system. The other half consists of the storage subsystem that supplies the I/O bandwidth necessary to move graph data between storage and main memory.

3.6.1 Stored Data Structures and Their Access Patterns

For each partition, Chaos records three data structures on storage: the vertex set, the edge set, and the update set. The accumulators are temporary structures and are never written to storage.

The access patterns of the three data structures are quite different. Edge sets are created during pre-processing and are read during scatter[3]. Update sets are created and written to storage during scatter and read during gather. After the end of a gather phase, they are deleted. Vertex sets are initialized during pre-processing, and always read in their entirety, both during scatter and gather. Read and write operations to edges and updates may be performed by the master or by any stealers. In contrast, read operations to the vertex state may be performed by the master or any stealers, but only the master updates the vertex values during apply and writes them back to storage.

These three data structures are implemented using the scatter storage architecture as follows.

[3]In an extended version of the model, edges may also be rewritten during the computation.

3.6.2 Edge and Update Sets

Since the order in which edges and updates for the same partition are processed is irrelevant, Chaos stores each partition of edges and update sets in a bag (§2.1.5). Each bag consists of chunks containing edges or updates that are spread uniformly across all storage nodes. Computation engines rely on storage clients to retrieve chunks for their partitions and process them accordingly until the bag is empty. Bags guarantee only once delivery of each chunk during an iteration, ensuring correct semantics for the graph algorithm. Bags also support dynamic work sharing, ensuring that multiple computation engines process disjoint data during work stealing (§3.5.3).

3.6.3 Vertex Sets

Unlike edges and update sets, vertex sets are always accessed in their entirety and the order in which the blocks are stored and accessed matters. Therefore, each vertex set is stored as a file whose blocks are also spread uniformly across all storage servers.

3.6.4 Achieving High Storage Utilization

Chaos relies on oversubscription to ensure throughput and storage utilization remain high the execution. Each compute engine maintains a fixed number of outstanding requests for blocks and chunks to storage servers through its storage client (§2.1.4). The bag abstraction used for edge and update sets supports prefetching, and since vertex sets are always read sequentially and in full, clients can use prefetching when requesting blocks from storage servers.

3.6.5 Fault Tolerance

The fact that all graph computation state is stored in the vertex values, combined with the synchronous nature of the computation, allows Chaos to tolerate transient machine failures simply and efficiently. At every barrier at the end of a scatter or gather phase, the vertex values are checkpointed using a 2-phase protocol [82] that makes sure that the new values are completely stored before the old values are removed.

Chaos minimizes the chances of storage failures by employing RAID [136] or similar redundancy techniques. The system can also support recovery from storage failures by replicating the files containing vertex sets as described in Section 2.3.

3.7 Implementation

Chaos is written in C++ and amounts to approximately 15’000 lines of code. Figure 3.1 shows the high-level architecture and typical deployment of Chaos.

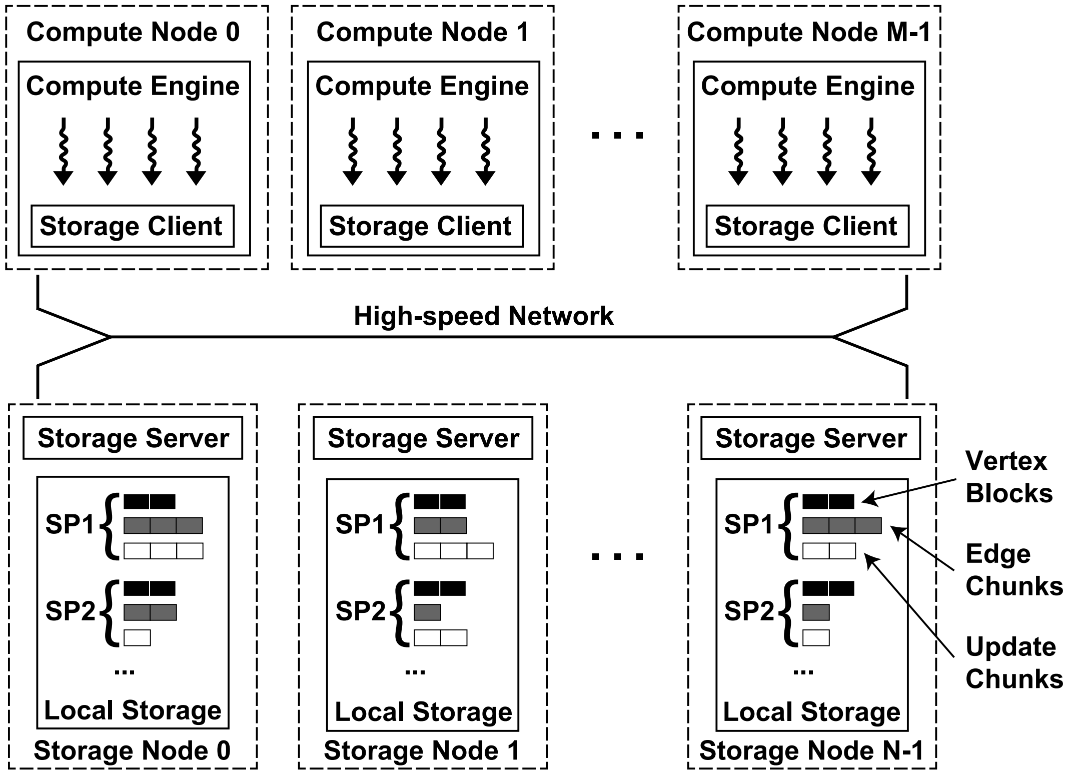


Figure 3.1 – Chaos Architecture. Compute engines run multiple threads on each compute node. Streaming partitions (SP) consisting of vertex, edge, and update sets are stored across all storage servers.

In its capacity as a scatter system, Chaos disaggregates computation and storage but runs the computation engine, the storage client, and the storage server on each machine in separate threads within the same process. We use ∅MQ [6] on top of TCP sockets for message-oriented communication between storage clients and servers, assuming a full bisection bandwidth network between the machines. We tune the number of ∅MQ threads for optimal performance.

We implement the scatter storage architecture described in Section 2.1. The storage servers provide a simple interface to the local ext4 [114] file system. Unlike X-Stream, which uses direct I/O, Chaos uses pagecache-mediated access to the storage devices. We select a block size 4 MB, leading to good sequentiality and performance on both magnetic disks and SSDs.

3.8 Experimental Environment and Benchmarks

We evaluate Chaos on a rack with 32 16-core machines, each equipped with 128 GB of main memory, a 480 GB SSD, and two 6 TB magnetic disks (arranged in RAID 0). Unless otherwise noted, the experiments use the SSDs as storage devices. The machines are connected through 40 GigE links to a top-of-rack switch. The SSDs and disks provide bandwidth in the range of 420 MB/s and 330 MB/s, respectively, well within the capacity of the 40 GigE interface on the machine. We limit the available main memory on each machine to 32 GB to ensure there is sufficient I/O in all experiments.

We use the same set of algorithms as used by X-Stream [144] to demonstrate that all the single machine algorithms used in the evaluation of X-Stream can be scaled to our cluster. Table 3.1 presents the complete set of algorithms, as well as the X-Stream runtime and the *single machine* Chaos runtime for an RMAT-27 graph. As can be seen, the single-machine runtimes are similar but not identical. In principle, Chaos running on a single machine is equivalent to X-Stream. The two systems have, however, different code bases and, in places, different implementation strategies. In particular, Chaos is a distributed system that relies on the scatter storage architecture and its client-server model for I/O. All storage servers in Chaos are pooled and access storage through the pagecache, whereas X-Stream uses direct I/O.

Algorithm	**Type**	**State Size**	**X-Stream**	**Chaos**
Breadth-First Search (BFS)	undirected	8B	497s	594s
Weakly Connected Comp. (WCC)	undirected	12B	729s	995s
Min. Cost Spanning Trees (MCST)	undirected	20B	1239s	2129s
Maximal Independent Sets (MIS)	undirected	1B	983s	944s
Single Source Shortest Paths (SSSP)	directed	20B	2688s	3243s
PageRank (PR)	directed	12B	884s	1358s
Strongly Connected Comp. (SCC)	directed	16B	1689s	1962s
Conductance (Cond)	directed	0B	123s	273s
Sparse Matrix Vector Mult. (SpMV)	undirected	20B	206s	508s
Belief Propagation (BP)	directed	32B	601s	610s

Table 3.1 – Algorithms, characteristics, and single-machine runtime on RMAT-27 for X-Stream and Chaos, SSD.

We use a combination of synthetic RMAT graphs [62] and the real-world Data Commons dataset [7]. RMAT graphs can be scaled in size easily: a scale-n RMAT graph has 2^n vertices and 2^{n+4} edges. In other words, the size of the vertex and edge sets doubles with each increment in the scale factor. We use the newer 2014 version of the Data Commons graph that encompasses 1.7 billion webpages and 64 billion hyperlinks between them.

Input to the computation consists of an unsorted edge list, with each edge represented by its source and target vertex and an optional weight. If necessary, we convert directed to undirected graphs by adding a reverse edge. Graphs with fewer than 2^{32} vertices are represented in a compact format, with 4 bytes for each vertex and the weight, if any. Graphs with more vertices are represented in non-compact format, using 8 bytes instead. A scale-32 graph with weights on the edges thus results in 768 GB of input data. The input of the unweighted Data Commons graph is 1 TB.

All results report the wall-clock time to go from the unsorted edge list, randomly distributed over all storage devices, to the final vertex state, recorded on storage. Therefore, all results include pre-processing time.

3.9 Scaling Results

3.9.1 Weak Scaling

In the weak scaling experiment we run RMAT-27 on one machine, and then double the size for each doubling of the number of machines, ending up with RMAT-32 on 32 machines.

Figure 3.2 shows the runtime results for these experiments, normalized to the runtime of a single machine. In this experiment, Chaos takes on average 1.61× the time taken by a single machine to solve a problem 32× the size on a single machine. The fastest algorithm (Cond) takes 0.97×, while the slowest (MCST) takes 2.29×.

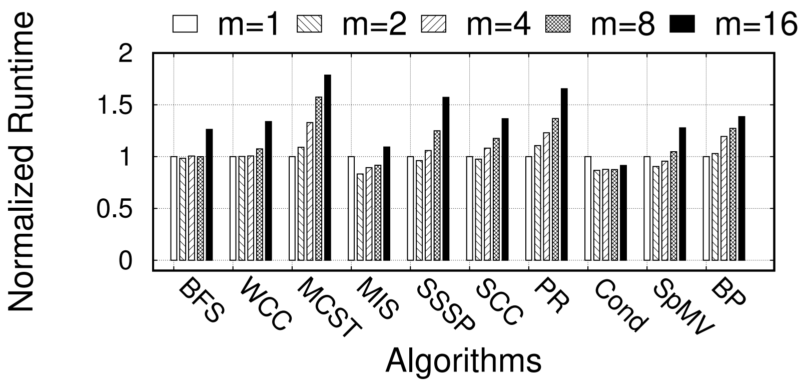


Figure 3.2 – Runtime normalized to 1-machine runtime. Weak scaling, RMAT-27 to RMAT-32, SSD.

The differences in scaling between algorithms result from a combination of characteristics of the algorithms, including the fact that the algorithm itself may not scale perfectly, the degree of load imbalance in the absence of stealing, and the size of the vertex sets. One interesting special case is Conductance, where the scaling factor is slightly smaller than 1. This somewhat surprising behavior is the result of the fact that with a larger number of machines, the updates fit in the buffer cache and do not require storage accesses.

3.9.2 Strong Scaling

In this experiment, we run all algorithms on 1 to 32 machines on the RMAT-27 graph. Figure 3.3 shows the runtime, again normalized to the runtime on one machine. For this RMAT graph, 32 machines provide, on average, a speedup of about 13× over a single machine. The fastest algorithm (Cond) runs 23× faster and the slowest (MCST) 8×. The results are somewhat inferior to the weak scaling results, because of the small size of the graph.

To illustrate this, we perform a strong scaling experiment on the much larger Data Commons graph. This graph does not fit on a single SSD, so we use HDDs. Furthermore, given the long running times, we only present results for two representative algorithms, BFS and PageRank. Figure 3.4 shows the runtimes on 1 to 32 machines, normalized to the single-machine runtime.

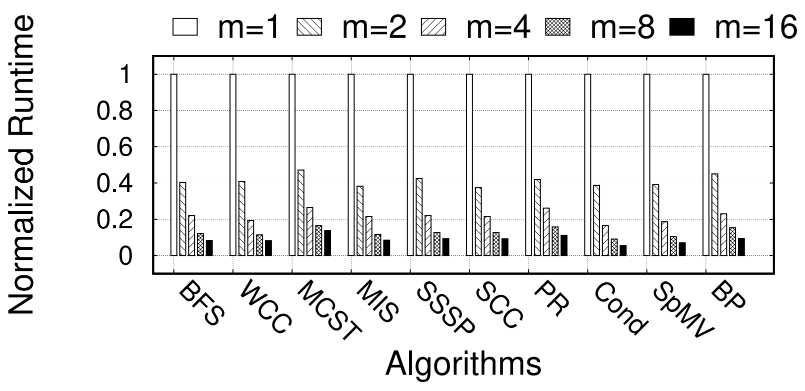


Figure 3.3 – Runtime normalized to 1-machine runtime. Strong scaling, RMAT-27, SSD .

Using 32 machines, Chaos provides a speedup of 20 for BFS and 18.5 for PageRank.

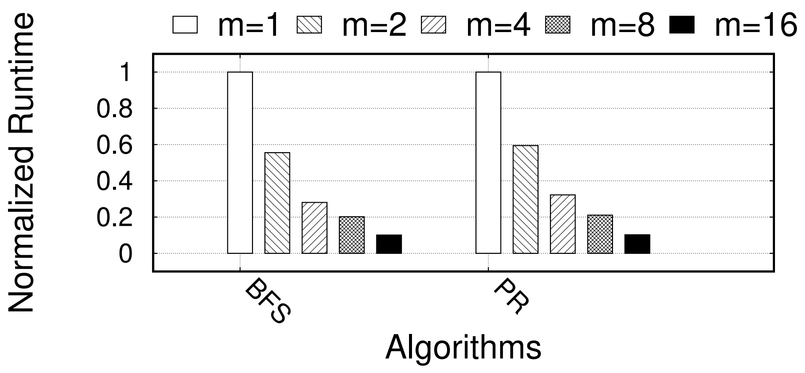


Figure 3.4 – Runtime normalized to 1-machine runtime. Strong scaling, Data Commons, RAID0-HDD.

3.9.3 Capacity Scaling

We use RMAT-36 with 250 billion vertices and 1 trillion edges to demonstrate that we can scale to large graphs. This graph requires 16 TB of input data, stored on HDDs. Chaos finds a breadth-first order of the vertices of the graph in a little over 9 hours. Similarly, Chaos runs 5 iterations of PR in 19 hours. These experiments require I/O in the range of 214 TB for BFS and 395 TB for PR, and the pooled storage servers provide an aggregate of 7 GB/s from the 64 magnetic disks running in orchestration.

3.9.4 Scaling Limitations

We evaluate the limitations to scaling Chaos with respect to the specific processor, storage bandwidth, and network links available.

Figure 3.5 presents the results of running BFS and PR as we vary the number of CPU cores available to Chaos. As can be seen, the system performs adequately, even with half the CPU cores available. It is nevertheless worth pointing out that Chaos requires a minimum number of cores to maintain good network throughput.

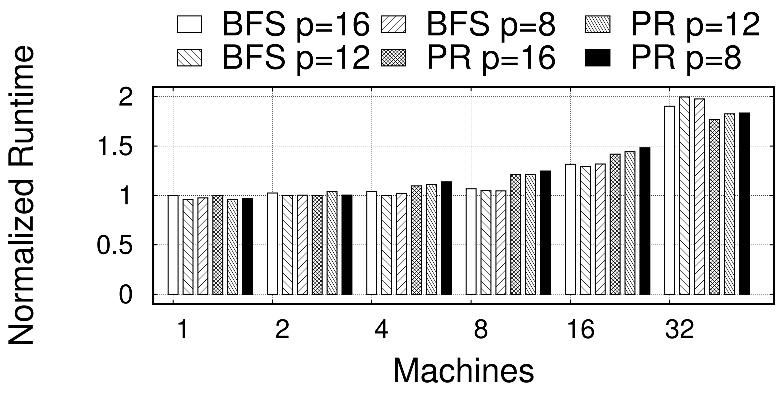


Figure 3.5 – Runtime for Chaos with different number of CPU cores, normalized to 1-machine runtime with cores=16.

Figure 3.6 compares the performance of BFS and PR when running from SSDs and HDDs. The HDD bandwidth is 2× less than the SSD bandwidth. Chaos scales as expected regardless of the bandwidth, but the application takes time inversely proportional to the available bandwidth.

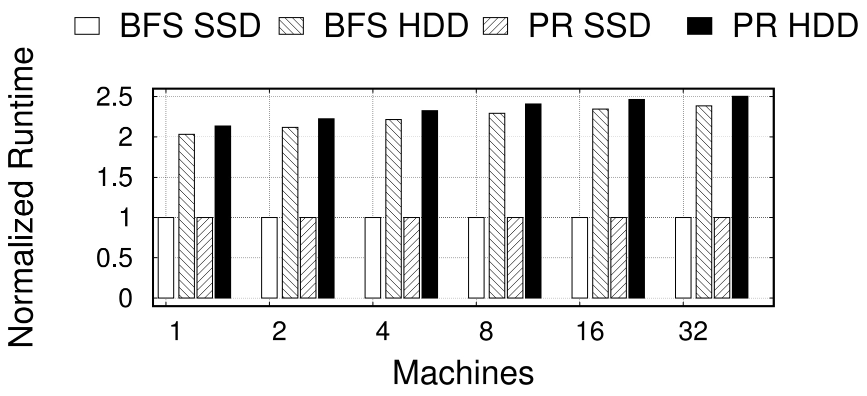


Figure 3.6 – Runtime for Chaos with SSD and HDD, normalized to 1-machine runtime with SSD.

Figure 3.7 looks into the performance impact of a slower network by using a 1 GigE interface to connect all machines instead of the faster 40 GigE. The throughput achieved by the 1 GigE interface is approximately 1/4th of the disk bandwidth, breaking the scatter architecture assumption that network is never the bottleneck (§2.4). We conclude from these results that Chaos does not scale as well in such a situation, highlighting the need for network links that are faster (or at least as fast) as the storage bandwidth per machine.

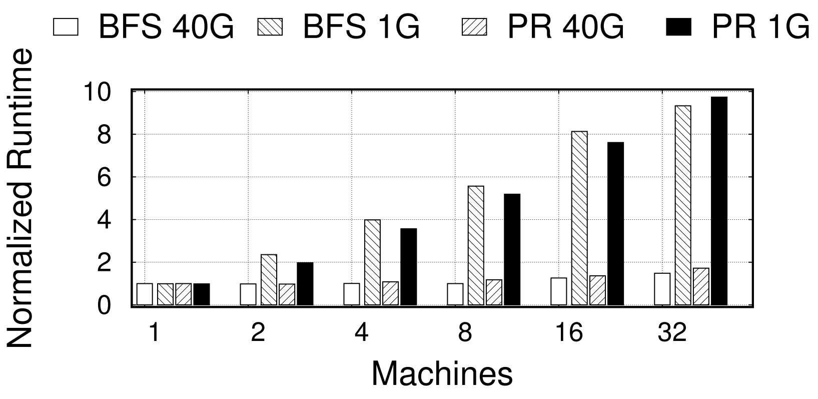


Figure 3.7 – Runtime for Chaos with 1 GigE and 40 GigE, normalized to 1-machine runtime.

3.9.5 Checkpointing

For large graph analytics problems, Chaos provides the ability to checkpoint state. Figure 3.8 shows the runtime overhead for checkpoints on a scale-36 graph for BFS and PR. As can be seen, the overhead is under 6% even though the executions write hundreds of terabytes of data to the Chaos store.

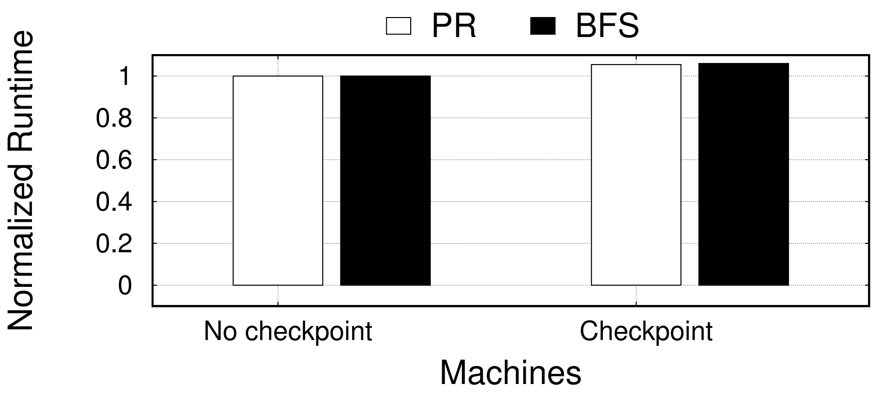


Figure 3.8 – Chaos vs. Chaos with checkpointing enabled (32 machines, RMAT-35, HDD, normalized to Chaos runtime).

3.10 Evaluation of Design Decisions

As a scatter system, Chaos does not attempt to achieve locality and pools compute and storage on all machines to improve load balance. Also, Chaos does not rely on complex partitioning. In this section, we evaluate the effect of these design decisions. All discussion in this section is based on the weak scaling experiments. The effect of the design decisions for other experiments is similar and not repeated here. For some experiments, we only show the results of BFS and PageRank as representative algorithms.

3.10.1 No Locality

Instead of seeking locality, Chaos pools all storage devices together, spreads all graph data uniformly randomly across all storage servers, and relies on oversubscription to achieve high storage throughput and utilization.

Figure 3.9 shows the aggregate bandwidth obtained as seen by all storage clients running as part of the computation engines during the weak scaling experiment. The figure also shows the maximum bandwidth of the storage devices, measured by fio [8].

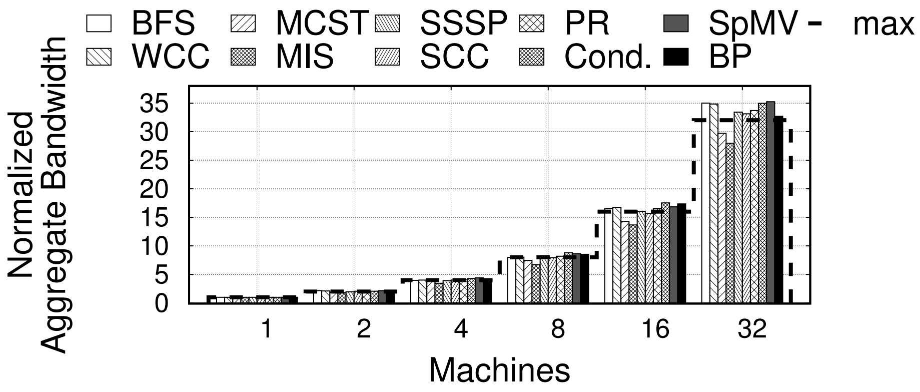


Figure 3.9 – Aggregate bandwidth normalized to 1-machine bandwidth and maximum theoretical aggregate bandwidth.

Two conclusions can be drawn from these results. First, the aggregate bandwidth achieved by Chaos scales linearly with the number of machines. Second, the bandwidth achieved by Chaos is within 3% of the available storage bandwidth, the bottleneck resource in the system.

We also evaluate a couple of more detailed design choices in terms of storage access, namely the pseudorandom selection of storage servers and the batch sampling technique designed to keep all storage servers busy.

Figure 3.10 compares the runtime for PageRank on 1 to 32 machines for Chaos to a design where a centralized entity selects the storage server for reading and writing a chunk. In short, all read and writes go through the centralized entity, which maintains a directory of where each chunk of each vertex, edge, or update set is located. As can be seen, the running time with Chaos increases more slowly as a function of the number of machines than with the centralized entity, which increasingly becomes a bottleneck.

Next, we evaluate the efficacy of batch sampling in our disk selection strategy. Figure 3.11 shows the effect of increasing the window size of outstanding requests on performance. We measured the latency to the SSD to be approximately equal to that on the 40 GigE network. This means $\Phi = 2$ (Equation 2.1). The graph shows a clear sweet spot at $\Phi K = 10$, which corresponds to $K = 5$. This means an utilization of 99.56% with 32 machines (Equation 2.2), indicating that the devices are near saturation. The experiment, therefore, agrees with the

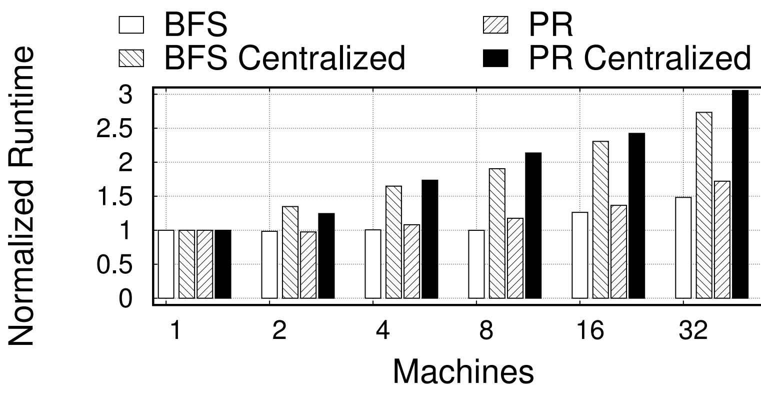


Figure 3.10 – Chaos vs. centralized chunk directory (weak scaling, RMAT-27 to -32, SSD).

theory. Further, Equation 2.3 tells us that even if we increase the number of machines in the deployment, this choice of settings means that we cannot drop below 99.3% given a fixed latency on the network. The increased runtime past this choice of settings can be attributed to increased queuing delays and incast congestion.

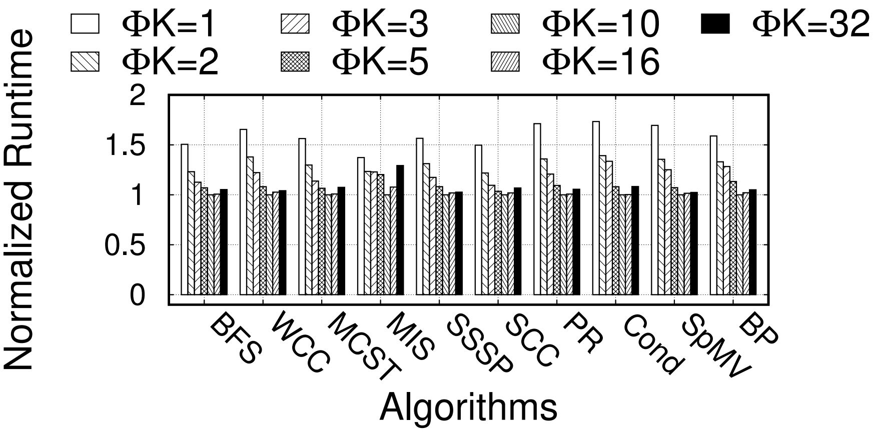


Figure 3.11 – Runtime as a function of batch sampling factor (32 machines, RMAT-32, SSD) normalized to Chaos ($\Phi K = 10$).

3.10.2 Dynamic Load Balancing

Chaos balances the load between different computation engines by randomized work stealing.

Figure 3.12 shows a breakdown of the runtime of the weak scaling experiments at 32 machines in three categories: graph processing time, idle time, and time spent copying and merging. The first category represents useful work, broken down further into processing time for the partitions for which the compute engine running on the machine is the master and processing time for partitions initially assigned to other compute engines. The idle time reflects load imbalance, and the copying and merging time represents the overhead of achieving load balance.

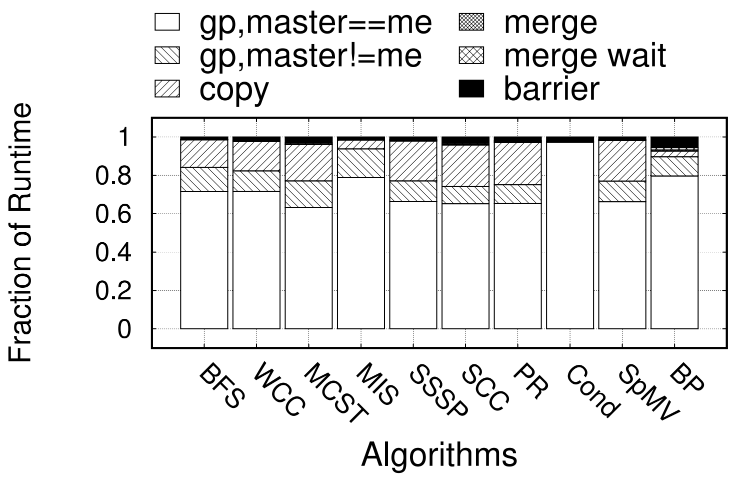


Figure 3.12 – Breakdown of runtime (32 machines, RMAT-32, SSD).

The results are somewhat different for different algorithms. The processing time ranges from 74 to 87% with an average of 83%. The idle time is very low for all algorithms, below 4%. The cost of copying and merging varies considerably, from 0 to 22%, with an average of 14%. Most of the idle time occurs at barriers between phases. Overall, we conclude from Figure 3.12 that load balancing is very good, but comes at a certain cost for some of the algorithms.

Next, we evaluate the quality of the decisions made by the stealing criterion we describe in Subsection 3.5.3. To do this, we introduce a factor α in Equation 3.2 as follows:

$$\frac{V}{B} + \frac{D}{B(H+1)} < \alpha \frac{D}{BH}$$

Varying the factor α allows us to explore a range of strategies.

- No stealing: $\alpha = 0$
- Less aggressive stealing: $\alpha = 0.8$
- Chaos default: $\alpha = 1$
- More aggressive stealing: $\alpha = 1.2$
- Always steal: $\alpha = \infty$

Figure 3.13 shows the running times for BFS and PageRank. The results clearly show that Chaos (with α=1) obtains the best performance - providing support to the reasoning of Subsection 3.5.4.

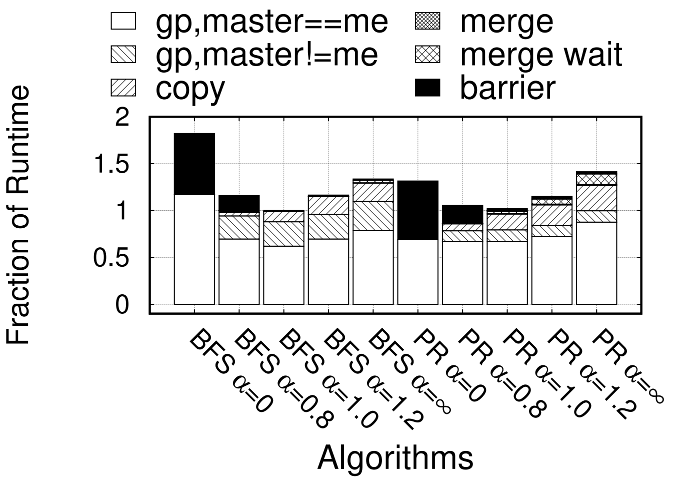


Figure 3.13 – Breakdown of runtime with work-stealing bias. 32 machines, RMAT-32, normalized to $\alpha = 1$.

As additional evidence for the need for dynamic work sharing, we compare the performance of Chaos to that of Giraph [1], an open-source implementation of Pregel, recently augmented with support for out-of-core graphs. Giraph uses a random partitioning of the vertices to distribute the graph across machines, without any attempt to address load balancing (similar to the experiment reported in Figure 3.13, with α equal to zero).

Out-of-core Giraph is an order of magnitude slower than Chaos in runtime, apparently largely due to engineering issues (in particular, JVM overheads in Giraph). To eliminate these differences and to focus on scalability, Figure 3.14 shows the runtime of both Chaos and Giraph on BFS and PageRank on RMAT-27, normalized to the single-machine runtime for each system. The results confirm that the static partitions in Giraph severely affect scalability.

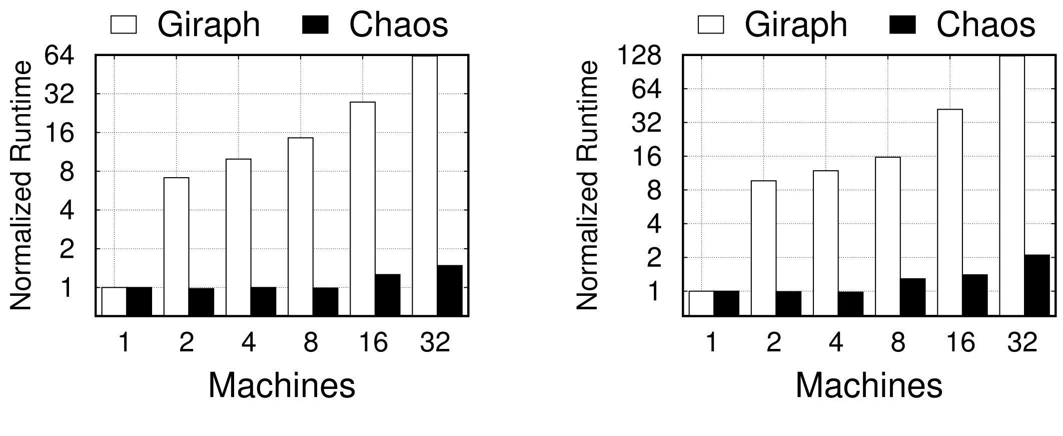


Figure 3.14 – BFS and PageRank runtime for Chaos and Giraph, normalized to the 1-machine runtime of each system.

3.10.3 Partitioning For Sequentiality Rather Than For Locality and Load Balance

An important question to ask is whether it would have been better to expend pre-processing time to generate high-quality partitions to avoid load imbalance in the first place instead of paying the cost of dynamic work sharing to improve load balance. To answer this question, we compare, for each algorithm on 32 machines, the worst-case dynamic load balancing cost across all compute engines to the time required to partition the graph initially. We use PowerGraph's [89] grid partitioning algorithm, which requires the graph to be in memory. We lack the necessary main memory in our cluster to fit the RMAT scale-32 graph that Chaos uses on 32 machines. Therefore, we run the PowerGraph grid partitioning algorithm on a smaller graph (RMAT scale-27) and assume that the partitioning time for PowerGraph scales perfectly with graph size. As Figure 3.15 shows, Chaos dynamic load balancing out-of-core takes only a tenth of the time required by PowerGraph to partition the graph in memory. From this comparison, carried out in circumstances highly favorable to partitioning, it is clear that dynamic load balancing in Chaos is more efficient than upfront partitioning in PowerGraph. Chaos, therefore, achieves its goal of providing high-performance graph processing while avoiding the need for high-quality partitions.

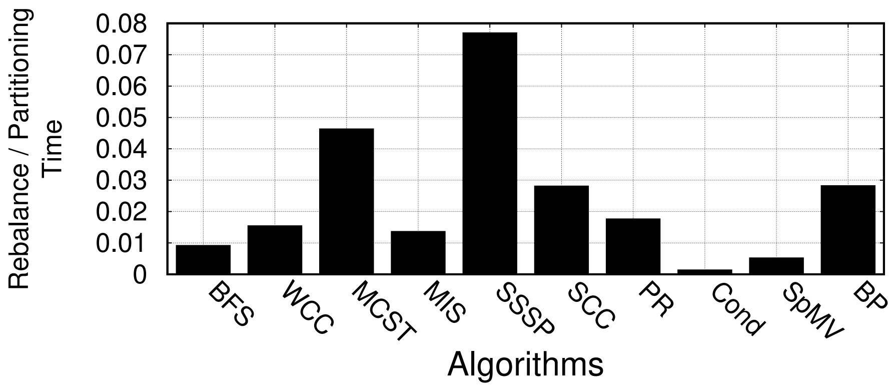


Figure 3.15 – Runtime for Chaos dynamic load balancing vs. PowerGraph static partitioning (RMAT-27)

3.11 Ragnarok: Towards Petascale Graph Processing[4]

In this section, we present and discuss *Ragnarok*, an extension to Chaos that allows the system to process graphs in the order of petabytes using only a small fraction of the input size as spare storage capacity. In its original design, Chaos requires an aggregate storage capacity that is multiple times (up to 3×) the size of the input graph to create, store, and process streaming partitions. As a result, cluster operators must provision more storage than necessary to store the unprocessed input, the partitioned input, the updates generated by the gather phase, and the vertex state. When the input graph is very large, this is undesirable and costly.

[4]The text and experiments in this section correspond to work performed after Chaos was published and are not part of the original paper [143].

Processing input graphs almost as large as the available storage capacity without running out of space requires that any data be either only read or read and immediately replaced. We define a new processing mode that merges edge and update sets for a streaming partition into a single file. Each entry in this file contains an edge and an optional update to be applied to the edge's source vertex. In this new processing mode, we combine the gather and scatter phases into a single phase, where the system reads in each entry from the streaming partition file, applies the update (if available), and immediately streams out the reversed edge and a new associated update. Reversing the edge ensures that the new update is applied to the destination vertex in the next iteration.

Consider, for example, the BFS algorithm executed in Ragnarok mode. In each iteration, the system marks edges with a flag, i.e., an update that indicates whether the edge is part of the BFS tree. When reading from a streaming partition, if a flag is present alongside the edge, the system updates the corresponding vertex state to mark the vertex visited and then drops the edge from the stream as it is no longer needed. If there is no flag, the system checks whether the source vertex is discovered, and, if so, it writes out the reversed edge along with a flag to mark the destination visited in the next iteration. Finally, if there is no flag and the source vertex is not marked, the system inserts the edge back in the stream without a flag. Since the BFS algorithm need not revisit an edge after its associated vertices have been visited, we drop all flagged entries after they are processed. As a result, the merged file for each partition shrinks over time to only the set of edges that can still be part of the BFS.

By using a single operator combining gather and scatter, we ensure that the storage capacity remains bounded because each entry read corresponds to a single entry written. Since we reverse source and destination vertices, new entries are usually written to a different partition, and therefore cannot be written in-place to the same file. We modify the storage subsystem in Chaos to implement circular files that automatically collapse the parts of the file that have been read while appending new entries. Also, Ragnarok mode supports undirected algorithms such as BFS by only storing each edge in one direction. In each iteration, the system makes two passes over the merged edge and update streaming partitions, reversing edges in each pass to access both endpoints.

Table 3.2 shows the latest top 10 ranking by capacity of the Graph500 benchmark for BFS. Since June 2016 and to this day, Chaos holds the third position in the ranking for successfully processing RMAT-40, a synthetic graph input of 250 terabytes containing 16 trillion edges and 1 trillion vertices. This experiment took a little over 10 hours to finish using 20 of the commodity machines described in 3.8. By comparison, all machines in the top 10 are supercomputers or custom-designed hardware. This result demonstrates the ability of scatter systems to scale to very large datasets by efficiently using few resources in a load-balanced manner.

Rank	Scale	Machine	Location	Machines	Cores
1	64T	IBM S922LC	DOE/NNSA/LLNL	2048	524288
2	32T	BlueGene/Q	Lawrence Livermore	98304	1572860
3	16T	Chaos (DALCO)	EPFL	20	128
4	16T	Cray CS300	Lawrence Livermore	300	38400
5	16T	IBM POWER9	Oak Ridge	2048	1048576
6	16T	Sunway MPP	Wuxi	40768	1304580
7	16T	BlueGene/Q	Argonne	49152	786432
8	16T	BlueGene/Q	Argonne	49152	786432
9	16T	Custom	RIKEN AICS	82944	1327100
10	8T	ThinkSystem SD530	Leibniz Rechenzentrum	4096	393216

Table 3.2 – Graph500 ranking by capacity (Nov 2019) [3].

3.12 Summary

Chaos is a system for processing graphs from the aggregate secondary storage of a cluster. It extends the reach of small clusters to graph problems with edges in the order of trillions. With very limited pre-processing, Chaos achieves sequential storage access, computational load balance and I/O load balance through the application of three synergistic techniques: streaming partitions adapted for parallel execution, scatter storage, and work stealing, allowing several machines to work simultaneously on a single partition.

We have demonstrated, through strong and weak scaling experiments, that Chaos scales on a cluster of 32 machines, and outperforms Giraph extended to out-of-core graphs by at least an order of magnitude. We have also quantified the dependence of Chaos' performance on various design decisions and environmental parameters. Finally, we showed how Chaos can process very large graphs by taking third place in the Graph500 capacity ranking for BFS.

We summarize the characteristics of Chaos and the scatter architecture techniques used below.

Area	Graph processing
Load imbalance	Power-law vertex degree distribution Uneven processing
Data spreading	Vertices as files Edges and updates as bags
Dynamic parallelism	Dynamic work sharing based on work stealing + merging
Partitioning	Bucket edges and updates by streaming partition
Misc. characteristics	Checkpoint-based failure recovery Petascale extension (Ragnarok)

4 Skew-Resilient, General-Purpose Analytics with Hurricane[1]

In the previous chapter, we described an analytics system for large graphs. We now examine Hurricane, a general-purpose, skew-resilient cluster analytics framework as a second application of the scatter architecture.

4.1 Introduction

Application runtimes in data analytics frameworks are unpredictable and underperforming on specific input datasets and software/hardware configurations. These issues often occur because different tasks within a job take different amounts of time to complete, causing a load imbalance where some machines sit idle while waiting for others to finish, thereby limiting the achievable degree of parallelism. Slower tasks can degrade performance for the entire parallel job, resulting in delayed job completion [43], resource under-utilization [92, 162], and even application crashes.

Task runtime variance is caused by skew. Tasks may be assigned different amounts of data due to data skew in the partitioning [106, 140]. Such skew occurs intrinsically in many real-world datasets, making it hard to create well-balanced partitions. For example, a web dataset may have millions of records referring to a website, map-reduce algorithms have popular keys, and social networking and graph datasets have high degree vertices. Tasks may also suffer from compute skew, wherein the execution time depends on the data, regardless of its size. For instance, an algorithm may do more processing on some inputs or selectively filter data [88]. Besides data and compute skew, task runtime can also be affected by machine skew, for example, heterogeneous or faulty machines [40]. A search for "skew" in analytics workloads on stackoverflow [17] yields hundreds of relevant results from programmers experiencing painful problems and unexpected crashes due to improper handling of skew.

[1]This chapter is based on the following publication: Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock You Like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the 13th EuroSys Conference*, EuroSys '18, pages 1–15. ACM, 2018. The author of this thesis is the creator of this system. The experimental evaluation was done collaboratively with Jasmina Malicevic and Nicolas Schiper. The author maintains the software to this day.

This chapter introduces Hurricane, a high-performance analytics system inspired by the scatter architecture described in Chapter 2 that achieves fast execution times and high cluster utilization with an adaptive task partitioning scheme. The core idea underlying this scheme is dynamic work sharing through *task cloning*, where an overloaded machine can clone its tasks on idle machines, and have *each clone process a subset of the original input.* This allows Hurricane to adjust parallelism adaptively *within* a task, and dynamically improve load balance across machines based on observed load, *at any point in time.* This is the key to handling load imbalance: underperforming tasks can be split across multiple machines *dynamically during their execution*, and idle machines can pick up a part of the task load.

By comparison, state-of-the-art frameworks such as Hadoop [93] and Spark [160] struggle to achieve load balance and good parallelism because they rely on data locality and static partitioning of work. Partitions are created based on storage blocks or programmer-defined split functions and assigned to machines. However, the partition sizes and processing requirements often depend on the dataset and are thus known only at runtime. Once partition bounds are fixed, the degree of parallelism for a stage cannot be dynamically adjusted: it is not possible to split the work or increase parallelism within a partition when it takes a long time to process, immaterial of the reason it takes that long. For this same reason, while traditional straggler mitigation techniques such as speculative execution [74] and tiny tasks [129] can help with slow machines, they do not directly address data or compute skew.

Hurricane supports task cloning by combining two novel techniques: *fine-grained independent access to data* and *programming model support for merging.* These techniques enable writing high-performance, skew-resilient applications that automatically achieve load balance with minimal programmer effort.

Fine-grained data access enables workers[2] executing tasks to compute on small partitions of *any* input or intermediate data independently of other workers, allowing fine-grained, on-demand task cloning. Hurricane stores *all* input and intermediate data in *data bags* (§2.1.5). Each data bag corresponds to the input or output of a task. A bag does not belong to a worker; rather, all workers executing clones of the same task share the bag.

Hurricane supports combining the partial outputs of cloned tasks using an application-specified *merge* procedure whose output is equivalent to the output of a single uncloned task. Our merging paradigm is more general than the traditional *shuffling and sorting* method for combining outputs from different partitions. It not only alleviates the need to sort, but also allows for data records associated with the same key to be simultaneously processed on multiple machines, providing more flexibility to balance load across partitions in the presence of key skew.

Hurricane uses scatter storage techniques to ensure efficient cloning of tasks by spreading the chunks in data bags uniformly randomly across all machines in the cluster and allow retrieving

[2]A worker is a container executing a task on a machine.

them efficiently using a decentralized scheme. This approach achieves high cluster-wide storage load balance, utilization, and throughput, thus ensuring that cloning does not lead to storage bottlenecks.

We have implemented several typical analytics applications on Hurricane. We evaluate the system on a cluster of 32 machines that are connected by a high-speed network. We show that Hurricane achieves load balance and scales with the number of machines in the cluster, the input data size, and the amount of skew. We observe a slowdown compared to uniform partitions of at most 2.4× in a click counting application in the presence of 64× imbalance between partitions. Hurricane can execute skewed hash joins 18× faster than Spark, while keeping the performance degradation with high skew below 2.3×, and outperforms Spark's GraphX [90] by calculating PageRank on real-world graphs 5-10× faster.

The contributions of this work are four-fold:

- We present the first analytics system designed to systematically perform adaptive partitioning of work based on observed load during execution, allowing it to improve application runtime by adaptively optimizing parallelism and providing load balance for both compute and storage resources (§4.2).
- We demonstrate how to implement such a system through a fine-grained, adaptive partitioning scheme based on a task cloning abstraction (§4.3.2).
- We maximize storage utilization and throughput while allowing workers to efficiently and independently access data (§4.3.3).
- We demonstrate Hurricane's performance for several typical analytics workloads (§4.5).

4.2 Programming Model

Hurricane supports a batch processing model. To achieve good parallelism and load balance even in the presence of high skew, Hurricane clones tasks based on the observed load at any point during their execution. This requires programming model support, specifically the ability for multiple workers to dynamically *share* the work (and data) in a partition, as well as the ability to reconcile multiple partial outputs into a single consistent output.

4.2.1 Application Model

Hurricane applications are specified as a directed graph of tasks, shown as circles, and data bags. The edges in the graph represent the flow of data between tasks and bags, i.e., the outputs of bags are connected to the inputs of tasks, and the outputs of tasks are connected to the inputs of bags. Although the application graph does not allow loops, it is possible to implement many iterative-style applications, such as PageRank [134] or k-means clustering [95] by creating multiple identical phases connected back-to-back in the graph.

Executing the application graph creates an execution graph, where machines in the cluster execute the various tasks on local workers. A worker can either execute a task or a clone of a task. The system ensures that tasks only start once their corresponding input bags are ready.

The Hurricane framework may *at any point* decide to clone a task to increase parallelism and ensure faster completion. The system entirely manages cloning tasks. When partial outputs from clones must be reconciled, the application specifies a merge procedure to combine them. If no such procedure is specified, Hurricane simply concatenates the outputs of all clones.

Figure 4.1 shows the graph topology of a typical Hurricane application called ClickLog that operates on a log of clicks on advertisements to count the number of unique IP addresses from each geographic region. This application uses three types of tasks, whose pseudo-code is shown in Listing 4.1. Phase 1 tasks map the source bag, containing the click log, into per-region output bags, Phase 2 tasks list the unique IP addresses in each region bag, and Phase 3 tasks count the size of the list.

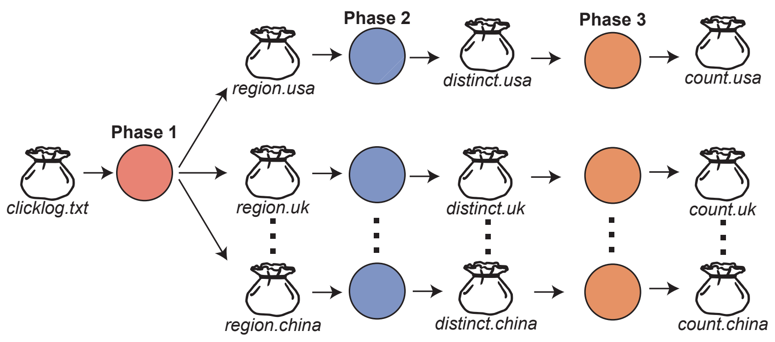


Figure 4.1 – ClickLog computation graph.

Figure 4.2 shows a possible execution graph using 4 machines. All workers execute tasks or cloned tasks, shown using dashed lines. In this example, the number of clicks on advertisements per region can vary significantly, causing skew in the tasks. As a result, Hurricane may decide to clone some tasks. For example, Phase 1 has one original worker executing the task and 3 clones. A task and its clone run the same code. The Phase 2 task operating on the USA region has two workers associated with it (the original worker and a clone). Phase 2 requires a custom merge, which is executed after all associated workers finish. Note that different tasks (not clones) may also run the same code. For example, Phase 3 has three different tasks, running the same code, but with different input bags.

4.2.2 Dynamic Fine-Grained Data Sharing

Multiple workers (clones) executing the same task on the same input data require a way to obtain disjoints subsets of the data. Since Hurricane may adjust the number of clones dynamically during task execution, workers should be able to independently and efficiently access finer-grained partitions of the data dynamically at runtime.

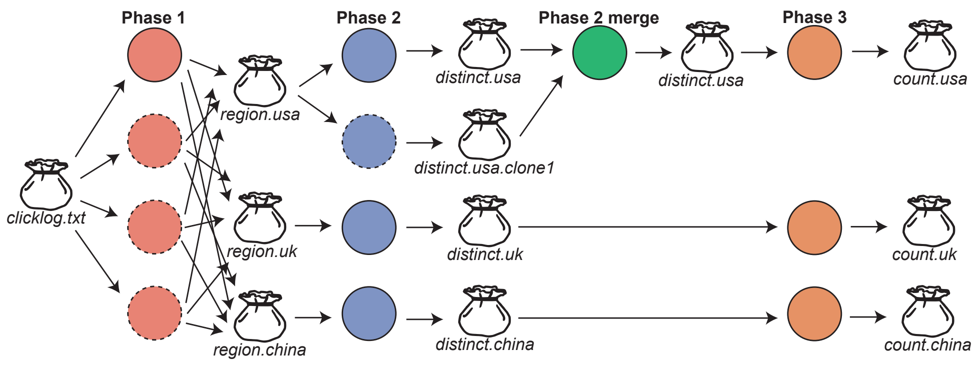


Figure 4.2 – A possible ClickLog execution graph on 4 machines. Hurricane automatically cloned the phase 1 task as well as the phase 2 task for the USA region. Note that cloning a phase 2 task requires the introduction of a corresponding merge.

Hurricane achieves fine-grained data sharing through a data bag abstraction (§2.1.5) that workers use to store data and communicate with each other. Each data bag contains fixed-size blocks of data called chunks that are stored in files in our distributed storage service.

```
Phase 1 task (input, outputs):
  while ip = input.remove() not empty:
    region = geolocate(ip)
    outputs[region].insert(ip)

Phase 1 merge (partial1, partial2, output):
  output = concat(partial1, partial2) // default merge

Phase 2 task (input, output):
  let distinct be a bitset
  while ip = input.remove() not empty:
    distinct |= ip // set corresponding bit to 1
  output.insert(distinct)

Phase 2 merge (partial1, partial2, output):
  output.insert(partial1 | partial2)

Phase 3 task (input, output):
  output.insert(len(input))

Phase 3 merge (partial1, partial2, output):
  output.insert(partial1 + partial2)
```

Listing 4.1 – ClickLog application code.

A worker serializes its application-specific data records into a chunk before inserting it into a bag. Similarly, after removing a chunk, it deserializes the chunk into its data records. Hurricane provides several typed iterators for serializing and deserializing common formats (integers, floats, strings, tuples, etc.), which can be combined to represent more complex data types (e.g., nested tuples). All serializers ensure that data records do not cross chunk boundaries, thus allowing chunks to be processed independently.

As described in Subsection 2.1.5, data bags allow multiple workers to concurrently insert or remove chunks from the same bag without interference. For instance, in Figure 4.2, the two workers processing the Phase 2 for the USA region read chunks from the same bag (`region.usa`). This property derives from the use of chunks as the basic indivisible unit of data used by workers. Data bags also support data processing at varying speeds by forcing workers to request individual chunks instead of being assigned key ranges upfront. This *late binding* of data chunks to workers is essential to handle skewed workloads as it makes it possible to partition the data dynamically during task execution.

4.2.3 Dynamic Merge-Based Task Sharing

Multiple workers (clones) executing the same task on different subsets of the same input data may need a way to reconcile their partial outputs into a single coherent output. Ideally, workers should be able to process subsets of the data in isolation, and produce individual outputs that can be *merged* to produce the final output.

Hurricane merges partial outputs through a (possibly null) merge procedure. Some tasks can support multiple workers without any additional merging effort. Examples of such tasks include preprocessing, map tasks (from MapReduce), filters, selects (in SQL), etc. In such cases, it is sufficient to concatenate the chunks produced by each worker into the output bag. In the ClickLog example, this is what happens if multiple workers execute Phase 1. Other tasks, however, require support for merging the partial results of the concurrent workers. Examples of such tasks include reduce tasks (from MapReduce), counting, sketches [70, 84], groupby, etc. In the ClickLog example, this is the case for tasks in Phases 2 and 3. Often, tasks requiring a merge must produce output satisfying some constraints (e.g., sorted result, aggregation). As a result, an intermediate merge is required to ensure output consistency. Since merging is application-specific, if the task requires a merge, we require the programmer to specify a merge procedure as part of the code for that task.

Specifying a merge procedure amounts to defining a function to combine two partial outputs into one. This merge procedure is relatively easy to write. In most cases, it is of similar complexity as writing a merge combiner in Spark. However, unlike merge combiners, the merge operation is more general. Among other things, non-aggregation outputs can be merged, for instance, through a merge sort. The merge operation also supports non-commutative-associative operators (e.g., unique counts, medians, duplicates removal). For convenience, Hurricane provides a library of typical merge operations.

4.3 Design

Hurricane is based on the scatter architecture described in Chapter 2. Figure 4.3 shows the architecture and typical deployment of Hurricane. A Hurricane cluster consists of a set of compute and storage nodes. The cluster administrator provisions machines for use by Hurricane, either manually or through a resource manager, such as YARN [152]. The compute and storage nodes may be co-located, but are provisioned independently. Then each storage node starts a storage server, and each compute node starts a storage client that stores and accesses data from storage servers and is configured with the list of storage nodes. The compute nodes run tasks on local workers. Each compute node runs on one or more cores, and workers can be multi-threaded. The storage nodes store all bags spread across the machines. These bags can be of two types: data bags (DB) and work bags (WB). Work bags are used to schedule tasks.

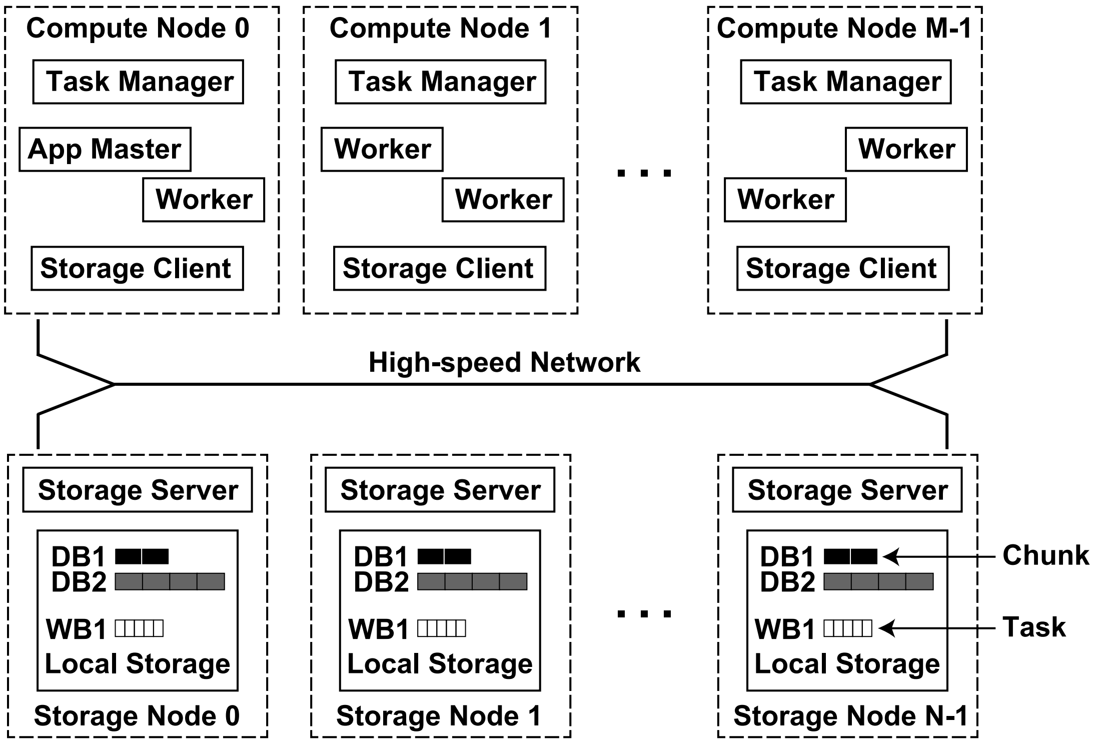


Figure 4.3 – The Hurricane Architecture. Although logically separated, compute nodes and servers can be co-located.

4.3.1 Execution Model

Each Hurricane application is associated with an application master that runs on one of the compute nodes. The master drives the application's computation by invoking functionality in the Hurricane framework. Also, it monitors application progress and facilitates the implementation of policies for cloning and resource management. The master is a lightweight component as it relies on distributed work bags to perform most of its functions.

Upon starting, the application master creates a task manager on each compute node that is responsible for executing tasks on local workers. The application master then reads the application graph and schedules tasks for execution. Each task consists of a task blueprint, containing a unique task identifier and the code necessary to execute the task, as well as the identifiers of its input and output bags.

The master maintains the application's progress in the execution graph and schedules new tasks once all the source bags for a task have completed. The overall execution ends once there are no more tasks to be scheduled, and no more tasks are being executed.

This execution model ensures that once an input bag becomes empty, it will remain empty, and thus workers know when they are done. For example, in Figure 4.1, the application master schedules the Phase 1 task, and only when it is finished, schedules all the Phase 2 tasks. Since Phase 3 tasks only depend on the bag containing the distinct list for their respective region, they can be scheduled immediately after the corresponding Phase 2 tasks finish. This simple model suffices for our batch analytics workloads, although we plan to explore more sophisticated dataflow execution models for streaming workloads [119].

4.3.2 Task Cloning

The application master automatically clones tasks on behalf of the application by modifying the execution graph to add a copy of the cloned task that reads from the same input bag as the original task, as well as (possibly) a merge task. When the task requires a merge, we add a merge task to the execution graph, when the first clone is created. Then, for each clone, the master creates a new bag dependency between the clone and the merge task. Once all the clones complete, we execute the merge task to produce the reconciled output.

Hurricane has two design goals when cloning tasks: automatically adjusting parallelism at runtime with minimal programmer effort and minimizing the overhead of executing a task on multiple workers. These goals present a trade-off between responsiveness to load imbalance and the cost of cloning and merging of results.

Dynamic Parallelism Hurricane clones a task repeatedly until it runs on every compute node, or the system determines that it already benefits from a sufficient degree of parallelism. The application master makes cloning decisions based on two criteria: 1) load information that helps detect task load imbalance, and 2) a cloning heuristic that determines whether cloning will benefit task execution time.

Hurricane detects load imbalance by monitoring two resources throughout the execution of a task: CPU load and network usage. If a worker experiences a high CPU load for a prolonged time or its network interface is saturated, this is an indication that the worker is experiencing overload, and we should re-evaluate the degree of parallelism in that task. The master then clones the task on an idle compute node if one is available, and the cloning heuristic allows cloning, as discussed below.

Hurricane clones a task worker repeatedly until it is no longer overloaded, thus increasing task parallelism only as needed.

Note that Hurricane need not monitor for storage bottlenecks because the storage system is designed to provide the maximum possible bandwidth to applications, and therefore running at peak storage bandwidth is the best-case scenario.

Cloning Heuristic Hurricane only clones a task when it expects that cloning will improve task execution time. Cloning introduces two costs that may require additional computation and I/O: 1) loading task state in a new clone, and 2) merging of clone outputs, which introduces an additional dependency in the execution graph. Hurricane estimates these costs to avoid cloning close to task completion.

Consider the ClickLog example from Figure 4.1. When a Phase 1 worker is overloaded, the application master will always clone the task since it has minimal state and does not require a merge. This process will repeat for each worker in Phase 1 until the task completes, or there are no more idle compute nodes. In contrast, when a Phase 2 worker is overloaded, the heuristic may reject cloning if the task is close to completion because the overhead of merging outweighs the benefits. The heuristic may also determine that it is worthwhile cloning, when the task runs for a long time, as in the *region.usa* case. If so, the master performs task cloning by scheduling a copy of the task on an idle node, as it would any other task, and adds the corresponding merge task to the execution graph.

4.3.3 Storage Architecture

Hurricane decouples computation from data, storing data on storage nodes, while processing is performed on separate compute nodes. The storage architecture in Hurricane is based on the scatter storage architecture described in Section 2.1. We pool all storage devices together, split all bags into chunks, and store these chunks uniformly on all storage servers. This architecture is necessary to support task clones, which require efficient and fast access to their subsets/partitions of the input data.

Data Placement and Access Hurricane workers use this distributed storage service to store and access data, which helps avoid storage bottlenecks. Hurricane mostly relies on the bag abstraction (§2.1.5) to facilitate dynamic task partitioning when data does not need to be ordered. In rare cases where data must be ordered or obey other constraints, e.g., at the end of a processing pipeline, the system uses regular files.

Storage Load Balancing Hurricane relies on oversubscription to ensure storage utilization remains high and balanced throughout the execution. Workers use a batch sampling technique to maintain a fixed number of outstanding requests for blocks to storage servers through their client (§2.1.4).

Batch sampling also helps reduce the latency associated with removing items from bags that are close to empty. This latency is roughly $\frac{N \cdot L}{K}$, where M is the number of storage nodes, K is the batch sampling factor, and L is the round-trip latency of a single probe operation.

4.3.4 Adding and Removing Nodes

Hurricane allows dynamic addition and removal of compute and storage nodes from an application. This is easy to support for two reasons: 1) data is stored at storage nodes, separately from compute nodes, and 2) the compute nodes run independently of each other.

The application master can add and remove compute nodes at any point during job execution to accommodate variations in load. A compute node is added by starting the Hurricane framework and configuring the framework with a list of storage nodes and starting a task manager on the node. A compute node is removed by stopping its task manager after its current workers have finished.

The application master may also add or remove storage nodes during job execution. A storage node is added by starting a storage server on the node. The application master then informs the storage clients running compute nodes about the new storage server, allowing clients to place data there. When a storage node is removed, it stops accepting insert requests while still allowing remove requests. When all its bags become empty, the node can be removed.

4.3.5 Opportunistic Load Balancing

In addition to cloning, Hurricane supports an opportunistic load balancing mechanism that extends the power of combiners, used by current frameworks, for handling skewed data sets. For example, consider a word counting application with significant skew due to the presence of frequent words. A combiner combines the tuples for the same word, by summing their counts, thereby reducing skew at the reducer. However, combiners are executed by workers on the data path. As such, they risk overloading the workers processing "hot" partitions with combining overhead.

Hurricane provides, *tamed transformers*[3], a facility for an application programmer to provide “tamed” code (in the form of a thunk) to transform data in a bag. Tamed transformers extend combiners in two ways. First, they can be applied on the compute nodes when data is read or written to a bag, but they can also be applied on the storage nodes when data is at rest. Since data is spread across the storage nodes, any lightly-loaded storage node can apply the transformer. Second, tamed transformers are applied opportunistically. They are implemented using *promises* by inserting them into the task list of a thread pool so that they are executed given enough time and CPU capacity. Hurricane never stalls data transmission (to worker or storage node) by canceling the promise before transmission and sending the original data.

[3]A play on Combiners and Transformers in the namesake media franchise.

Unlike combiners, tamed transformers are not restricted to commutative and associative operations (where it is always safe to combine). They are allowed to combine across keys, and can change the output format of data records. For example, when there is no semantically correct way to combine a set of data records into a smaller set of data records, transformers can still be used to reduce the total amount of data by compressing data within a chunk. Such transformations require the definition of an inverse transformation. They should be used sparingly as they introduce a possibility of stalls since the inverse transformation must always be applied before supplying data to workers.

4.4 Implementation

This section describes the implementation of the various components in Hurricane.

4.4.1 Task Scheduling

Hurricane minimizes the overhead of task cloning by performing efficient low-latency scheduling through a reliable, distributed task queuing interface called *work bags.* Work bags are similar to data bags and expose the same interface, except they contain tasks, not data. Compute nodes remove tasks (including cloned tasks) from work bags to execute on local workers. Similar to data bags, tasks in work bags are distributed across all storage nodes and accessed by compute nodes independently without any single point of control. Unlike traditional scheduling queues, work bags are unordered, allowing for fast decentralized access to their contents.

Each application has three work bags associated with it, a *ready bag*, a *running bag*, and a *done bag*, corresponding to the ready, running, and exited, task states. Compute nodes remove tasks from the ready bag to create workers. Workers execute application code by removing fixed-size chunks from one or more input data bags, computing on the chunks, and then inserting transformed chunks in one or more output data bags. When a worker finishes executing, it inserts its task identifier in the *done* work bag. The application master monitors the done bag and inserts tasks into the ready bag once their dependencies have been completed. The running work bag is used for handling compute node failures.

4.4.2 Task Cloning

By default, Hurricane runs a single worker for an input bag. At any point, each compute node can signal the application master that it is overloaded and would like a particular task to be cloned to alleviate the load. The application master may accept or ignore the cloning request based on a cloning heuristic. We now consider the implementation of overload detection and the heuristic for cloning.

Detecting Overload A task overload can occur either when the task is CPU-bound or I/O-bound. For detecting a CPU bound task, we need to measure the CPU load on the machine simply. The disk or the network could limit an I/O bound task. Since we distribute the chunks in a bag across storage nodes, a disk-bound task will maximize storage bandwidth, helping us achieve our goal of improving the performance of large datasets with skew. Assuming high bisection bandwidth, a network bottleneck may occur when a node is limited by its endpoint bandwidth. We can detect such a bottleneck by measuring the network throughput at each node. As a result, a compute node generates a clone message periodically when the CPU or its local network interface is saturated. Currently, we send clone messages at least 2 seconds apart.

Cloning Heuristic Hurricane uses a simple heuristic to estimate whether cloning a task is worthwhile, using the following quantities: k is the number of clones processing a task, T is the expected time to finish the task without cloning, T_C is the expected time to finish the task with cloning, T_{IO} is the expected additional I/O time due to cloning, i.e., the time to read and load additional task state and the time to merge the clone's output data. It follows that $T_C = \frac{k}{k+1} T + T_{IO}$.

Given the above quantities, Hurricane clones a task if $T_C < T$, i.e., $\frac{k}{k+1} \cdot T + T_{IO} < T$, which simplifies to:

$$T > (k+1) \cdot T_{IO} \tag{4.1}$$

In other words, cloning is worthwhile when the time to finish the task without cloning is greater than the product of the number of clones and the I/O time resulting from cloning. For example, assume a task is expected to finish in 10 seconds with 4 clones, and the clones are overloaded. Adding a fifth clone brings down the completion time to 8 seconds. So the cloning overhead cannot be more than 2 seconds, or else it will likely delay task completion. The cloning heuristic avoids cloning close to the end of a task, and so we only need a rough estimate of these quantities.

The application master knows the value of k. T is estimated by sampling the input bag on a few storage nodes to estimate how much data is left and how fast it is emptying. While T_{IO} is application-specific, we estimate it as two times the size of the remaining portion of the input bag that the task will read (for input and output).

4.4.3 Storage Nodes

Storage nodes provide storage for data bags and work bags. Hurricane implements the bag API as described in Subsection 2.1.5 Bags are stored as files and stored at each storage server as Linux ext4 [114] buffered files. In addition to the `insert` and `remove` operations, the implementation includes other operations such as reusing the contents of a bag, sampling the amount of data remaining in a bag, and garbage collecting a bag.

4.4.4 Fault Tolerance

Fault tolerance is especially important for complex application graphs that must process large amounts of data.

The application master provides a single point of control for the application's execution. This component is application-specific, while all other Hurricane components are application agnostic, and run independently of each other. A consequence of this design is that the crash of a compute node does not interfere or block any other compute node from making progress since compute nodes are not aware of each other. Similarly, the crash of a storage node does not prevent other storage nodes from serving/storing data. Hurricane applications place all persistent state at the storage nodes, while computes nodes contain only soft state. This approach allows us to use a simple checkpoint-replay mechanism for handling compute node failures and primary-backup replication for storage node failures.

Application Master Failure The application master is the only entity that knows about the state of the computation. However, this state is stored in its work bags that are stored on the storage nodes. When the application master fails, we restart it and *replay* the done work bag. Replaying the done work bag involves rereading the entire bag, taking note of each completed task to update the execution graph. Replaying these task completions lets the application master recover the state of the execution graph to its pre-failure state. Once replay completes, the application master resumes normal operation. Neither compute nodes nor storage nodes need to be aware of an application master failure and can continue to execute tasks normally.

Short-lived application master crashes will not usually cause application slowdown as compute and storage nodes can proceed independently of the application master because the latter is only required to schedule new tasks or to clone existing tasks. Nonetheless, cluster operators may opt to replicate the application master using Apache ZooKeeper [97] for increased resilience to failures.

Compute Node Failure When a compute node fails, the application master restarts all currently running tasks on the node. To do so, it scans the running work bag for all tasks that the compute node was currently executing. It then terminates all running clones of these tasks. Next, it discards data in the output bags and rewinds the input bags of these tasks, and finally reschedules them by moving them back to the ready bag. From the application's perspective, restarting failed tasks from scratch enables maintaining the exactly once invariant when reading data from input bags.

Our approach is simple to implement and reason about but comes at the expense of potential slow progress in the presence of many failures. We leave the implementation of a more fine-grained recovery approach to address compute node failures that avoids restarting associated clones as future work.

Storage Node Failure Hurricane protects against storage failures using redundancy techniques such as RAID [136]. Besides, the system supports primary-backup replication of the data stored on storage nodes. Since data is spread across all storage nodes, an application can tolerate n storage node failures by using $n+1$ replication. Each bag, including data and work bags, is replicated along with bag state, such as the current file pointer position from which the next chunk will be read. The replication level is configurable for each application. In the event of a storage node failure, the application master informs each compute node to use a backup storage node. Compute nodes re-issue requests to the backup storage node and proceed as usual.

Decentralized Control In the current version of Hurricane, the application master serves as a centralized control plane. On the other hand, the data plane is fully decentralized. While we do not foresee any scalability bottlenecks as a result of this decision, we leave the implementation of a decentralized application control plane for future work.

Speculative Execution Hurricane does not currently provide a mechanism for speculative execution [74]. We do not attempt to restart crashed, hung, or slow tasks on compute nodes speculatively. Crashed or hung tasks will eventually be detected by the application master and will be killed and restarted then. Cloning successfully mitigates stragglers, as slow tasks will eventually be cloned, but this is not done speculatively. We leave the implementation of speculative cloning as future work.

4.4.5 Software and Configuration

Hurricane is written in Scala and runs on a standard Java Virtual Machine (JVM) version 8 [12]. The system and all benchmark applications are roughly 7'000 lines of code. Similarly to all scatter systems, Hurricane disaggregates compute and storage logically by running separate JVM processes for storage and compute nodes. We use the Akka toolkit (version 2.4) [13] for high-performance concurrency and distribution. We use TCP-based netty (version 4) [14] for inter-machine communication, but also support UDP-based Aeron (version 1.2) [15] for high-throughput low-latency messaging. Due to observed instabilities in Aeron, we opted for netty as the default. We use reasonable defaults for all system parameters. In particular, we do not tune the network stack, the storage subsystem, or the JVM.

The chunk size is chosen to minimize the overhead of remote data access, reduce internal fragmentation caused by small bags, and minimize random accesses to the disk. Our system uses a 4 MB chunk size.

4.5 Evaluation

This section evaluates the performance of the Hurricane system and compares it with the Hadoop (version 2.7.4) and Spark (version 2.2.0) systems. Our experiments show the effect of increasing skew and input data size on the system. Then, we evaluate the various design choices we made in Hurricane. Finally, we evaluate the performance of three realistic applications.

We evaluate Hurricane on 32 16-core machines (2 Xeon E5-2630v3), each equipped with 128 GB of DDR3 ECC main memory. The machines have two 6 TB magnetic disks arranged in RAID 0. The RAID array sustains a bandwidth of approximately 330 MB/s, as reported by fio [8]. The machines are connected through 40 GigE links to a top-of-rack switch that provides full bisection bandwidth. We run no other workload on the machines to ensure that each system can fully utilize the 128 GB of main memory. We co-locate compute nodes and storage nodes and run one storage node per machine using all available storage. We do not enable replication of bags unless explicitly stated.

4.5.1 Taming Skew

We first evaluate how well Hurricane can deal with skewed workloads along two dimensions: increasing skew in the data, and increasing input data size. To do so, we use the ClickLog application presented in Subsection 4.2.1. This application is representative of many analytics workloads, such as the MapReduce paradigm that transforms input data before aggregating it along some dimension.

The input takes the form of text files uniformly distributed across all storage nodes. Each input line contains an IP address. The output is the count of the number of unique IP addresses in each geographic region. We simulate the geolocation function to avoid external API calls.

For the evaluation under skew, we normalize the skew runtimes with the corresponding runtimes for uniform inputs. Table 4.1 establishes the baseline ClickLog runtimes on uniform inputs with increasing size. We start with 320 MB (10 MB per machine) and multiply the size by 10 until the input size is 3.2 TB (100 GB per machine). At 10 MB, 100 MB, and 1 GB per machine, the experiment runs from memory and the performance scales sub-linearly due to execution overhead. The 320 GB (10 GB per machine) and 3.2 TB (100 GB per machine) runs execute from disk and scale almost linearly at aggregate disk bandwidth.

Input size	**320 MB**	**3.2 GB**	**32 GB**	**320 GB**	**3.2 TB**
Runtime	5.7s	8.9s	22.8s	90s	959s

Table 4.1 – ClickLog runtime over a uniform input (baseline). The total size of the input is scaled from 320 MB to 3.2 TB of total input.

To evaluate performance in the presence of skew, we use a synthetic input generator that takes two parameters: *input size* and *skew*. We use a Zipfian distribution with parameter s ($0 \le s \le 1$) to obtain different amounts of skew. Then we generate partitions by dividing the key range into equal parts so that adjacent keys are placed in each partition.

We show how increasing the skew affects Hurricane. We introduce increasing skew in the input data using skew parameter s, with values 0 (uniform), 0.2 (mild skew), 0.5 (medium skew), 0.8 (medium high skew), and 1 (high skew). The corresponding imbalance between the largest and smallest region is 1×, 2.3× , 8×, 28×, and 64×.

Given $s = 1$, the largest region makes up 19.6% of the total input. Using Amdahl's law, and assuming that the largest region is the serial (non-parallelizable) fraction of the parallel execution and that the processing requirements are uniform, we can estimate that the maximum achievable speedup in this scenario is 4.5× when the largest region is not broken up. When using 32 machines, this corresponds to a best-case slowdown of 7.1× (32/4.5).

Figure 4.4 shows Hurricane's slowdown with increasing skew and input sizes. We observe that Hurricane suffers at most 2.4× slowdown across all configurations and significantly less in most cases. By spreading data chunks across all storage nodes and cloning tasks processing large regions to split the work at runtime, Hurricane achieves a much better slowdown than 7.1×.

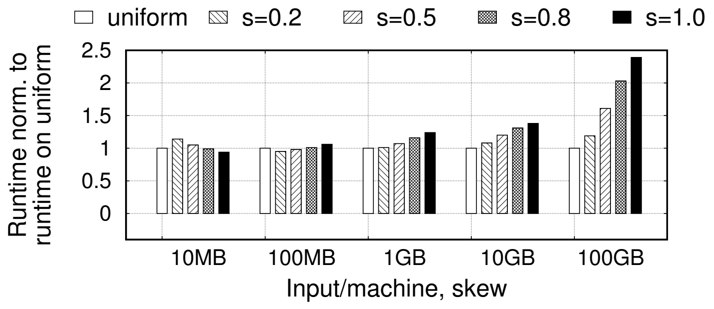


Figure 4.4 – ClickLog runtime with increasing skew.

In Figure 4.4, the normalized runtime increases with increasing skew due to task cloning and merging overheads. Tasks are cloned every two seconds, and so it takes some time until all compute nodes are busy (e.g., in Phase 1). The merge operation introduces overheads because it reconciles the partial outputs of clones after their execution. There is no task cloning (and therefore no merging) for the first two input sizes (10 MB and 100 MB). These experiments run fast due to the small input size and have little overhead caused by skew. The third input size (1 GB) experiences some task cloning in the presence of skew. In the worst case ($s = 1$), this overhead is 0.24×, of which 63% is due to cloning delays, and the rest is from merging partial outputs. Experiments for the 10 GB input size lead to a significant amount of task cloning for large skew ($s = 0.8$ and $s = 1$). The worst-case overhead is 0.38×, of which 39% is due to cloning delay in the first phase, and the rest is from merging partial outputs. Figure 4.8 shows these effects by plotting the sustained throughput over time when the skew s=1, for 10 GB input size. Note that as the input sizes become larger, the application executes for a longer time, and therefore the relative overhead due to cloning delay decreases.

The largest input size (100 GB) suffers from the largest overhead across all experiments, 1.4× for $s = 1$. Unlike smaller input sizes, half of this overhead is due to desynchronized garbage collection pauses at storage nodes, which prevents the system from achieving peak I/O throughput [110]. We are actively looking into this problem, and expect our solution to bring the overhead down to similar levels as that of smaller input sizes.

4.5.2 Design Evaluation

Varying Partition Sizes We now evaluate how decreasing the partition size, i.e., creating more tasks of smaller sizes and scheduling them statically without cloning affects the runtime for a skewed workload. To that end, we run Hurricane with and without cloning (dubbed HurricaneNC) on a 32 GB input with skew parameter $s = 1$. We increase the number of partitions from 32 to 4096 so that the average task size decreases with more partitions. The average task size with 32 partitions is 1 GB, whereas, with 4096 partitions, it is 8 MB (comparable to the chunk size).

Figure 4.5 shows the results for this experiment. We break down the runtimes of each phase. The first phase buckets the IP addresses into regions, and the second phase uses a bitset to list unique IP addresses, while the third phase counts the size of the bitset. Hurricane starts with a single worker in Phase 1, which it can clone based on load conditions, while HurricaneNC always runs a single worker per task since it does not clone workers. To ensure a fair comparison for HurricaneNC, we split the Phase 1 input into equal-sized partitions such that each compute node is assigned at least one partition of the input.

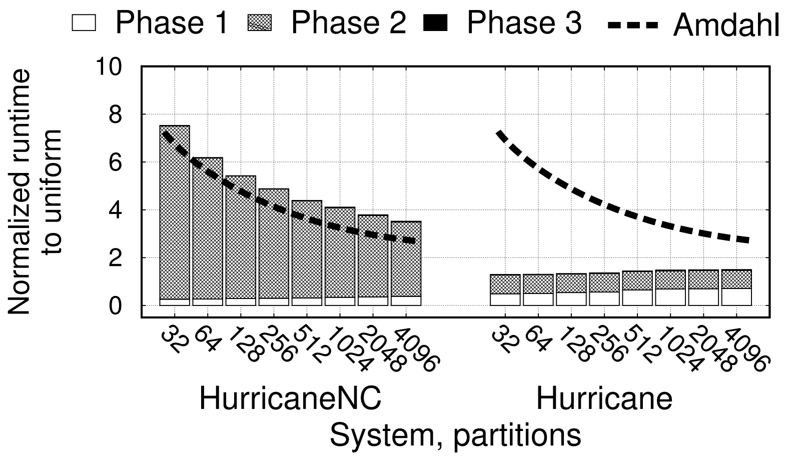


Figure 4.5 – HurricaneNC (no cloning) and Hurricane with increasing number of partitions on an input of 32 GB with skew $s = 1$. Dashed lines represent the best case slowdown using Amdahl's law.

There is no skew in Phase 1, and thus the size of tasks has little impact on the phase's runtime. We observe that Hurricane takes a little longer to complete Phase 1 because it starts the phase with a single worker, and on-demand cloning introduces some delay for detecting overload. However, the benefit of Hurricane's approach is that the application does not need to specify the correct number of clones. Phase 2 has significant skew, and here we observe how cloning reduces the phase's runtime, even though it comes at the expense of a merge. Hurricane can parallelize the processing of large partitions through cloning, whereas HurricaneNC's runtime is dominated by the time it takes to process the largest partition on a single worker. Phase 3 runs very quickly as it does little work.

We plot the best case slowdown as computed in Subsection 4.5.1 in dashed lines as a reference. We observe that HurricaneNC's performance closely matches the curve, whereas Hurricane stays below. The shape of the results for HurricaneNC indicates that its speedup becomes less significant every time we double the number of partitions until, eventually, it cannot achieve a better runtime.

We conclude from these results that smaller partitions alone are insufficient for addressing skew: even though the average partition size decreases, large partitions remain comparatively large. In the absence of cloning, a single worker must process the largest partition sequentially, and so the system cannot fully leverage the presence of more tasks to achieve better load balancing. Finally, we observe that creating too many small partitions introduces scheduling and storage overheads, as evidenced by the increase in runtime for Phase 1 for both systems.

Cloning and Spreading We now seek to evaluate which feature of Hurricane works best to address skew, and in particular, whether both cloning and spreading data across storage nodes are necessary for good performance. We only present the runtime for the first two phases, since the third phase runs for a short time.

We consider four configurations of Hurricane with different features turned off:

- Configuration 1: *Cloning Off, local data.* Cloning is disabled. We create one task per bag, i.e., one task for Phase 1 and r tasks for Phase 2 (where r is the number of regions). Phase 1 task input is on local disk, and its output data is written locally. Phase 2 tasks read their input data from remote machines in parallel.

- Configuration 2: *Cloning Off, spread data.* Cloning is disabled. We create the tasks as before. All data (including initial input) is spread.

- Configuration 3: *Cloning On, local data.* Cloning is enabled. We create one task per bag as before, but the system can clone both Phase 1 and Phase 2. Data is placed as in Configuration 1.

- Configuration 4: *Cloning On, spread data.* Cloning is enabled, as in Configuration 3. All data (including initial input) is spread.

We run the ClickLog application on 8 machines in each of the above four configurations with 80 GB of input data (10 GB per machine). Figures 4.6 and 4.7 show the results for Phase 1 and Phase 2 respectively. Phase 1 is not impacted by skew since each IP is geolocated and placed in the corresponding region bag independently. We observe that spreading data in the bag is essential for good performance as local data places the burden of serving that data on a single storage node. For instance, with local data, turning cloning on only speeds up Phase 1 by 25% because, even though the output of clones is placed on local storage, one machine must still supply the entire input. Figure 4.7 shows that Phase 2 is severely impacted by skew, as shown in the first configuration. Spreading the data improves performance by 33% (second configuration) because it helps achieve better storage load balance, allowing the machine processing the heaviest region to use all disks when it is the last task remaining, effectively increasing its storage bandwidth. Cloning with local data (configuration 3) is slower than cloning with data that is spread (configuration 4) because clones are introduced with a delay, hence the output is not uniformly distributed across all storage nodes if it is kept local. Finally, we observe that cloning has the most impact with increasing skew since it allows the heaviest region to be simultaneously processed by multiple workers.

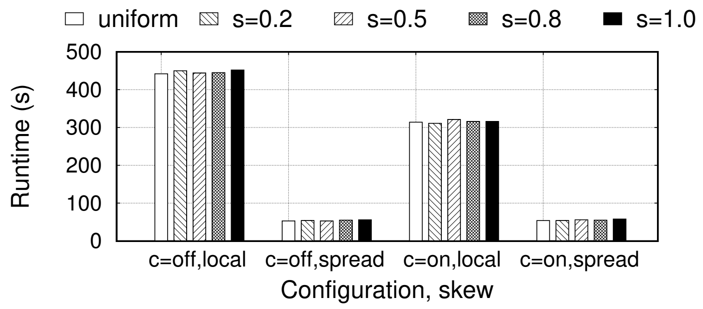


Figure 4.6 – Runtime of ClickLog Phase 1 for different configurations with various features turned off.

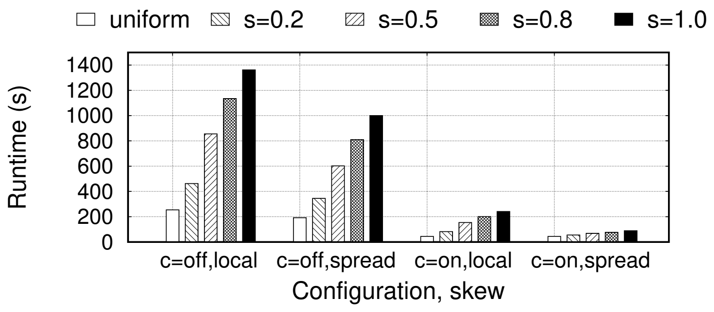


Figure 4.7 – Runtime of ClickLog Phase 2 for different configurations with various features turned off.

Locality One might wonder whether Hurricane takes a performance hit by spreading data uniformly in the absence of skew. As we can see from Figures 4.6 and 4.7, this is not the case in our deployment because the network is fast enough to match storage bandwidth. As a result, remote bandwidth is roughly the same as local bandwidth.

Overload Detection & Cloning Heuristic Hurricane clones tasks to rebalance load and increase parallelism for large tasks, allowing the system to utilize both CPU and storage resources better. We evaluate the effectiveness of our overload detection mechanism and cloning heuristic with ClickLog running on 32 machines with 320 GB input. We set the skew parameter to 1 (high skew).

Figure 4.8 shows the aggregate throughput achieved by all compute nodes in the system sampled at one-second intervals. Phase 1 starts with one worker executing the single task. Since the task is CPU bound, it clones rapidly until all 32 machines are running clones around the 15 second time point (the number of clones doubles approximately every 2 seconds). There is no merge in Phase 1, and all workers complete roughly at the same time.

Phase 2 then starts with one task per region, which together occupy all available worker slots at compute nodes. As tasks associated with small regions complete, their associated Phase 3 tasks are scheduled and executed. When they finish, some compute nodes become idle because there are no more available tasks, allowing compute nodes processing larger regions to get higher storage bandwidth. This overloads their CPU, so they issue cloning requests to the application master, which grants them on a case-by-case basis. Eventually, only the largest region remains, with 26 workers simultaneously processing it. Cloning stops beyond 26 workers because storage, and not the CPU, becomes the bottleneck. As this region gets close to the end, the application master rejects further cloning requests, as the merge overhead would become larger than the benefits of cloning. Once the region is processed, the outputs of each clone are combined by a merge task, and application execution terminates.

Throughput remains nearly constant for Phase 2 despite significant skew because the system clones tasks on idle compute nodes when storage is not the bottleneck.

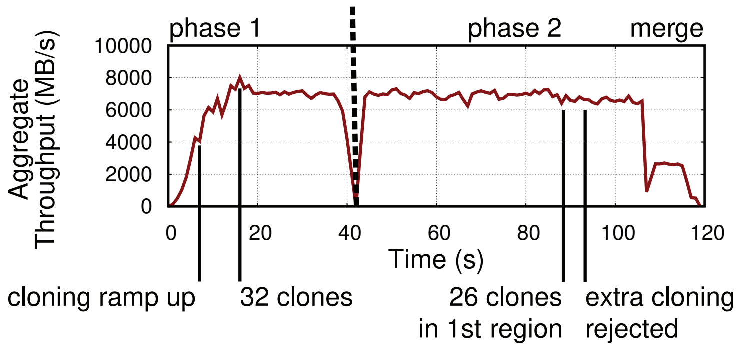


Figure 4.8 – ClickLog throughput over time on 32 machines. The vertical dashed line separates Phase 1 from Phase 2.

Batch Sampling We consider the batch sampling technique presented in Subsection 4.3.3 and Subsection 2.1.4 and evaluate its impact on performance. Batch sampling of chunks is a form of oversubscription performed by storage clients on behalf of workers is a means for ensuring that storage nodes remain busy throughout their execution and that workers are not starved for data, essentially overlapping computation and communication through prefetching of chunks.

We consider Phase 1 of ClickLog with various batch sampling factor values, from $\Phi K = 1$ (i.e., one chunk at a time) to $\Phi K = 32$ (i.e., one in-flight request per storage node). Figure 4.9 shows that allowing workers to prefetch multiple chunks is essential for good performance and to keep storage nodes busy. However, prefetching too many chunks ($\Phi K = 32$) is undesirable since it risks overwhelming storage nodes and could lead to unfairness. $\Phi K = 10$ is the sweet spot, allowing us to achieve 33% runtime improvements simply through better overlapping of computation with storage I/O.

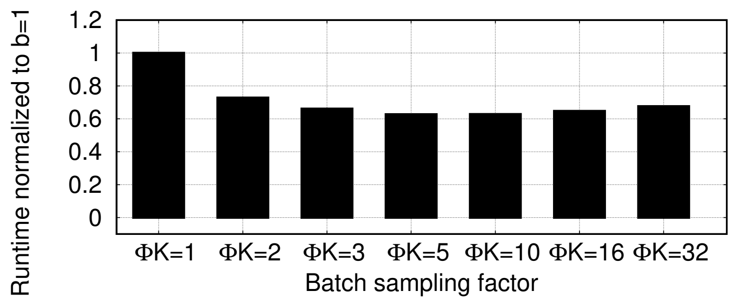


Figure 4.9 – Runtime of ClickLog Phase 1 on 32 machines for various batch sampling factors.

Throughput and Storage Utilization In the scatter architecture, storage nodes are designed to scale storage I/O throughput observed by compute nodes with an increasing number of storage nodes. We verify that this is the case by running a synthetic benchmark where each worker writes a fixed amount of random data (100 GB) and then reads the data back. We start on one machine and then double the number of machines until 32, thus doubling the aggregate amount of data stored in storage nodes. The results indicate that Hurricane sustains maximum I/O bandwidth, regardless of the number of machines involved. For instance, we achieve 330 MB/s read bandwidth with one machine and 10.53 GB/s read bandwidth with 32, an increase of 31.9× for 32× more machines. Similarly, we achieve 327 MB/s write bandwidth with one machine and 10.39 GB/s write bandwidth with 32, i.e., 31.7× speedup. By increasing the number of storage nodes, applications can scale throughput while maintaining high storage throughput.

Fault Tolerance We evaluate the impact of compute node and application master crashes on throughput. Figure 4.10 shows the aggregate throughput over time for an execution of ClickLog on a 320 GB input using 32 machines (10 GB per machine). We forcibly crash a compute node twice: once during phase 1 and once during phase 2. In each case, we also crash the application master 20 seconds after recovering from the compute node crash. Compute node crashes cause throughput to deteriorate temporarily, as the system must stop all corresponding task clones and restart the crashed task. Since phase 1 consists of a single task, the crash of the compute node requires restarting all workers in the system. In phase 2, the same crash only requires restarting all associated clones of the task (recall, different regions have different tasks), and thus the throughput only degrades by ~ 25%. Application master crashes have little impact on throughput for two reasons: the master's recovery is speedy (less than 1 second), and once tasks are placed in the work bag, compute nodes can proceed independently of the master's status.

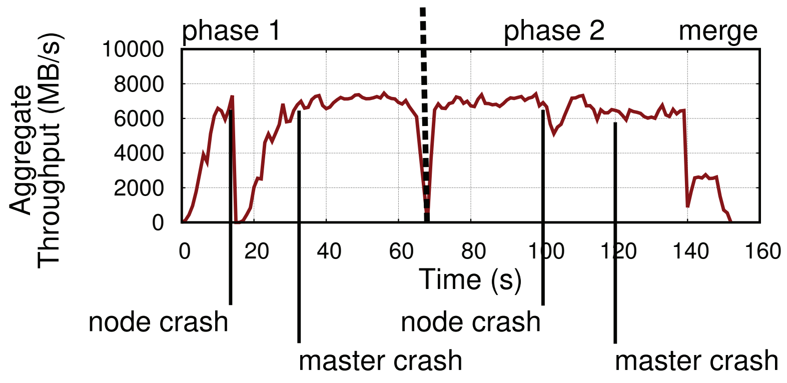


Figure 4.10 – ClickLog throughput over time on 32 machines with worker and application master crashes. The vertical dashed line separates Phase 1 from Phase 2.

4.5.3 Applications and Comparisons

Finally, we consider three workloads, representative of real-world applications, described below. We compare the performance of Hurricane on these workloads with optimized implementations in Hadoop and Spark.

- *ClickLog*: count the distinct number of occurrences of each IP address per region in a log of clicks on advertisements. This application was presented in Section 4.2.
- *HashJoin*: given two relations and an equality operator between values, for each distinct value of the join attribute, return the set of tuples in each relation that have that value. This is a classic problem in relational databases.
- *PageRank*: execute 5 iterations of the PageRank algorithm [134] on a large real-world power-law graph. PageRank has many real-world applications and is a well-known benchmark used in graph processing systems. This is an iterative multi-stage application.

ClickLog We compare our ClickLog results with Hadoop and Spark by evaluating each system's performance under different levels of input skew.

The implementation of ClickLog in Hadoop maps parts of the input text to the workers, which tokenize it, parse the IP addresses, geolocate the IP address per region, and output intermediate lists of IP addresses in each region. The reduce phase goes over the intermediate lists to perform a distinct count. Spark operates in much the same way on resilient distributed datasets, mapping the input to workers, tokenizing, geolocating by region, and counting distinct IP addresses. Wherever possible, we use the same data structures and perform the same operations for all implementations. In particular, all implementations use bitsets to perform the distinct count.

We use HDFS [148] in the case of Hadoop and Spark. We make sure that both Hadoop and Spark read their input data from the local disk and write the much smaller output without replication. We also split the job into enough tasks to ensure that Hadoop and Spark can utilize all available cores in the cluster and have enough opportunities to balance the load. We try multiple values for the number of partitions (ranging from 100 to 1'0000) and report the best runtime across all configurations. We verify that no task was restarted because of a crash during the execution.

Table 4.2 shows the runtime of all three systems on uniform inputs for two input dataset sizes. The 320 MB input is guaranteed to fit in memory on a single machine even in the presence of high skew, while the 32 GB may not fit in a single machine due to Java runtime overheads. All three systems (in particular Hadoop) experience some overhead when executing on the small 320 MB input as a result of small task sizes.

Hurricane achieves lower overall runtimes because it does not need to sort intermediate data, which allows better overlap between computation and communication. Both Hadoop and Spark must sort intermediate data to ensure key ranges do not overlap. However, eliminating sorting in Hurricane does not come free, since the system must perform an additional merge task for cloned tasks.

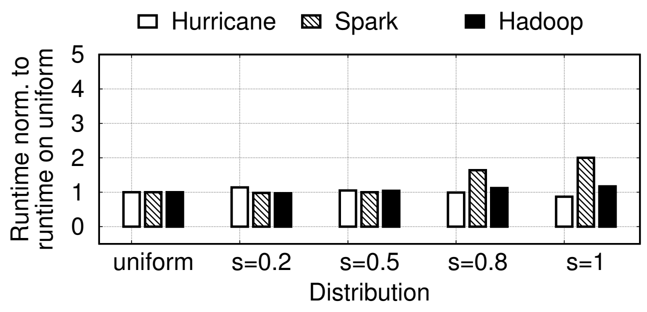
System	320 MB	32 GB
Spark	8.2s	32.4s
Hadoop	37.1s	50.3s
Hurricane	5.7s	22.8s

Table 4.2 – ClickLog runtime over a uniform input for 320 MB and 32 GB input sizes.

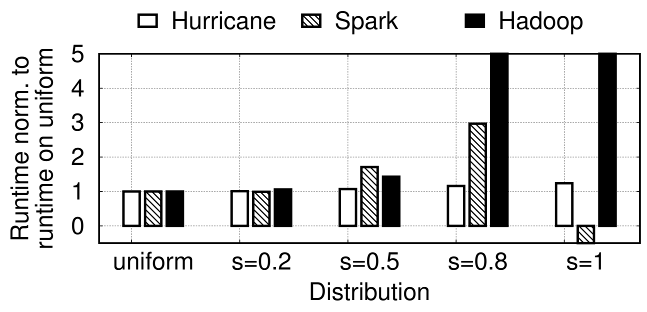
Figure 4.11 shows the slowdown on all three systems as skew is introduced in the input. To ensure a *fair* comparison, the runtime for each system is normalized to its *own* runtime with the uniform input.

We observe that both Hadoop and Spark suffer significant performance degradation in the presence of skew, particularly as the input size increases. Spark runs out of memory and crashes with highly skewed tasks due to a hard limitation of 16 GB placed on task memory. Hadoop suffers from a large increase in runtime due to the impact of skew on a few reducers, forcing them to spill.

HashJoin Table 4.3 shows the runtimes for two joins, one between a small 3.2 GB relation and a larger 32 GB relation, and the second between a 32 GB relation and a 320 GB relation,



(a) 320 MB



(b) 32 GB

Figure 4.11 – Comparison of Hurricane, Spark, and Hadoop when the skew is increased for input sizes 320 MB and 32 GB. A full bar indicates that the execution did not terminate in under an hour and was forcibly terminated. Negative bars indicate a crash.

for both Hurricane and Spark. For both joins, we introduce skew in the first (smaller) relation, causing a much larger hit rate for some keys. The join in Hurricane splits the smaller relation into 32 equal-sized partitions and sorts them in memory. It then creates 32 corresponding partitions in the larger relation, and finally streams the larger partitions, while the smaller partition is in memory, outputting matching keys. Spark's implementation proceeds similarly but with more partitions to make sure all available CPU cores are used. We try varying the number of partitions (ranging from 100 to 1'0000) and report the best overall runtime. As before, we ensure that input data is read from the local disk and that there are no task crashes during execution. We also disable output replication.

System	**3.2 GB ⋈ 32 GB**		**32 GB ⋈ 320 GB**	
	s=0	**s=1**	**s=0**	**s=1**
Hurricane	56s	89s	519s	1216s
Spark	81s	1615s	920s	>12h

Table 4.3 – HashJoin runtime for two different relation sizes and different amounts of skew. $s = 0$ is uniform.

We can observe that Spark struggles with the load imbalance caused by skew and that this effect worsens as the input size increases. The slowdown is directly caused by a larger hit rate in some partitions. Hurricane handles the situation more gracefully due to its ability to spread both input and output across storage nodes as well as its ability to clone the tasks containing keys with a larger hit rate.

PageRank Table 4.4 compares the runtime of PageRank in Hurricane and Spark's GraphX, a state-of-the-art graph-parallel library for graph applications. We compare different input sizes using 32 machines. We use the RMAT graph generator [62] to generate real-world power-law input graphs, i.e., graphs whose degree distribution is skewed. RMAT-24 has 16 million vertices and 256 million edges, RMAT-27 has 128 million vertices and 2 billion edges, while RMAT-30 has 1 billion vertices 16 billion edges.

PageRank is computed iteratively for 5 phases after an initialization phase. The PageRank implementation in Hurricane is based on the Chaos model described in Chapter 3. In each phase, each vertex in the graph sends its current PageRank along outgoing edges to neighboring vertices and then aggregates the PageRanks received by neighbors along incoming edges to compute its new PageRank. PageRank is essentially a *join* of vertex identifiers with outgoing edge source vertex identifiers to send out vertex values, followed by a *groupby* aggregation of received values on vertex identifiers. Because this is an iterative algorithm with changing input data, it is representative of long multi-phase application graphs. We use GraphX's example PageRank implementation for comparison, ensure the input is read locally, and check that no crashes occur during execution.

System	**RMAT-24**	**RMAT-27**	**RMAT-30**
Hurricane	38s	225s	688s
GraphX (Spark)	189s	3007s	> 12h

Table 4.4 – Comparison of Hurricane and GraphX on 5 iterations of PageRank over an RMAT-27, RMAT-30, and RMAT-32 graph.

Hurricane performs much better than GraphX on all input sizes. We observe significant task cloning in Hurricane throughout the execution, particularly for partitions with high-degree vertices, which allows each stage of the computation to finish in a timely fashion. GraphX struggles to finish executing on larger input sizes due to spilling and shuffling overhead. These results demonstrate that Hurricane handles skew effectively in multi-stage applications.

4.6 Summary

Hurricane is a system for high-throughput analytics designed from the ground up for handling skewed workloads. Hurricane works well because it is designed to dynamically partition work based on load imbalance. It allows programmers to seamlessly write applications whose performance does not degrade significantly in the presence of skew. Applications using Hurricane benefit from high capacity and scalability, as well as inherent load balance and high parallelism.

We provide a summary of Hurricane's features, as well as the specific scatter architecture techniques used by the framework below.

Area	General-purpose analytics
Load imbalance	Skew in input and intermediate data Processing skew due to, e.g., filtering Machine skew due to, e.g., interference
Data spreading	All data as bags Sorted data as files Tasks in work bags
Dynamic parallelism	Dynamic work sharing based on cloning + merging
Partitioning	Simple, user-defined
Misc. characteristics	Decentralized task scheduling Opportunistic load balancing

5 Load Balanced LSM-Based Distributed Databases with Hailstorm[1]

In this chapter, we depart the world of big data analytics for that of distributed databases. We present Hailstorm, a filesystem substrate inspired by the scatter architecture that improves load balance for large-scale distributed databases based on Log-Structured Merge-tree storage engines.

5.1 Introduction

Distributed databases such as MongoDB [19], Couchbase Server [59], or Apache Cassandra [21] have become the new standard for data storage in cloud applications. Internet companies use them to power large-scale services such as search engines [63, 69], social networks [21, 26, 44, 47, 57, 154], online shopping [19, 75], media services [22], messaging [52], financial services [23, 61], graph analytics [27], and blockchain [28, 86, 121]. As distributed databases become the *de facto* storage systems for distributed applications, ensuring their fast and reliable operation becomes critically important.

Distributed databases shard data across multiple machines and manage the data on each machine using embedded storage engines such as RocksDB [57], a Log-Structured Merge-tree [133] (LSM) key-value (KV) store. These databases can suffer from unpredictable performance and low utilization for two reasons. First, skew occurs naturally in many workloads and causes CPU and I/O imbalance, which degrades overall throughput and response time [96, 113, 120, 158]. Current LSM-based databases address skew by resharding data across machines [19, 20, 21, 24, 25]. However, this operation is expensive because it involves bulk migration of data, which affects foreground operations. Second, background operations such as flushing and compaction can cause significant I/O and CPU bursts, leading to severe latency spikes, especially for queries spanning multiple machines such as range

[1]This chapter is based on the following publication: Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 301–316. ACM, 2020. Hailstorm was designed, implemented, and evaluated solely by the author of this dissertation. The author maintains the software to this day.

queries [38, 41, 69, 78, 110]. These problems are hard to address in existing systems because the storage engines operate independently of each other and thus are unaware of resource usage and background operations on other machines. As a result, these databases experience significant imbalance in terms of CPU and I/O load, and storage capacity.

This paper presents Hailstorm, a lightweight distributed filesystem specifically designed to improve load balance and utilization of LSM-based distributed databases. This filesystem is inspired by the scatter architecture (see Chapter 2). Figure 5.1 shows the high-level architecture of a generic distributed database running with Hailstorm. Hailstorm is deployed under the storage engines running on each machine.

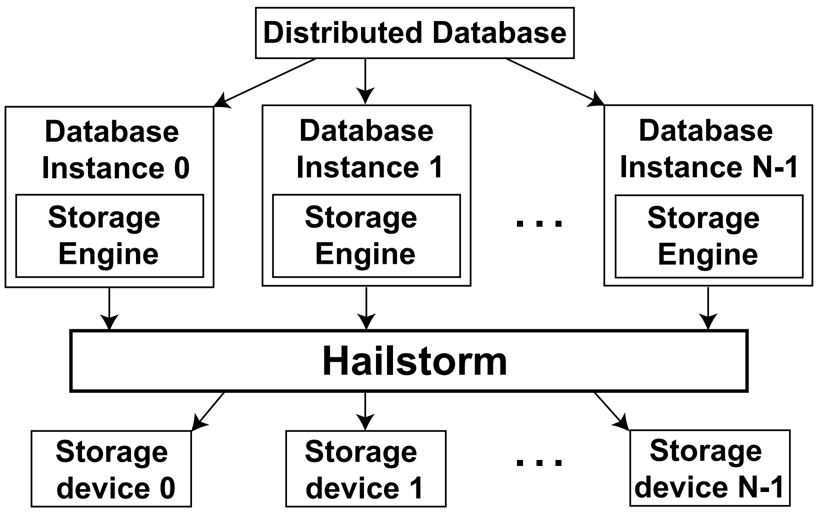


Figure 5.1 – High-level architecture of a distributed database with Hailstorm. Storage engines access storage devices through the Hailstorm filesystem which pools all storage devices within a rack.

The key idea in Hailstorm is to *disaggregate* compute and storage, allowing each to be load balanced and scaled independently, thus improving overall resource utilization.

Hailstorm scales and balances storage using the scatter storage techniques presented in Section 2.1. We pooling storage within a rack at a fine granularity so that each database storage engine can seamlessly access the aggregate rack storage bandwidth. The data for each database shard is spread uniformly across *all* the storage devices in a rack in small blocks (1 MB). This approach effectively provides a second, storage-level sharding layer that guarantees high storage utilization even in the presence of skew, removes per-machine disk space constraints, and eliminates the need for database-level resharding within the rack.

Hailstorm scales computation by offloading expensive background compaction tasks to other less utilized machines, leveraging uniform, fine-grained storage pooling (§2.2.3). Our approach reduces the CPU impact of compactions on overloaded machines, frees up CPU cycles for user requests, and lowers the memory footprint, thereby improving throughput and query response time.

We evaluate the performance of Hailstorm with MongoDB [19], a widely-used distributed database with a key-value store interface, running over Mongo-Rocks [24], an adapter for the popular RocksDB [57] storage engine. For our benchmarks, we use the reference Yahoo Cloud Serving Benchmark (YCSB) workloads [68] as well as two production workloads from Nutanix. We also experiment with TiDB [25], a state-of-the-art distributed database that supports SQL ACID transactions and bundles RocksDB as its storage engine. With TiDB, we evaluate the benefits of Hailstorm on industry-standard TPC-C [138] and TPC-E [65] benchmarks.

Hailstorm provides throughput improvements for skewed YCSB workloads running on MongoDB of 60% on average and up to 22× for scan workloads. It also reduces tail latency by 4-5× in skewed write workloads. On the production traces with skew, Hailstorm achieves 3× higher and stable throughput. With TiDB, Hailstorm improves throughput by 56% on TPC-C and 47% on TPC-E.

We make the following contributions in this work:

- We present Hailstorm, a system that disaggregates storage and compute for distributed LSM-based databases (§5.3).
- We demonstrate how the scatter-inspired Hailstorm filesystem for LSM storage engines (§5.3.2) uses pooled storage and fine-grained spreading of data across machines to scale storage within a rack (§5.3.3).
- We leverage our filesystem design to scale CPU resources within a shard by seamlessly offloading expensive background tasks to less utilized machines (§5.3.4).
- We show that Hailstorm's approach is the proper way to mitigate skew in various workloads and databases and that the system successfully achieves load balance for both storage and compute (§5.5).

5.2 Background & Challenges

We discuss the issues and challenges involved in dealing with skew and I/O bursts in distributed databases and LSM stores, which we support with empirical evidence.

5.2.1 Skew in Distributed Databases

Distributed databases [19, 20, 21, 22, 23, 44, 63, 69, 75, 154] store data on many machines, and are designed for large-scale data storage and low-latency data access. Although different distributed databases offer query abilities ranging from simple key-value semantics to SQL transactions, they all require *sharding*, i.e., partitioning data across multiple database instances, in order to store large datasets. Data is usually partitioned by collection or table using range-based or hash-based key partitioning.

The database engine translates user queries into individual queries that are routed to one or multiple database instances for execution. The set of database instances accessed by a query can vary significantly from one query to another and depends on the sharding policy. For example, while many reads and writes typically access a single instance, range queries and transactions may involve many instances.

Skew occurs naturally in many distributed workloads [96, 113, 120, 158], because some keys are more popular than others. As a result, sharding inevitably leads to data imbalance across the machines that make up a distributed database. This uneven key distribution can cause capacity problems if a machine has too much data to store locally. More importantly, uneven key distribution results in uneven accesses that cause load imbalance as some machines perform more operations than others.

5.2.2 Compaction in LSM KV Stores

Log-Structured Merge-tree (LSM) KV stores [29, 57, 86, 139] are a popular way to provide persistent storage in single-machine production environments, especially for write-heavy workloads. They provide key-value semantics using an in-memory buffer that is periodically *flushed* to disk when it becomes full. In case of failure, LSM KV stores recover the data in the in-memory buffer using a write-ahead log.

LSM KV stores organize data on disk in files sorted by key, called Sorted String Tables (*sstables*). The sstables are maintained in a tree-like data structure, with higher levels of the tree containing larger sstables. The key ranges of different sstables in a given level L_i do not overlap, except for the first level, L_0 that corresponds to flushed in-memory buffers. LSM KV stores preserve the tree structure using background operations called *compactions* that merge sstables in L_i with overlapping sstables in L_{i+1}, while discarding duplicates.

Compactions run in separate background threads as they are typically expensive operations in terms of both CPU and I/O. Executing a compaction at L_i requires reading all overlapping sstables in L_{i+1} to perform an external merge sort before writing back the new sstables. Since sstables are larger as we go to higher levels, this results in high write amplification, where a single, small sstable causes multiple larger sstables in the higher level to be read and written. The larger the sstables involved are, the longer the compaction and the more resources are required. For example, if the highest LSM level contains hundreds of gigabytes, a single compaction involving that level can take minutes to hours.

Figure 5.2 shows the throughput fluctuation over a one-hour time period for RocksDB [57], running YCSB A, a workload consisting of 50% reads and 50% writes. The throughput generally remains between 18 KOps/s and 32 KOps/s with an average of 22.4 KOps/s, but repeatedly drops to ~2.7 KOps/s. The performance degradation is due to compaction threads competing for CPU and I/O resources with the threads servicing client requests. Profiling this particular experiment reveals that storage is saturated as I/O bandwidth remains almost constantly close to 320 MB/s, the maximum write bandwidth for our SSD. We also observe peaks of CPU utilization when compaction tasks run. As more data is stored, compactions become more expensive, throughput drops to as low as ~0.6 KOps/s around 01:00, and compaction duration quadruples to 9 seconds.

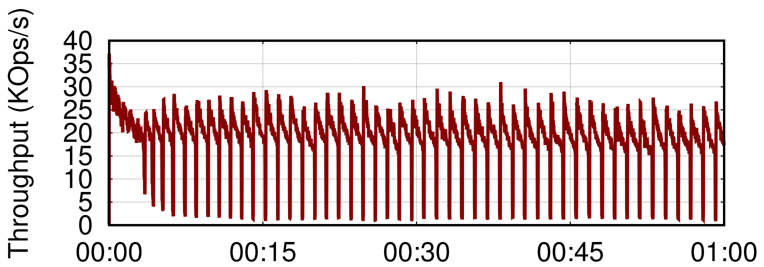


Figure 5.2 – Embedded RocksDB throughput over time (HH:mm) on a machine equipped with an Intel S3500 Series SSD using the YCSB A [68] workload (50% reads and 50% writes).

Table 5.1 shows the average and tail latencies in the same experiment. Latency spikes coincide with the execution of compaction tasks. Compaction tasks not only limit the available CPU and I/O bandwidth for read operations and writes to the write-ahead log, but also slow down flushing of the in-memory buffer when it becomes full, preventing the system from accepting more writes. Upon profiling, we find that writer threads are stalled 48.4% of the time due to flushing.

Mean	**P50**	**P99**	**P99.9**	**P99.99**	**Max**
1.6ms	0.7ms	35.1ms	72.1ms	181.3ms	69.5s

Table 5.1 – RocksDB latency profile. YCSB A, 50% reads - 50% writes.

Several solutions have been proposed to reduce the impact of background operations in LSMs on individual machines. However, they do not take advantage of spare resources and capacity on other machines [29, 50, 139].

5.2.3 Distributed Databases with LSM Storage Engines

Many distributed databases rely on embedded LSM KV stores for local storage on each of their database instances in order to benefit from their high performance [20, 21, 24, 25, 154].

This two-layer architecture can, however, lead to situations where both layers interfere with each other, combining the undesired effects of skew (§5.2.1) and expensive background operations (§5.2.2) and causing severe performance degradation for database users. Skew causes some machines to experience higher load from user requests, causing an overload, and background operations become more intensive since these machines manage more data than others. If the requests have dependencies or high fan-out, these overloaded machines become stragglers, and performance across the whole database collapses.

MongoDB, like many other distributed databases, addresses skew by resharding, i.e., sharding again, to remove hotspots and improve load balance. Resharding operations, performed manually or automatically at the database layer, include, e.g., adding a new shard, splitting one shard into multiple shards, or merging multiple shards into one. Resharding involves migrating existing data from one shard to another, often located on a different database instance. Resharding is expensive because it introduces background operations that compete for resources with regular operations. Unlike resharding in B-tree-based databases, which is usually performed by splitting B-tree nodes [39, 149], resharding in LSM-based databases is more complicated because data is stored in files with overlapping key ranges. As database instances strive to maintain their LSM data structure in the presence of additions and deletions due to migrations, additional flushing and compaction tasks are required. These compactions cause significant amplification of I/O and CPU usage. Furthermore, resharding decisions are usually taken by the distributed database based on its own load metrics, without regard to I/O load on individual instances and their current background operations.

Figure 5.3 compares the throughput fluctuation over time for 8 MongoDB instances using RocksDB for storage. We run YCSB A, a write-intensive workload, YCSB C, a read-only workload, both with single-key queries, as well as YCSB E, a read-write workload with range queries, with both uniform and skewed (Zipfian) distributions.

In all three cases, the throughput is degraded with Zipfian request distribution, whereas the uniform throughput is higher and more constant. Skewed workloads, therefore, take longer to finish: YCSB A, C, and E respectively run 3.3×, 1.4×, and 8.5× slower due to skew.

In YCSB A with Zipfian distribution, the per-instance throughput is much higher on one machine than it is on the others. That machine serves ~75% of requests and experiences high CPU usage and I/O load. This highlights the problem when skew in the workload and background operations are combined. During the entire execution, MongoDB attempts to recover from this hotspot by rebalancing shards and redistributing a total of 25.4 GB of data across other database instances, thus reducing data skew from 9× to 5×. However, this data migration causes more expensive flushes and compactions as the offloaded keys are deleted from one instance and written to another, and results in additional performance degradation and longer throughput drops, e.g., at 00:37.

In YCSB C, the per-instance throughput in the skewed scenario also suffers from significant imbalance, with one machine serving almost 3× more requests than the others. MongoDB's shard rebalancer constantly runs during this workload and migrates data from the machine experiencing high load to the others, leading to an increase in throughput from ~140 KOps/s to ~150 KOps/s in 15 minutes. We observe flushes and compaction tasks on each database instance as a result of data migration, causing occasional sharp drops in throughput, e.g., at 00:04. These results demonstrate that even read-only workloads can suffer throughput degradation and exhibit similar characteristics as write-heavy workloads due to resharding.

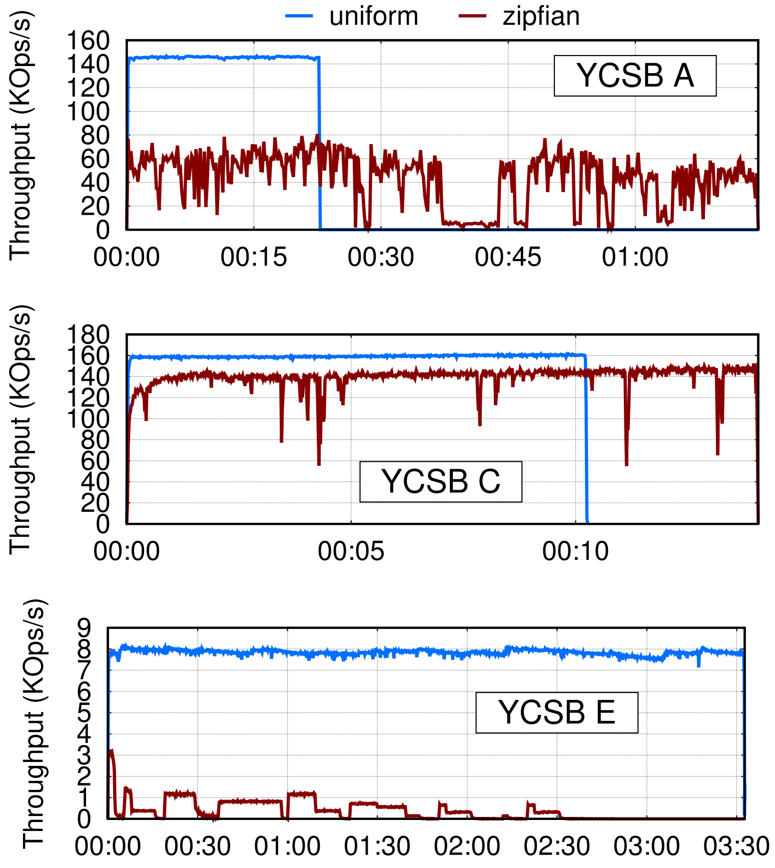


Figure 5.3 – MongoDB aggregate throughput on 8 instances over time (HH:mm) for the YCSB workloads A, C, and E with uniform and Zipfian key distribution. The database is first populated with 100 GB of data before executing each workload with an additional 100 GB. Each MongoDB instance is configured to use the LSM-based RocksDB [57] storage engine.

In YCSB E, the throughput in the Zipfian case is significantly degraded. It frequently drops close to 0 as range queries are stalled by compactions competing for I/O bandwidth and CPU resources on overloaded machines. We profile system resource usage and find that the most overloaded machine oscillates between 100% CPU and 100% I/O usage for 2 hours. While the uniform workload completes after 3.5 hours, the skewed workload completes in 30 hours, with 1/8th the throughput and 99-percentile latency 5 orders of magnitude higher. After 2.5 hours of execution, the combination of resharding overhead and increasingly expensive compactions cause near-zero throughput for over an hour.

Overall, the MongoDB shard rebalancer is unable to address imbalance in the face of skew and high request rate. Resharding often comes too late and is too slow to be useful. Besides, data migrations trigger additional background operations on database instances that impact the foreground tasks and make the overload worse.

5.2.4 Summary

We have shown that skew has a significant impact on application performance, and storage engines suffer from I/O bursts due to background operations such as flushing and compaction. When skew and I/O bursts are combined, performance can collapse. Also, resharding rarely improves performance, especially when run during peak loads and in the presence of hotspots. By rebalancing shards, distributed databases attempt to solve three orthogonal problems: CPU, I/O load, and I/O capacity imbalance. These challenges motivate a more synergistic approach, taken in Hailstorm, where we disaggregate resources to address load balance at the database and the storage layers independently.

5.3 The Hailstorm Design

Figure 5.4 expands on Figure 5.1 to show the detailed system architecture corresponding to a typical deployment on top of Hailstorm. The Hailstorm architecture is based on the scatter architecture described in Chapter 2 and consists of a set of compute and storage nodes. Each compute node runs a database instance with an embedded storage engine alongside the high-performance, distributed Hailstorm filesystem, consisting of a storage client that provides a filesystem interface to storage engines, and a Hailstorm agent that schedules and outsources compaction tasks on behalf of the local storage engine. Database instances use the Hailstorm filesystem instead of local storage for data persistence. Hailstorm pools all storage servers within the same rack together to provide its storage service. Each storage node runs a storage server that allows storage clients to store and access data. Compute and storage nodes are provisioned independently and can run on separate, possibly dedicated machines. In the rest of this paper, we make the assumption that compute and storage nodes are co-located.

The distributed database operates the same way as in traditional deployments as it is oblivious to the fact that storage engines are using Hailstorm. User queries are served as usual, but individual storage engines now perform storage operations across the network using the Hailstorm pooled storage service.

In the remainder of this section, we present an overview of the Hailstorm design. We first discuss and motivate the filesystem approach. We then describe the storage architecture, including some optimizations and handling of fault tolerance. Finally, we demonstrate how our approach supports efficient compaction offloading.

5.3.1 Hailstorm Design Principles

Hailstorm derives its design principles from the scatter architecture. The system disaggregates compute and storage resources in order to scale and load balance each resource independently. Hailstorm pools storage devices within a rack by introducing a filesystem layer below LSM storage engines, and spreads all data uniformly across all storage devices, disregarding locality

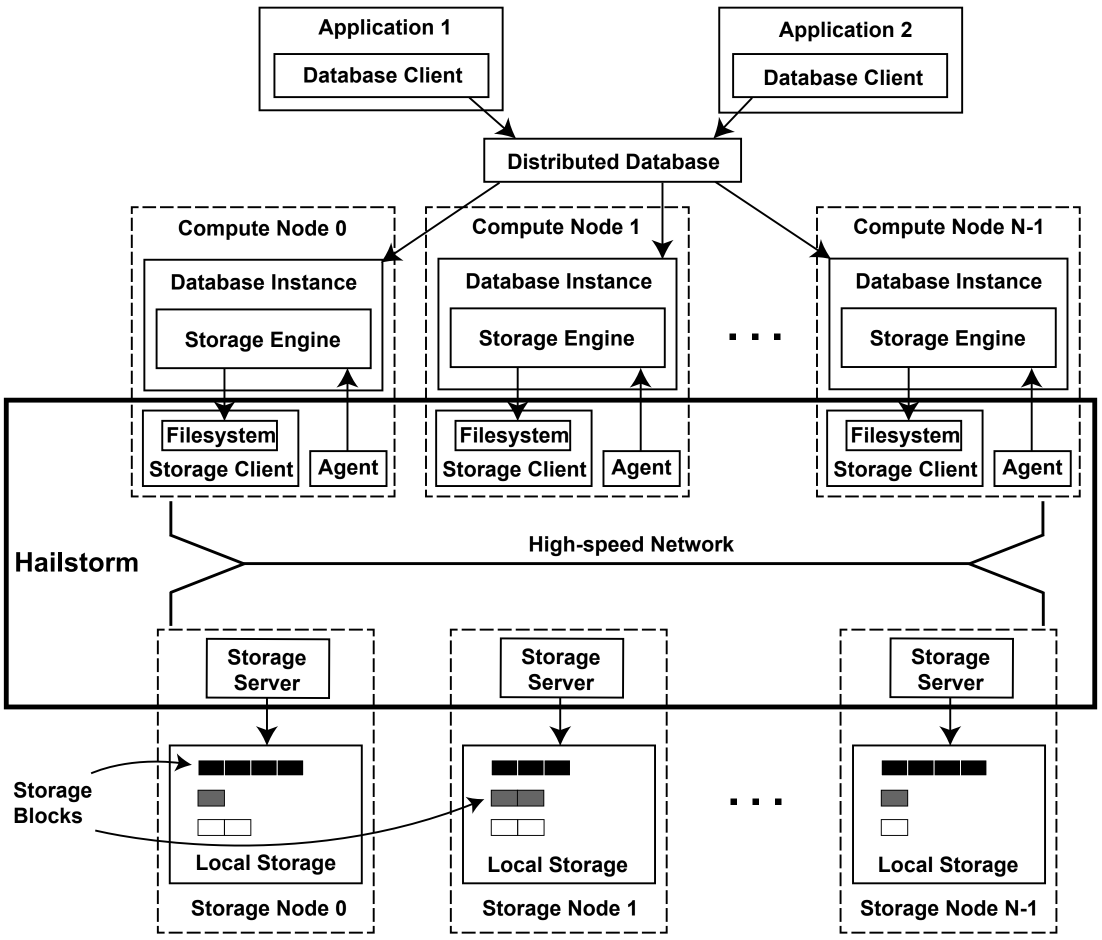


Figure 5.4 – Distributed database deployed on top of Hailstorm. Hailstorm spreads data uniformly for each storage engine across all pooled storage devices within the rack.

(§2.1.2 and §2.1.3). This approach allows the system to guarantee uniform data placement and to mitigate storage hotspots by spreading the I/O load. Finally, Hailstorm pools computation resources within a rack together by offloading the background compaction tasks necessary to maintain the LSM structure to other machines with spare CPU and memory (§2.2.3). In so doing, the system provides relief to machines with high load whose resources are significantly taxed by compactions.

5.3.2 Filesystem Architecture

Hailstorm exposes a subset of the standard POSIX filesystem interface as required by storage engines. This filesystem interface serves as a drop-in replacement for the local filesystem used by storage engines for data persistence. The Hailstorm filesystem uses a client-server architecture, where each client can access and store data on all servers within the same rack, thereby allowing the shards of one storage engine to span multiple storage devices (§2.1.2).

Why a Filesystem? We choose to expose a filesystem interface, instead of providing a block-level interface, because it provides the desired visibility into the operations of storage engines. In particular, we require knowledge of the sstable files used by the LSM store to perform compaction offloading. Also, the filesystem interface provides support for operations such as `mmap()` that are commonly used by storage engines. File-level visibility also allows us to perform more informed prefetching. Another significant benefit of using a standard POSIX file interface is that it requires minimal modifications to the storage engine code.

What About Using Existing Filesystems? Unlike existing distributed filesystems [30, 72, 87, 148, 156], Hailstorm is specialized for LSM KV stores. This specialization obviates the need for Hailstorm to implement many complex features found in traditional distributed and cluster filesystems. Since sstable files are not modified in place and only shared across storage engines for compaction offloading, Hailstorm does not require any support for fine-grained file sharing. Therefore, Hailstorm keeps most file metadata locally, avoiding the need for centralized metadata management. As indicated in Subsection 2.1.3, we use smaller block sizes (e.g., 1 MB) than most distributed filesystems to keep I/O latency low. Since LSM KV stores already use journaling, Hailstorm does not need to implement journaling to ensure filesystem consistency.

Hailstorm can leverage its specialization to optimize for efficient data access, in particular fast sequential operations on sstables, which is necessary for good compaction performance. We perform aggressive prefetching on behalf of the compaction tasks (§2.1.4) and provide remote, in-memory caching for large data sets using the page cache-backed nature of our implementation. Optionally, we allow write-ahead logs to remain on fast, local storage, facilitating failure recovery.

5.3.3 Storage Architecture

Hailstorm uses the scatter storage architecture described in Section 2.1 to scale storage within a rack. Hailstorm pools storage from all machines within a rack using a client-server filesystem approach.

Pooling Storage Each storage client exposes a filesystem interface to its co-located LSM storage engine and stores the data at block granularity on a pool of all storage devices (§2.1.2). This makes it possible to absorb storage load in the presence of peaks on database instances by spreading I/O operations to all storage servers within the rack. LSM storage engines running on Hailstorm can, therefore, sustain reads and writes during compactions, and avoid flush stalls [51]. It also enables efficient compaction offloading since the sstables for a particular storage engine can be accessed from any client, and thus, storage does not become a bottleneck. Pooling storage also enables small and large shards to cohabit within a rack without requiring additional provisioning or expensive rebalancing.

Data Placement and Access Hailstorm only uses files, not bags. Clients store data in fine-grained blocks uniformly across all storage servers within the rack (§2.1.3). Clients access data blocks as requested by the filesystem layer (§2.1.4).

Storage Load Balance Hailstorm ensures high storage utilization and balanced using over-subscription (§2.1.4). However, unlike in previous applications, Hailstorm clients cannot use prefetching for standard foreground requests since the access pattern is determined by user queries and is therefore random. As a result, clients may not always be able to maintain a fixed number of outstanding requests to storage servers. In practice, we find that this is rarely a problem because read requests generally involved multiple I/O operations, and writes are absorbed into the in-memory buffer before being written out sequentially. Also, the I/O pattern during compactions is always sequential, and therefore supports prefetching of next blocks by clients.

Read Optimizations LSM KV stores must often access multiple sstables to find the value for a key. If reads across the network were to use the same block granularity as writes, this may cause long delays. Hailstorm optimizes for this scenario by having reads from foreground threads execute at smaller block granularity, thereby reducing block access latency. Flushing and compaction use the default block granularity to maximize I/O performance. In order to guarantee high storage utilization with smaller granularity requests, our batch sampling technique uses a larger amplification factor Φ value for reads (§2.1.4).

Asynchronous I/O Hailstorm performs most I/O operations asynchronously with the exception of `fsync()`. Storage engines rely on `fsync()` to guarantee that storage is in sync with the in-core state, so we use a blocking implementation to ensure correct `fsync()` semantics.

Fault Tolerance Distributed databases rely on replication to provide fault tolerance. They replicate data across replica sets that are located on different machines, different racks, different availability zones within a datacenter, and possibly across datacenters. LSM storage engines ensure durability by using a write-ahead log (WAL).

In the event of a crash, Hailstorm primarily relies upon the failure recovery mechanisms implemented by LSM storage engines and distributed databases. Replication is a concern for the distributed database layer and is better implemented there than at the filesystem level where there is insufficient visibility into the entire database. Hailstorm allows databases to perform replication transparently but requires that replicas be placed in different racks, so they do not all become unavailable at the same time due to failures in a storage pool.

When deploying distributed databases on top of Hailstorm, a single disk failure or machine crash may cause all shards within the rack to become unavailable since data for each shard is spread uniformly. Hailstorm mitigates single-disk failures using standard techniques to ensure redundancy, e.g., RAID [136]. Also, the system supports optional primary-backup replication at the block level to further protect data durability and filesystem availability. File metadata is persisted locally and replicated. All other state in Hailstorm is soft state and can be lost without affecting correctness.

5.3.4 Compaction Offloading

Hailstorm uses the scatter compute architecture described in Section 2.2 to increase parallelism as needed for heavy database shards. Specifically, we rely on offloading of background tasks (§2.2.3), specifically compactions, to alleviate CPU load on overloaded machines. In this work, we do not explore dynamic work sharing techniques (§2.2.2).

Compaction Mechanism Hailstorm runs a lightweight agent on compute nodes alongside each storage client and database instance to monitor resource usage. Agents intercept all automatically triggered compaction jobs on their co-located LSM storage engine. If the agent believes that the local machine is overloaded, it pauses the compaction threads and attempts to offload the compaction to another compute node in the rack with a lower load. Otherwise, it allows the compaction job to proceed locally. If the agent decides to offload the compaction job, it extracts the relevant parameters (e.g., which sstable files should be compacted), and contacts a peer on another compute node to run the compaction on its behalf. The agent informs its peer of the details of the compaction job and transfers the associated file metadata. The remote agent spawns a new LSM storage engine process on its compute node with the sole purpose of running a manual compaction job equivalent to the one that was offloaded. Since compaction does not modify the files in place, no additional synchronization between the two agents is necessary. When compaction completes, the remote agent notifies the agent on the original compute node with the list of newly created sstable files and their associated file metadata and wakes up the paused compaction threads. This allows the original LSM storage engine to take ownership of the new sstables and install the compaction locally.

Overload Detection If a database instance is already experiencing significant load, the additional execution of background tasks using significant resources such as compaction can lead to request queuing and stall flushing of the in-memory buffer, thereby causing longer tail latencies and degradation in throughput. Since our design pools secondary storage and the network is fast at the rack level (as is the case in many deployments), compaction tasks primarily lead to CPU or memory bottlenecks in Hailstorm.

Compaction Policy Database operators can implement various compaction policies based on their service-level objectives (SLOs), such as running dedicated compaction machines.

By default, Hailstorm uses a simple heuristic to determine whether to try and offload a local compaction task. Each Hailstorm agent measures CPU utilization periodically and maintains an exponential moving average (EMA) with weight α that is shared with other agents on the same rack. Whenever an agent intercepts a local compaction task, it offloads compaction to the compute node with the lowest EMA value, provided that the difference between its EMA value and the target compute node's is larger than a customizable threshold θ. This scheme balances CPU load within a rack over time.

In practice, we find that values of $\alpha = 0.5$ (with 1-second CPU sampling period) and $\theta = 0.2$ work fairly well. Disabling compaction offloading can be achieved by setting $\theta \geq 1$.

5.4 Implementation

Hailstorm is implemented in about 1,000 lines of C++ code. We use FUSE [31] to provide a filesystem interface for storage engines and use about 2,000 lines of Scala code to implement distribution, client-server communication, and Hailstorm agents. We choose FUSE to simplify development, but alternative approaches such as Parallel NFS [32] are also possible. We interface Scala with our C++ FUSE module using the Java Abstracted Foreign Function Layer [33] for high-performance and low overhead. We use the Akka toolkit [13] for high-performance concurrency and distribution. For simplicity, we use the local ext4 [114] filesystem to store blocks on storage servers. We find that the overhead of using a filesystem on the server side is negligible with our block sizes.

Supported databases Hailstorm implements the scatter storage architecture described in Section 2.1 and provides a subset of the POSIX API for the filesystem it exposes through storage clients. As a result, Hailstorm does not require any modifications to storage engines or databases. We have successfully tested Hailstorm with RocksDB [57], as well as API-compatible variants of LevelDB [86], including PebblesDB [139] and HyperLevelDB [29].

Hailstorm has been tested for deployment under MongoDB [19] using MongoRocks [24] as its storage engine, as well as TiDB [25], whose KV store, TiKV [25], uses an embedded RocksDB engine. It should, in principle, be possible to deploy Hailstorm in any distributed database environment which uses compatible LSM-based storage.

Compaction Offloading For compaction offloading, we intercept compaction tasks and invoke Hailstorm agents in order to execute these compactions remotely, and therefore need to make small changes to RocksDB (~70 lines of code). Also, to perform remote compaction, we spawn a RocksDB process modified to remove some checks that would otherwise prevent compaction to run (6 lines of code commented out).

I/O Granularity and Batch Sampling Hailstorm uses a block size of 1 MB (§2.1.3). Not only does 1 MB provide a good balance between performance and remote access latency, but it also helps us minimize FUSE overhead by reducing the number of transitions to kernel mode. We pick a block size of 64 KB for client reads, which provides a good balance between latency and overhead from I/O accesses and FUSE. Each Hailstorm client concurrently has $\Phi K = 10$ pending requests for 1 MB blocks and $\Phi K = 100$ for 64 KB blocks (§5.3.3), which we have empirically determined to work best in our cluster environment.

5.5 Evaluation

5.5.1 Goals

We evaluate Hailstorm using synthetic and production workloads on popular storage engines and distributed databases. Our evaluation sets out to answer the following questions:

1. How do distributed databases perform when deployed on Hailstorm in terms of throughput and latency, especially in the presence of skew? (§5.5.3)
2. Does resharding help in traditional database deployments? How does it compare with Hailstorm? (§5.5.3)
3. Can databases supporting distributed SQL transactions also benefit from using Hailstorm? (§5.5.4)
4. What is the impact of different features of Hailstorm on performance and how does it compare with other distributed filesystems such as HDFS? (§5.5.5) Do configuration values affect performance? (§5.5.6) Can Hailstorm improve throughput for B-trees? (§5.5.7)

5.5.2 Experimental Environment

Hardware We run this evaluation on up to 18 dedicated 16-core machines (2 CPU sockets with Xeon E5-2630v3). Each machine has 128 GB of DDR3 ECC main memory and an SSD providing a read bandwidth of 420 MB/s and a write bandwidth of 320 MB/s, as reported by fio [8]. The machines are connected to a 40 GigE top-of-rack switch that provides full bisection bandwidth.

LSM KV stores We evaluate the performance of Hailstorm using RocksDB [57] (version 6.1), a popular LSM-based single-machine KV store.

Distributed databases We use two distributed databases with different designs and characteristics:

- MongoDB [19] (version 3.6), a popular and widely used database with a key-value store interface. MongoDB offers a powerful JSON-based document model and query language, many configuration options, and supports a multitude of storage engines. In this evaluation, we run MongoDB with Mongo-Rocks integration [24] (version 3.6), a RocksDB-based storage engine developed by Facebook and Percona.

- TiDB [25] (version 3.0), a distributed database supporting SQL ACID transactions built on top of TiKV [20], a scalable distributed KV store whose design is inspired by Google Spanner [69] and HBase [154]. TiKV instances embed RocksDB for storage. TiDB requires the use of separate placement drivers responsible for metadata, load balancing, and scheduling.

Deployment Unless otherwise specified, we always run 8 database instances with embedded storage engines (MongoDB shard servers or TiKV instances and placement drivers) on 8 dedicated machines. When using Hailstorm, we co-locate compute node and storage node on the same machines as the 8 database instances. Different deployments, such as using separate machines to run storage nodes are possible and supported.

When evaluating MongoDB, we deploy 2 configuration servers co-located with 2 routing nodes (mongos) on 2 additional machines. When running with Hailstorm, we turn off MongoDB's shard balancer for the entire experiment. We run 8 YCSB load generators on 8 separate, dedicated machines, which we have empirically determined to be sufficient to saturate MongoDB. For TiDB, we deploy 8 TiDB servers (responsible for receiving and processing requests) on 8 additional machines. We run the TPC-C and TPC-E load generators on a separate, dedicated machine, which we confirmed were sufficient to saturate TiDB. We do not enable data replication and clear the buffer cache as well as the database caches before each experiment.

System configuration We limit the total physical memory available to 32 GB of main memory on MongoDB shard servers and TiKV instances to ensure that all workloads are served from both main memory and secondary storage. We allocate up to 8 GB of main memory out of the available 32 GB for Hailstorm and use default block sizes.

We refer to deployments of RocksDB, MongoDB, or TiDB using local storage only as *Baseline* (BL). When deployments rely on Hailstorm to pool storage and distribute file data in blocks across all storage servers, but not on compaction offloading, we refer to them as *Storage Pooling* (HS-SP), and we call them *Hailstorm* (HS) when they rely on both.

5.5.3 Distributed Database: MongoDB

We evaluate Hailstorm in a distributed database setting with MongoDB [19] using synthetic and production workloads, and show how our approach benefits such deployments.

Workloads We use the Yahoo! Cloud Serving Benchmark (YCSB) [68] workloads as well as two production workloads from Nutanix. A summary of workloads used in this section is shown in Table 5.2, including their profiles (write:read:scan ratio) and the item sizes. YCSB provides 6 synthetic benchmarks covering a wide range of workload characteristics. For completeness, we consider an additional YCSB benchmark consisting of 100% inserts, which we refer to as *YCSB I*, and which corresponds to the load execution mode in YCSB. This write-only workload is added to provide a complete spectrum. To evaluate the impact of skew, we consider uniform and Zipfian key distributions for YCSB workloads. Zipfian key distributions are skewed, simulating the effect of popular keys. Nutanix's workloads are write-intensive workloads from production clusters. *Nutanix 1* is more uniform than *Nutanix 2*, which has some skew.

Workload	**Description**	**Profile (W:R:S)**	**Item size**
YCSB A	write-intensive	50:50:0	1 KB
YCSB B	read-intensive	5:95:0	1 KB
YCSB C	read-only	0:100:0	1 KB
YCSB D	read-latest	5:95:0	1 KB
YCSB E	scan-intensive	5:0:95	1 KB
YCSB F	read-modify-write	25:75:0	1 KB
YCSB I	write-only	100:0:0	1 KB
Nutanix 1	write-intensive	57:41:2	250B-1 KB
Nutanix 2	write-intensive	57:41:2	250B-1 KB

Table 5.2 – MongoDB workloads description and characteristics.

We first populate the database with 100 GB of data (100 million keys) from each workload before executing the workload with an additional 100 GB of data (100 million keys). We execute Nutanix's workloads with a pre-populated database containing 256 GB of data and execute each workload with an additional dataset size of 256 GB (approximately 700 million keys).

5.5.3.1 Synthetic Benchmark (YCSB)

Throughput Figure 5.5 compares the average throughput achieved by MongoDB for Baseline and Hailstorm for all YCSB workloads using uniform and Zipfian distributions.

Deploying MongoDB over Hailstorm allows the database to maintain good throughput even in the presence of high skew. Throughput is better with Hailstorm than with Baseline for write workloads (YCSB A, F, and I) thanks to storage pooling and compaction offloading. In particular, the throughput for YCSB A and I improves by ~2.2× and ~2.3×. Read-intensive workloads (YCSB B, C, and D) mostly take advantage of storage pooling, and their throughput improves by 46%, 15%, and 5%, respectively. Scan-intensive workloads (YCSB E) improve by over 22× with Hailstorm when there is skew in the workload. These benefits stem from offloading compactions, which lowers the load on the overloaded machine. Range queries almost always involve all MongoDB instances, and the presence of a single overloaded instance is sufficient to degrade performance dramatically. Range queries are commonplace in real deployments, and so Hailstorm will have significant benefits in these environments.

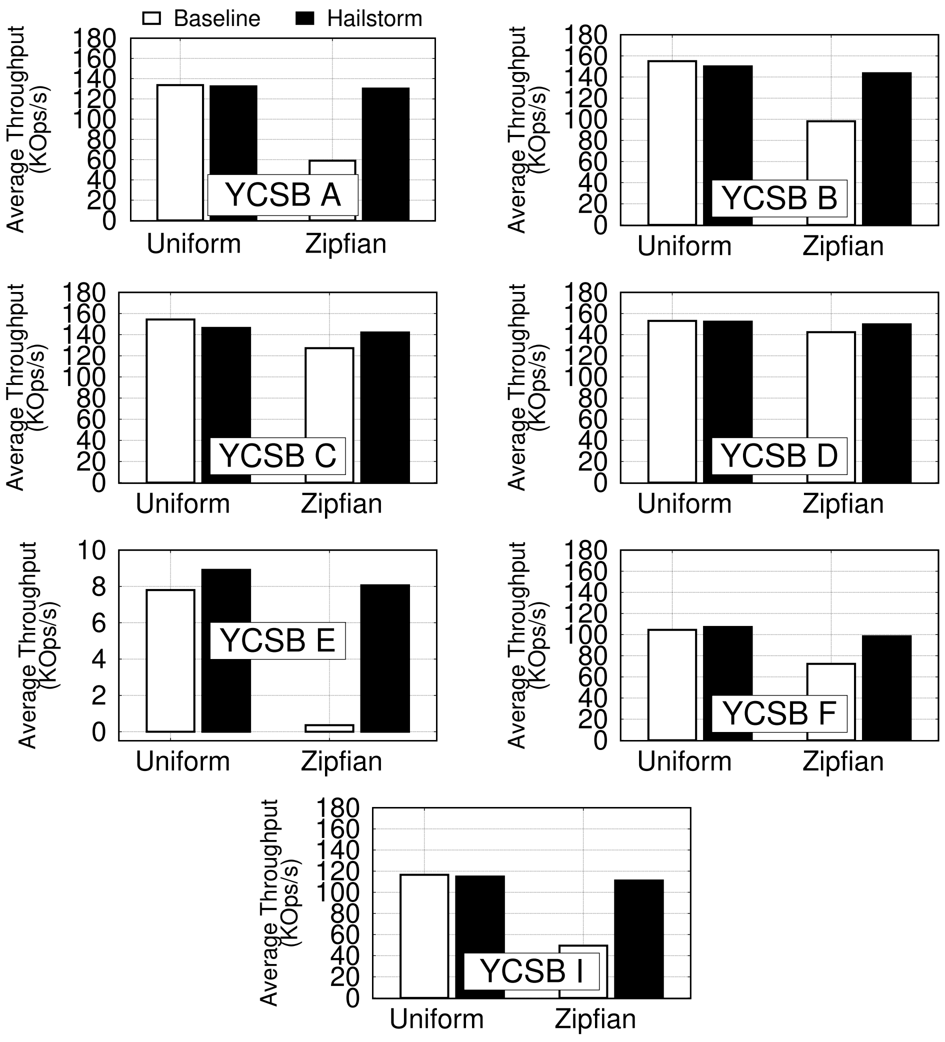


Figure 5.5 – MongoDB average throughput for Baseline and Hailstorm for YCSB workloads with uniform and Zipfian key distributions. Hailstorm maintains high throughput on all workloads.

Some workloads take a small throughput penalty in the uniform case when running on top of Hailstorm, due to FUSE and network overheads. However, these overheads are more than compensated if the workload has skew.

Throughput over time Figure 5.6 shows the throughput over time on all YCSB workloads with Zipfian key distribution for Baseline and Hailstorm.

These results demonstrate the ability of Hailstorm to maintain high and relatively constant throughput in the presence of skew. In particular, the throughput for write-intensive workloads (YCSB A and I) is lower for Baseline compared to Hailstorm as a result of skew, since one MongoDB instance is absorbing most of the load and can only do so using its local storage. Even with a smaller proportion of writes in YCSB B and F, the Baseline throughput is lower.

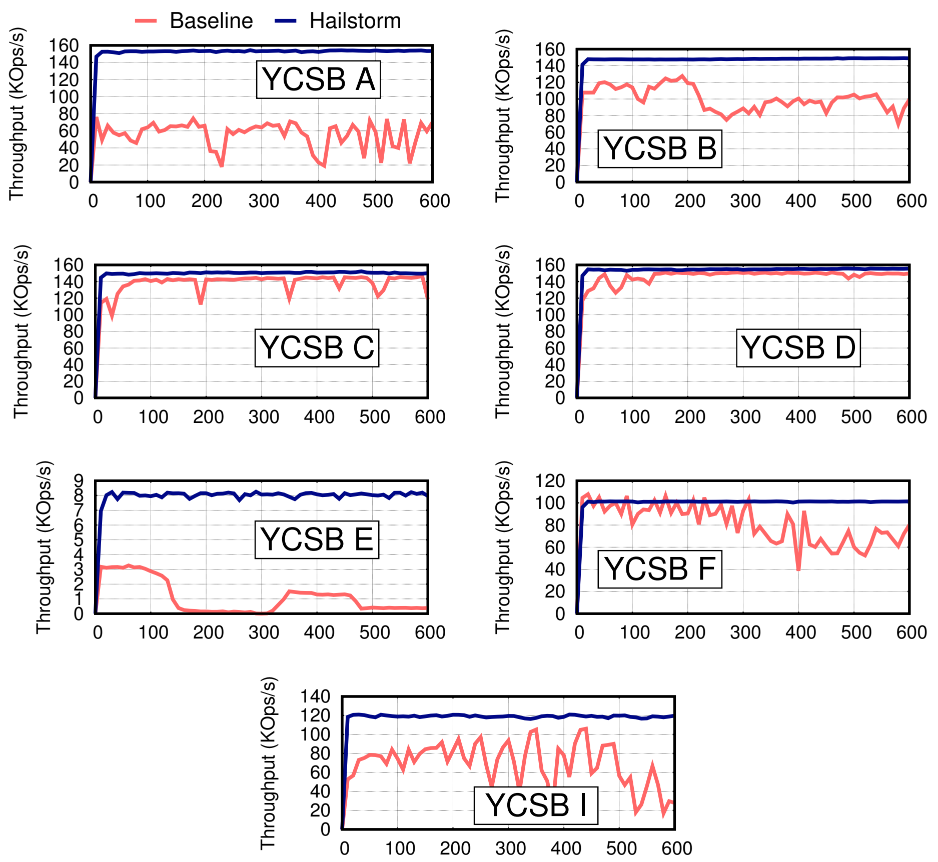


Figure 5.6 – MongoDB per second throughput timelines for Baseline and Hailstorm with YCSB workloads. Hailstorm manages to keep the throughput relatively constant throughout the execution.

Also, the throughput for the Baseline repeatedly falls by ~3× for YCSB A and ~4× for YCSB I, and to 0 in YCSB E as a result of a series of particularly costly compaction tasks. Throughput for YCSB C and D (100% reads) remains similar in both cases, as the database is not large enough to cause significant amounts of local I/O in the Baseline case.

Latency Figure 5.7 presents the mean and tail client response times for the MongoDB Baseline and Hailstorm for both request distributions using 3 representative workloads: YCSB A, C, and I. Hailstorm significantly improves response times under skew, especially for write workloads. For example, it reduces the mean response time by 37%, 18%, and 29%, as well as tail latencies by ~4×, ~30%, and ~5× for YCSB A, C, and I respectively. Also, Hailstorm does not adversely affect the mean response times in the uniform case.

5.5.3.2 Production Traces

Figure 5.8 compares the average throughput achieved by MongoDB for Baseline and Hailstorm and for both production workloads from Nutanix.

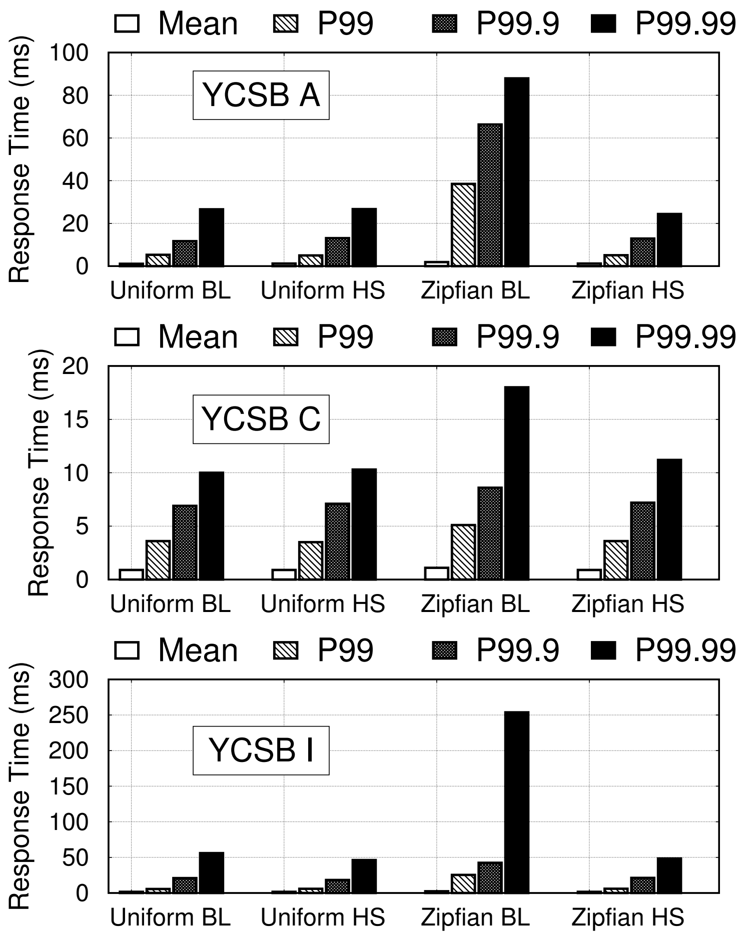


Figure 5.7 – MongoDB mean and tail response times for Baseline (BL) and Hailstorm (HS) for YCSB A, C, and I with uniform and Zipfian requests distributions.

Hailstorm provides consistent throughput for both Nutanix 1 (uniform) and Nutanix 2 (skewed). In contrast, the MongoDB Baseline suffers from a 3× performance degradation on Nutanix 2 resulting from a hotspot on one of the database instances. Hailstorm, therefore, does not add significant overhead on uniform workloads and successfully maintains performance close to uniform when the workload is skewed.

5.5.3.3 Large Database

Until now, we have shown the significant benefits of Hailstorm when workloads have skew. In this experiment, we show that Hailstorm benefits uniform workloads as well. Figure 5.9 shows the throughput over time for Baseline and Hailstorm with YCSB A on a uniform distribution starting with a large 1 TB database. Baseline performance suffers from sudden drops that increase over time as a result of larger compactions taking place, while Hailstorm has consistent performance over time. This experiment shows that even with uniform workloads, I/O bursts due to background operations cause skew across storage engines.

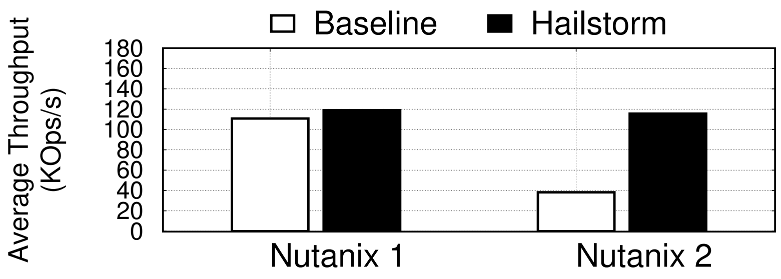


Figure 5.8 – MongoDB average throughput for Baseline and Hailstorm and for Nutanix's workloads. Hailstorm improves throughput in both cases, especially for workload Nutanix 2.

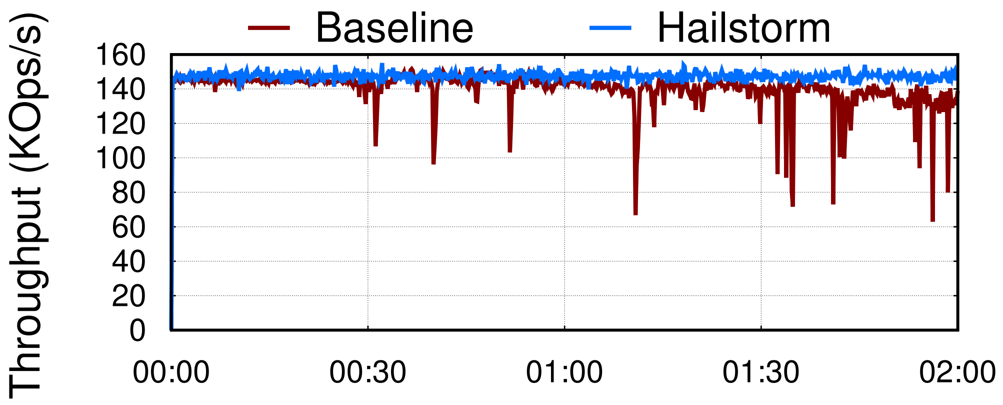


Figure 5.9 – MongoDB per-second throughput for Baseline and Hailstorm on a large 1 TB database with YCSB A and uniform distribution. Baseline experiences drops over time as larger compactions occur, causing load imbalance across storage engines.

5.5.3.4 Resharding Costs

Table 5.3 shows the average throughput in MongoDB for Baseline and Hailstorm for YCSB A with Zipfian distribution and with resharding enabled or disabled.

	Resharding=OFF	**Resharding=ON**
Baseline	42.9 KOps/s	58.9 KOps/s
Hailstorm	130.2 KOps/s	113.0 KOps/s

Table 5.3 – MongoDB average throughput for Baseline and Hailstorm for YCSB A Zipfian distribution with resharding enabled or disabled.

This table presents several interesting results. First, turning resharding off for MongoDB causes throughput to drop by 27%. Resharding in MongoDB is beneficial in skewed workloads. Second, Hailstorm performs better than Baseline with or without resharding, indicating that storage pooling and compaction offloading are more effective than resharding. Indeed, proper skew mitigation requires synergistic approaches at both the distributed database and storage layers. Finally, resharding causes a 15% throughput drop for Hailstorm due to increased I/O operations. This justifies disabling MongoDB's balancer for experiments with Hailstorm.

5.5.4 Distributed SQL Transactions: TiDB

We now consider distributed SQL transactions in TiDB [25], a popular horizontally-scalable database compatible with MySQL [80]. TiDB is built on top of TiKV [20], a distributed database with a key-value interface.

Workloads We use both the industry-standard TPC-C [138] benchmark and the more recent TPC-E [65] benchmark. TPC-C models a number of warehouses with orders, entries, payments, monitoring of stock, etc. Multiple transactions execute simultaneously, and the performance metric is the number of new-order transactions per minute (*tpmC*). TPC-E models a broker whose customers generate trades, account balance checks, market analysis, etc., and the performance metric is the number of trade-result (executed trades) transactions per second (*tpsE*). Both benchmarks also include a price per performance metric based on the total cost of ownership of the cluster used for 3 years.

Bench	Model	Tables	Txs	R:W	RW:RO	Sec Idx
TPC-C	Warehouses	9	5	65:35	92:8	2
TPC-E	Brokerage	33	12	91:9	23:76	10

Table 5.4 – TiDB benchmarks description and characteristics.

The main characteristics of each benchmark are shown in Table 5.4, including the type of business modeled by the benchmark, the number of tables, the number of distinct transactions, the I/O read to write ratio (R:W), the read-write to read-only transaction ratio (RW:RO), and the number of transactions using a secondary index. [66] contains more details and comparisons about these benchmarks.

Benchmark Results Table 5.5 summarizes the benchmark results for both TPC benchmarks, with Baseline and Hailstorm. We show performance and price per unit of performance (price-performance) metrics for our cluster. We conclude that Hailstorm provides significant performance improvements and cost reduction for distributed databases.

	TPC-C		TPC-E	
Configuration	**tpmC**	**$ / tpmC**	**tpsE**	**$ / tpsE**
Baseline	32,184	3.10	277.3	360.60
Hailstorm	50,178	2.00	408.1	245.05

Table 5.5 – TiDB TPC-C and TPC-E results for Baseline and Hailstorm. Estimated total system cost for our cluster is USD 100,000. Hailstorm improves throughput by 1.56× and 1.47× respectively.

Figure 5.10 compares the throughput (measured in transactions per second) over 1 hour for both scenarios and both benchmarks.

Baseline suffers from unstable throughput and frequent, drastic drops in throughput in both benchmarks. These drops are caused by compactions running on TiKV instances and resharding operations executed by placement drivers trying to remove hotspots. We notice a

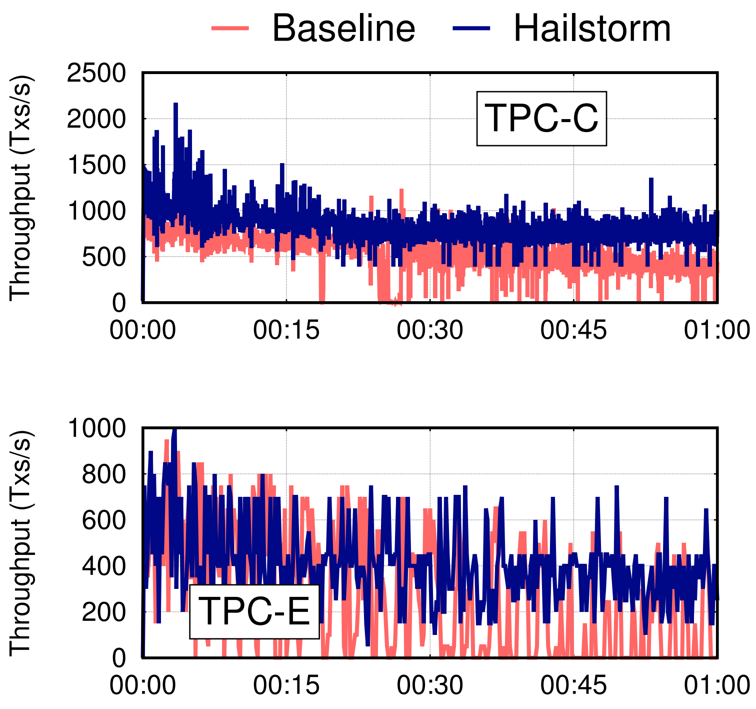


Figure 5.10 – TiDB per 10-second throughput timelines (HH:mm) for Baseline and Hailstorm with TPC-C and TPC-E.

significant amount of data migration due to TiDB's resharding policies. Short bursts of data migration consume as much as 90% of the I/O bandwidth for a single TiKV instance and are responsible for prolonged drops at approximately 00:30 and 00:40 for TPC-C. Although TPC-C's request distribution is uniform and TPC-E is only mildly skewed, there is significant imbalance across TiKV instances due to compaction and uneven data placement.

Hailstorm offers a more stable and overall higher throughput than Baseline. Compaction offloading helps limit pressure on overloaded instances, and storage pooling removes many I/O bottlenecks.

5.5.5 Comparison with HDFS

In this section, we compare Hailstorm with HDFS [148], an existing production distributed file system. We perform experiments directly on standalone RocksDB [57], thus avoiding any interference or overheads of using a distributed database. To this end, we design a microbenchmark that uses YCSB and its driver for embedded RocksDB.

Workloads We consider three custom YCSB workloads: a read-only workload, a write-only workload, and a mixed workload consisting of 50% writes and 50% reads. Keys are selected uniformly at random, and values are 1 KB each.

Configurations We run each workload on 8 machines in parallel using separate RocksDB databases in 4 configurations *i:8* for i values of 8, 4, 2, and 1. An i:8 configuration represents a scenario where 8 machines are running a RocksDB database, but only i of them are executing the workload with the other machines remaining idle. *8:8* corresponds to uniform, *4:8* to mild skew, *2:8* to intermediate skew, and *1:8* to high skew.

We execute each of the above 4 configurations 4 times: first with RocksDB using the local ext4 [114] filesystem, thereby establishing a Baseline (BL), then with RocksDB using HDFS with a replication factor of 1 to maximize performance, then with RocksDB using Hailstorm for storage pooling (HS-SP), and finally with RocksDB running on top of Hailstorm with both storage pooling and compaction offloading (HS).

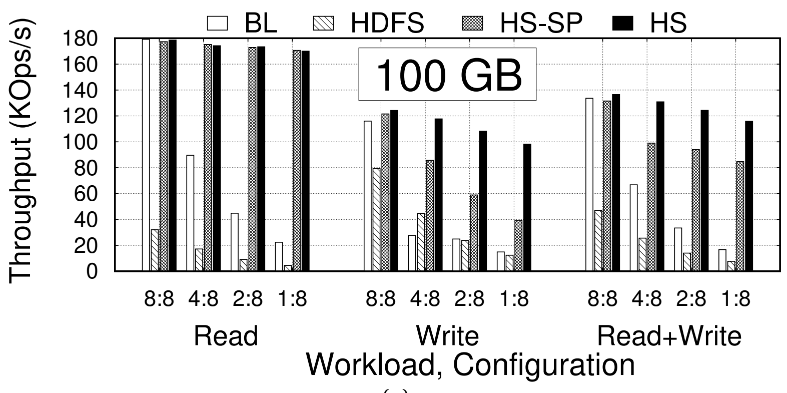
We run this experiment with two workload sizes: 100 GB (storage workload, the dataset does not fit in memory), and 20 GB (in-memory workload, the dataset fits in memory). The in-memory workload shows Hailstorm performance when the workload is CPU bound. We first discuss the results for the storage workload and then the in-memory workload.

Storage Workload Results Figure 5.11a shows the aggregate throughput (over all 8 machines) for each of the 3 workloads in each of the 4 configurations for the 100 GB workload.

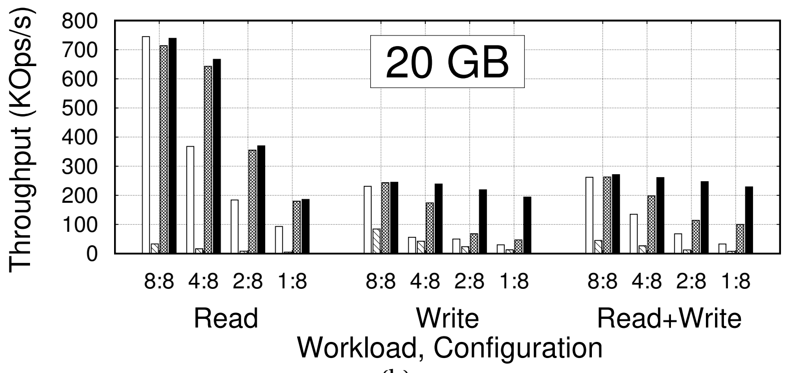
Hailstorm in the *8:8* configuration (uniform case) fares comparably with vanilla RocksDB. However, in the presence of skew, as expected from previous experiments, Hailstorm's throughput is much higher than the corresponding vanilla RocksDB (Baseline) throughput. Storage pooling and compaction offloading together enable Hailstorm to keep throughput close to the *8:8* configuration. Hailstorm performance decreases mildly with increasing skew due to remote reads and writes and increased compaction offloading.

In contrast, the throughput of RocksDB over HDFS is lower than the corresponding Baseline case, even though it uses distributed storage. We profile this experiment and find that the low performance stems from synchronous calls to the namenode before accessing data, performing I/O one block at a time, and writing blocks preferentially to the local disk. This shows the need for a specialized filesystem designed to maximize storage bandwidth. As an aside, we also experimented with running RocksDB on two full-featured, distributed filesystems (Ceph [156] and GlusterFS [72]), and unfortunately both filesystems would invariably crash after some time, and RockDB would hang, for unclear reasons.

In-memory Workload Results Figure 5.11b shows the aggregate throughput results for the 20 GB CPU-bound workload. The read workload results show that Hailstorm improves performance compared to the baseline by roughly a factor of 2 under skew. This benefit results from storage pooling, which allows the initial dataset to be loaded faster from disk. To understand why Hailstorm read performance goes down with increasing skew, we measured the maximum achievable RocksDB random read throughput on a single machine using a RAM disk and found that the system becomes CPU-bound at 200 KOps/s. RocksDB spends significant times on



(a)



(b)

Figure 5.11 – RocksDB aggregate throughput comparison between Baseline (BL), RocksDB over HDFS (HDFS), Hailstorm with storage pooling (HS-SP), and full Hailstorm (HS) on 8 machines. 3 workloads are considered in 4 different configurations with increasing skew, and data sizes of 100 GB and 20 GB.

binary search to find a random key, decompression, and checksums, which limits Hailstorm performance.

Write and read+write numbers are qualitatively similar for both workload sizes. When comparing with the 100 GB workload, the 20 GB write and read+write throughputs are almost double since the data can be cached in memory. However, unlike the read-only workloads, the throughput cannot go over 2× due to write-ahead logging. Also, HS-SP throughput suffers from a steep drop from 4:8 to 2:8 because the bottleneck switches from storage to CPU. This is not the case for HS due to compaction offloading.

5.5.6 Sensitivity Analysis

We perform a sensitivity analysis for the compaction offloading threshold θ using two machines: node1, which receives full load, and node 2, which receives 10%, 50%, or 80% of the full load. We use the same workloads and configurations as in the previous section (§5.5.5). For each scenario, we consider three θ values: 0.1, 0.2, and 0.5. Figure 5.12 reports the average throughput with the read+write workload for each instance in each scenario.

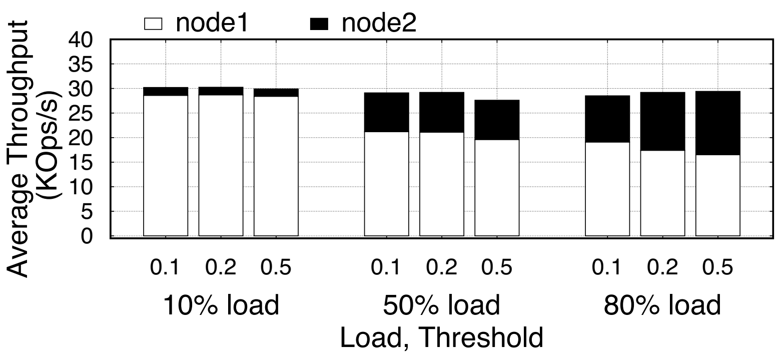


Figure 5.12 – RocksDB average throughput for the read+write workload with different compaction offloading thresholds θ using two RocksDB instances where one machine receives 10%, 50%, or 80% load.

Overall, different θ values have little impact. Large values (e.g., $\theta = 0.5$) make compaction offloading less frequent, and thus lead to slightly lower throughput on overloaded machines.

5.5.7 Using Hailstorm with B-Trees

Although Hailstorm is primarily intended for use with LSM-based storage engines, we expect storage pooling to provide benefits even when storage engines are based on B-trees, e.g., Aerospike [23], Couchbase Server [59], KVell [104], and WiredTiger [34]. B-trees exhibit different access patterns and storage behavior than LSMs and do not require compaction [67].

Figure 5.13 compares the average throughput achieved by MongoDB with the B-tree-based WiredTiger [34] storage engine for Baseline and Hailstorm for all YCSB workloads in Table 5.2 using both uniform and Zipfian distributions on 8 machines. We only use Hailstorm for storage pooling and disable compaction offloading.

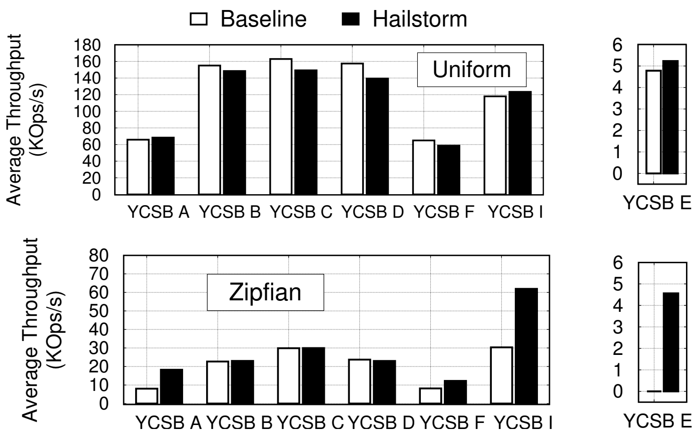


Figure 5.13 – MongoDB with WiredTiger [34] storage engine average throughput for Baseline and Hailstorm for YCSB workloads with uniform and Zipfian key distributions.

Unlike with LSM stores, Hailstorm does not improve performance for reads in the presence of skew because the CPU, not the I/O, is the bottleneck. Hailstorm's storage pooling provides ~2× throughput improvements in the Zipfian case for write workloads YCSB A, F, and I. Hailstorm also improves performance for range-based queries in YCSB E as it partially relieves the overloaded machine that becomes a straggler. We expect that offloading B-tree background tasks such as garbage collection in a similar way as we offload compaction tasks in LSMs would further improve write performance.

5.6 Conclusions

As the scale of distributed databases grows and their performance requirements become more stringent, solutions that can address challenging issues such as the presence of skew at scale become necessary. We have made a case for deploying distributed databases over Hailstorm, a system that disaggregates compute and storage in order to scale both independently and improve load balance. Hailstorm consists of a storage layer that pools storage across the machines of a rack, allowing each storage engine to utilize rack storage bandwidth. This effectively provides storage-level sharding, which helps mitigate the impact of skew, addresses per-machine capacity limitations, and absorbs I/O spikes. Hailstorm leverages its storage layer to perform compaction offloading and reduce CPU pressure on overloaded machines.

We include a summary of Hailstorm characteristics and specific features inspired by the scatter architecture below.

Area	Distributed LSM-based databases
Load imbalance	Partitioning skew Queries Compactions
Data spreading	Sstables as files Write-ahead logs as files
Dynamic parallelism	Compaction offloading
Partitioning	Range- or hash-based (database-specific)
Misc. characteristics	POSIX filesystem Fine-granularity reads optimization

6 Related Work

In this chapter, we survey the state-of-the-art closest to the work presented in this thesis. We begin by identifying and discussing literature related to the scatter architecture presented in Chapter 2. We then discuss related work for Chaos (Chapter 3), Hurricane (Chapter 4), and Hailstorm (Chapter 5).

6.1 Storage Disaggregation and Pooling

6.1.1 Disaggregated Storage

Many systems performing storage disaggregation were proposed recently.

Flat datacenter storage [124] (FDS) shows how one could take our assumption of local storage bandwidth being the same as remote storage bandwidth and scale it out to an entire datacenter. Remote storage access is minimal for small-to-medium clusters with sufficient bisection bandwidth [55]. The scatter architecture exploits this property to decouple storage and computation makes it possible to achieve better utilization and load balance across workloads. Also, unlike FDS (and similar systems such as CORFU [49]), we leverage pseudorandom uniform data placement to remove the central bottleneck of a metadata server.

Flash storage disaggregation [100] aims to improve storage capacity and IOPS utilization by accessing remote flash devices. The scatter architecture is partially inspired by this work but focuses more on load imbalance concerns. Each file in the scatter storage architecture is sharded at block-level and distributed uniformly across all storage devices. Also, we do not perform disaggregation at block-level, but rather at file-level, allowing us to support high-level operations such as compaction offloading.

LegoOS [147] is an operating system designed specifically for hardware resource disaggregation. However, storage is not the main focus of this work, and LegoOS does not attempt to achieve load balance or maximize storage utilization.

6.1.2 Distributed In-Memory Storage

Several distributed in-memory storage systems have been proposed recently to improve performance and mitigate load imbalance [55, 79, 127]). These systems apply pooling higher up the memory hierarchy by pooling main memory to improve performance for workloads whose memory footprint varies. Although the scatter architecture leverages the buffer cache when pooling storage devices, its primary focus remains on pooling secondary storage. We also address load imbalance for both storage and computation.

6.1.3 Distributed Filesystems

Distributed parallel filesystems such as HDFS [148], GFS [87], GlusterFS [72], or Ceph [156] are used in large scale intra-/inter- datacenter deployments [30]. These systems often focus on providing high availability and scalability by spreading blocks of data and replicating them across servers. In contrast, the scatter architecture aims to improve load balance and storage utilization. As a result, we spread all data blocks uniformly in a pseudorandom fashion. This deterministic data placement strategy allows us to design a decentralized data plane and to maximize storage utilization via client-based prefetching. Our storage architecture also intentionally does not support concurrent parallel accesses to the same file, leaving such concerns up to the application developers, and thereby greatly simplifying metadata and consistency.

6.1.4 Two-Level Sharding

Social Hash [146] is a framework running in production at Facebook that aims to optimize query response time and load balance in large social graphs by using two-level sharding. In this scheme, data objects are first partitioned using a graph partitioning algorithm before being dynamically assigned in groups. By dynamically assigning data objects to groups, the system can react to changes in the workload and improve load balance. Similarly, scatter storage leverages the increased flexibility offered by using a two-step data assignment scheme but works with files and data blocks rather than objects. Social Hash, like Akkio [47], relies on locality to improve response time. In contrast, our approach abandons locality entirely and spreads data blocks uniformly across all storage devices.

6.1.5 Storage Utilization

The batch sampling technique used in the scatter storage architecture is inspired by *power-of-two scheduling* [117], although the goal is quite different. Power-of-two scheduling aims to find the least loaded servers to achieve load balance. Our approach aims to prevent storage engines from becoming idle through oversubscription.

6.2 Graph Processing

In recent years a large number of graph processing systems have been proposed [64, 89, 90, 94, 99, 103, 111, 122, 123, 125, 137, 144, 155, 163]. We mention here only those most closely related to our work.

6.2.1 Distributed In-Memory Systems

Pregel [111] and its open-source implementation Giraph [1] follow an edge-cut approach. They partition the vertices of a graph and place each partition on a different machine, potentially leading to severe load imbalance. Mizan [99] addresses this problem by migrating vertices between iterations in the hope of balancing load in the next iteration. Chaos addresses load imbalance within each iteration, by allowing more than one machine to work on a partition if needed. Pregel optimizes network traffic by aggregating updates to the same vertex. While this optimization is also possible in Chaos, we find that the cost of merging the updates to the same vertex outweighs the benefits of reduced network traffic.

PowerGraph proposes the GAS model, and PowerLyra [64] introduces a more straightforward variant, which we adopt in Chaos. PowerGraph [89] introduces the vertex-cut approach, partitioning the set of edges across machines and replicating vertices on machines that have an attached edge. PowerLyra improves on PowerGraph by treating high- and low-degree nodes differently, reducing communication and replication. Both systems require lengthy pre-processing times. Also, in both systems, each partition is assigned to exactly one machine. In contrast, Chaos performs only minimal pre-processing and allows multiple machines to work on the same partition.

GraM [157] has shown how a graph with a trillion edges can be handled in the main memory of the machines in a cluster. Chaos represents a different approach where the graph is too large to be held in memory. Thus, while Chaos is slower than GrAM it requires only a fraction of the amount of main memory to process a similarly sized graph.

6.2.2 Single-Machine Out-Of-Core Systems

GraphChi [103] was one of the first systems to propose graph processing from secondary storage. It uses the concept of parallel sliding windows to achieve sequential secondary storage access. X-Stream [144] improves on GraphChi by using streaming partitions to provide better sequentiality. GridGraph [163] further improves on both GraphChi and X-Stream by reducing the amount of I/O necessary. Chaos extends out-of-core graph processing to clusters.

6.3 General-Purpose Analytics

Skew Mitigation SkewTune [102] mitigates skew in MapReduce programs by identifying slow tasks and repartitioning them to run on idle nodes. Since the system is intended to be a drop-in replacement for MapReduce, it suffers similar limitations, namely that the output order must be preserved and the data placed locally on the original worker. While this approach can help with skew, it also causes significant data movement, which can overwhelm already overloaded workers. SkewTune can also worsen performance inadvertently by repartitioning tasks that are close to completion.

Camdoop [71] performs in-network data aggregation during the shuffle phase of MapReduce applications, which can help mitigate data skew by decreasing the amount of data moved and the overall load on the network. Unfortunately, this solution requires specialized hardware that is not currently available. We believe such hardware would also benefit Hurricane deployments. Tamed transformers allow implementing similar functionality in software.

Straggler tasks are a challenge for analytics workloads [43]. A commonly used method for handling stragglers is speculative execution, which involves detecting a straggler as soon as possible and restarting a copy of the task on another machine [42]. While this approach helps with machine skew, it does not address data or compute skew. Hurricane allows slower workers to split their task via cloning, avoiding the need to restart the task from scratch.

Garbage Collection (GC) can be a major cause of skew for applications written in garbage-collected languages such as Java, Scala, or Python. GC induces uncoordinated pauses across JVM [131], thereby reducing overall throughput and increasing tail-latency. Recent research attempts to mitigate this problem by synchronizing (or desynchronizing) garbage collection across all workers running the same application to minimize unpredictability [110]. Hurricane is also prone to GC pauses, but its decentralized design and finer-grained partitioning help reduce its impact.

6.3.1 Adaptive Partitioning of Work

As far as we can tell, Hurricane is the first cluster computing framework to adaptively partition work based on load observed by workers during task execution. This design is made possible through fine-grained data sharing among multiple workers executing the same task and programmer-defined merge procedures.

Several techniques based on over-decomposition have been proposed to split analytics jobs into smaller tasks to mitigate skew and improve load balance. These techniques require manual intervention from the programmer and are application- and input-specific. For instance, they require fine-tuning the programmer-defined split function [129], exploiting commutativity and associativity to combine identical keys[159], and/or splitting records for the same key across multiple partitions[16]. Hurricane mitigates skew in an application-independent manner by dynamically splitting partitions when a task is cloned. We have shown that our approach mitigates skew effectively, without requiring tuning of the application for specific data sets, and that it applies to arbitrary operations, such as finding unique values.

Traditional cluster computing frameworks split data into partitions and use shuffling and sorting to merge them back in an application-independent way [93, 119, 160]. This often comes at the cost of sorting intermediate outputs and prevents records with the same key being sent to multiple reducers, which can cause load imbalance in the presence of skew. More importantly, this approach places constraints on the shape of partitions, making it harder to redistribute a partition in a balanced way. Hurricane takes a different approach by empowering application developers to provide a custom merge method, when applicable. This merge subsumes the traditional shuffling and sorting paradigm while being more flexible because it allows the outputs of clones that have been created at any point during execution to be merged in an application-specific manner.

Although adding a merge procedure to existing frameworks is relatively simple, taking full advantage of it would require significant re-engineering and changes to the execution model to allow for tasks to be repartitioned on-the-fly. Fault tolerance mechanisms would also need to be adapted to account for the possible presence of multiple partial outputs. Finally, frameworks that rely on key sorting to send records to the appropriate reduce may also end up losing the ability to combine records by key as a result of such changes.

6.3.2 Joins

Load balancing for parallel joins has been extensively studied in parallel databases. Earlier work [83, 128] focused on careful partitioning based on input sampling to achieve load balance. At the same time, more recent approaches [145] use late binding to gain flexibility and reassign partitions to other workers. Hurricane requires less focus on partitioning, relying instead on cloning and merging for handling skew and load imbalance.

6.3.3 Distributed Scheduling

Support for scheduling tiny tasks has led to the design of distributed or hybrid schedulers such as Sparrow [132] or Hawk [76]. Sparrow uses a batch sampling algorithm to schedule tasks, whereas Hawk partitions the cluster for large and small jobs, and uses a randomized work stealing algorithm to place short jobs. Hurricane also recognizes the need for efficient scheduling, in particular for clones, and schedules tasks in a distributed, decentralized way through work bags.

6.3.4 Remote Memory

When analytics applications suffer from skew in their input or intermediate data, they may be forced to spill data to disk because it does not fit in memory, leading to serious performance issues. Spongefiles [81] allows machines with large datasets to use the memory of remote machines as a backing store, thereby avoiding spilling. Hurricane spreads data by default across all machines through the bag abstraction. Since files back bags, the spreading of data helps even when the dataset size does not fit in the main memory of the entire cluster, because it allows spreading the disk I/O.

6.3.5 Avoiding Scale-Out

Nobody ever got fired for using Hadoop on a cluster [142] suggests that many analytics workloads and applications should not use scale-out solutions but instead run on single machines with large main memory. This is particularly true when processing datasets whose size is comparatively small (e.g., less than 100 GB). Hurricane targets scenarios with larger input sizes.

6.4 Distributed Databases

Distributed databases distribute data across many nodes, often on a global scale. Many new databases rely on distributed KV stores and store data on local storage [19, 20, 23, 25]. Other databases such as Apache HBase [154] or Google Spanner [69] rely on a distributed filesystem to store data, which they leverage mainly for data replication. Hailstorm is primarily intended for databases using local storage, effectively providing storage-layer resharding to remove I/O hotspots by spreading block data uniformly across storage devices.

6.4.1 Shard Rebalancing

Skew is often intrinsic to the application and cannot easily be removed. It is often mitigated manually or automatically using shard rebalancing [35, 69]. This is achieved by identifying hotspots in the workload and migrating data to less utilized database instances [47, 105]. However, shard rebalancing requires in-depth knowledge of the shape of the data managed [48]. Furthermore, rebalancing data between shards is costly as it triggers additional compaction, flushing, and garbage collection tasks, and is performed too late to be effective. Hailstorm automatically shards data uniformly across all storage devices, which works in all cases without requiring knowledge of the shape of data.

6.4.2 Compaction Offloading

Using a dedicated remote compaction server for a replicated store has been previously proposed in the context of HBase [41]. In this scheme, the system offloads large compactions to a dedicated remote compaction server relying on replication to provide fast data access. Hailstorm takes a different approach by exploiting rack-locality to create a storage management layer underneath storage engines that allows fast access to data without depending on replication. We offload compaction tasks in a peer-to-peer manner that does not require complex centralized decision making. Finally, our solution works for any distributed database using LSM storage engines.

6.4.3 LSM KV Stores

Many systems attempt to solve the write amplification problem in LSMs. HyperLevelDB increases parallelism and modifies the compaction algorithm to reduce compaction costs [29]. PebblesDB combines LSM with skip-lists to fragment data in smaller chunks, thereby avoiding complete rewrites of sstables within a level [139]. TRIAD delays compactions until there is sufficient overlap and pins “hot” key entries in memory to avoid creating many copies on storage [50]. Silk attempts to opportunistically execute compactions during low load and preempt them at high load [51]. While these approaches provide temporary relief, they often lead to higher costs in the long run as uncompacted or fragmented LSMs suffer from increased read latency and delayed compactions inevitably trickle down the LSM levels. Furthermore, all these solutions apply to a single node configuration and do not take advantage of distributed storage. Nevertheless, these optimizations are orthogonal to our approach and could be combined with it.

6.4.4 B-Tree Load Balancing

Yesquel’s approach to splitting B+ tree nodes improves load balance and reduces contention but does not achieve uniform distribution across all database instances [39]. Furthermore, this approach may lead to many unnecessary splits if load intensity varies across keys over time. Although Hailstorm provides improvements mainly for write-intensive workloads, our block-level sharding would still improve storage load balance in Yesquel. Although we have not explored this, it should be possible for Yesquel or other similar systems to use Hailstorm for split offloading. MoSQL relies on a B-tree design and keeps all data in main memory for fast access [149]. This reduces contention and load imbalance but places a hard cap on the total size of the database. Hailstorm has no such limitation since we use secondary storage.

7 Conclusions

How should we design scalable and load-balanced applications for computer clusters? The answer proposed in this dissertation is simple, yet surprising: abandon data locality. We propose a new architecture for distributed systems that separates computation and storage, enabling each resource to scale independently, and subsequently, pools all compute and storage resources together to spread the load evenly. We show how to balance load across storage devices by spreading data uniformly and designing a decentralized and randomized access scheme based on overprovisioning. We also provide solutions to balance compute load across the processing resources of a cluster by adjusting parallelism on demand using a bag abstraction that enables dynamic work sharing and a background task offloading mechanism.

While our approach may not always be possible to integrate into current mainstream systems, we hope that, at the very least, it provides a series of useful guidelines to inspire the design and implementation of future systems. Cluster computing will no doubt evolve in the coming years, as new applications, new systems, new requirements, and new challenges appear. Even if the scatter architecture is not a part of this evolution, we believe that its main contribution is to question the overreliance on data locality in the era of Big Data.

We anticipate that, with the coming data deluge, there will be a need to store and process Big Data in an efficient and load-balanced way. In this context, it is critical to provide tools that enable application developers to focus on writing high-value business application code rather than spending time fine-tuning partitioning and system parameters to achieve load balance, high utilization, and good performance.

In the rest of this chapter, we summarize the results obtained with the three scatter applications presented in this thesis. We then discuss a few of the lessons learned. Finally, we sketch a few directions for future research.

7.1 Scatter Applications

We implemented three applications based on the scatter architecture and demonstrated how the resulting systems achieve load balance. Table 7.1 summarizes our results. For each system, we list its area of application, the data structures that are spread across all disks, the technique used to achieve dynamic parallelism, as well as the maximum evaluated imbalance ratio between machines (Max Imb.) and the resulting slowdown experienced by the system compared to a uniform distribution.

System	**Area**	**Spreading**	**Parallelism**	**Max Imb.**	**Slowdown**
Chaos	Graphs	Streaming Partitions	Work Sharing	23:1	2.3×
Hurricane	Analytics	Data + Work bags	Work Sharing	64:1	2.4×
Hailstorm	Databases	Sstables + WAL	Offloading	11:1	1.1×

Table 7.1 – Summary of the key results presented in this thesis.

7.2 Lessons Learned

Locality is nonessential for Big Data. The main thread underlying the work presented in this thesis is the departure from data locality as a first-class principle in high-throughput cluster computing systems. Locality should not be treated as a primary requirement but rather as a nonessential optimization. While this position may seem controversial, we observe that data locality, specifically placing partitions on the machines that must perform computation on them, is fundamentally problematic to ensure load balance in systems that process large amounts of data.

It is important to point out again that abandoning locality is only feasible if network bandwidth is larger than storage bandwidth. This assumption is valid today for typical infrastructure dedicated to batch processing or data storage. As storage and network technologies evolve in the future, new trends may invalidate this assumption. However, in a scenario where the storage becomes systematically faster than the network, it may no longer be worthwhile to scale out many applications as they could instead leverage fast and large local storage in a single machine to process data efficiently.

Load imbalance is everyone's problem. We argue in this dissertation that load imbalance should not be addressed unilaterally at a single layer or component, e.g., by improving partitioning or rebalancing partitions. In other words, optimizing the bottleneck without taking into account the context of an entire system will likely only move it elsewhere. Successfully mitigating load imbalance requires a more synergistic approach that involves all layers and resources of a distributed system. The ultimate goal of the scatter architecture is for the system bottleneck to be on an aggregated resource, e.g., storage, ensuring maximum achievable throughput.

Pray for the best, prepare for the worst and expect the unexpected. A key decision in the scatter architecture design is the combination of a proactive approach (spreading data uniformly) and a reactive approach (adjusting parallelism within a partition dynamically). The combination of both approaches is what makes the scatter architecture so effective at dealing with load imbalance by avoiding uneven data distribution ahead of time while also actively working on getting out of an imbalanced state when it (inevitably) occurs.

Moving computation is cheaper than moving storage. When locality is not a concern, it does not matter where the computation executes. Consequently, the scatter storage architecture brings data to the computation rather than the converse. In this architecture, computation can execute anywhere and multiple units of computation can operate simultaneously on separate subsets of the data. Not only does this pull-based approach at block granularity help overlap computation and communication, but it also provides late binding of data to compute nodes, a key property for load balance. When combined with a fixed number of outstanding requests per storage client, this approach ensures that compute nodes self-regulate the data ingestion rate and avoid data starvation.

Avoid centralization when possible. An intriguing consequence of the scatter storage design decision to spread data in blocks uniformly across storage devices in a locality-oblivious manner is the fact that doing so removes the need for a centralized directory service. This demonstrates that relaxing locality allows us to simplify our design and remove a potential bottleneck and a single point of failure.

Summarizing the trade-offs between locality and load balance explored in this dissertation inspires us a simple trilemma theorem regarding the incompatibility of these two properties in high-throughput distributed systems.

Theorem 1 (BLP Theorem)**.** *It is impossible for a Big Data distributed system to provide more than two of the following three guarantees simultaneously:*

- ***Balance**: The system achieves load balance in both compute and storage.*
- ***Locality**: The data for every partition is always stored and processed entirely on the same machine.*
- ***Partitioning**: The system is distributed, and the dataset is divided into partitions processed assigned to disjoint machines.*

This theorem follows intuitively from the fact that it is impossible to find perfectly balanced partitions in many problems, and therefore enforcing data locality will cause load imbalance.

We illustrate this trilemma in Figure 7.1. The scatter architecture chooses balance and partitioning at the expense of locality (BP). Many existing systems, such as Apache Hadoop or

Spark opt for locality and partitioning at the expense of load balance (LP). Finally, selecting both locality and load balance corresponds to a single-machine system (BL).

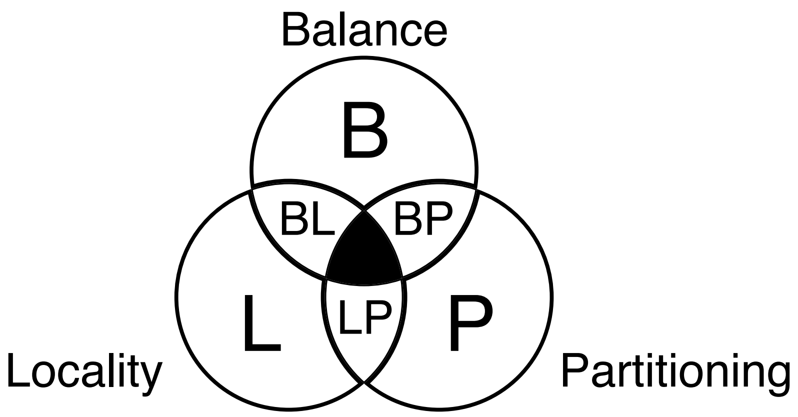


Figure 7.1 – BLP Theorem. Achieving load balance (B), locality (L), and partitioning (P) in a single distributed system is impossible. Pick any two of these properties.

7.3 Future Work

The scatter architecture described in this thesis suffers from various limitations. We list some examples below:

- **Priorities.** Scatter storage servers treat all requests from clients in a FIFO order, but some applications could benefit from the ability to send priority requests, e.g., for latency-sensitive operations. Hailstorm supports simplified priorities by using a different read granularity for requests issued by a foreground or background thread.

- **Heterogeneous resources.** When spreading data uniformly across all storage devices, we assume that all storage devices have the same capacity and support the same bandwidth. While this is generally verified within a rack or a small cluster, it is desirable to incorporate support for uniform distribution based on available resources.

- **Adaptive batch sampling factor.** As demonstrated, the batch sampling strategy described in Subsection 2.1.4 works well in practice. However, there are extreme cases where it may be sub-optimal, for example, when a client is alone accessing a large number of servers. A worthwhile improvement to the batch sampling algorithm would be for clients to dynamically adjust their batch sampling factor based on overall storage server load to maximize storage utilization.

We also suggest additional scatter applications in the three areas surveyed in this thesis.

Graph Processing Applications Graph pattern mining algorithms look for instances of interesting patterns in a graph. This process generates exponentially more data than the input size. It often exhibits skew in the distribution of pattern instances over the graph, making it a prime candidate for the scatter architecture. Another possible application would be graph databases where the data is structured in a graph and retrieved using graph-style queries. Due to the difficulty of finding good partitions in graphs, distributed graph databases are natural candidates for a scatter approach.

Analytics Applications In this thesis, we explore Big Data but do not focus on "Fast Data", i.e., stream processing. Load balance may be a lesser concern in low latency, continuous processing applications, but it is nonetheless an exciting area to explore. A similar approach as Spark's micro-batching to support streaming could be investigated with Hurricane.

Another area of analytics that garnered much attention lately is Machine Learning (ML). Many ML problems exhibit skew or load imbalance in some form. For instance, supervised learning algorithms may be subject to significant load imbalance, e.g., due to uneven class distribution in a classification problem or because of skew in the distribution of events during logistic regression. Similarly, reinforcement learning is well-known to manifest high variance in the exploration process due to pruning or horizon-mitigating techniques.

Distributed Database Applications Hailstorm explored the feasibility of inserting a filesystem substrate below database storage engines to transform distributed databases into scatter applications. While successful in achieving load balance, the system is limited by the original database design. A worthwhile avenue for future research would be building a scatter distributed database from the ground up instead. This approach could incorporate new features, such as the ability to service requests for a shard from multiple machines, thereby achieving dynamic work sharing for distributed databases, or the ability to migrate data across racks more efficiently to replace shard rebalancing.

Finally, we very much look forward to the scatter architecture being used in new and exciting application areas.

Bibliography

[1] http://giraph.apache.org/.

[2] http://www.twitter.com/.

[3] http://www.graph500.org/.

[4] http://www.graph500.org/results_jun_2014.

[5] https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920.

[6] http://zeromq.org/.

[7] http://webdatacommons.org/hyperlinkgraph/.

[8] http://freecode.com/projects/fio.

[9] http://www.dbms2.com/2011/07/06/petabyte-hadoop-clusters/, 2011.

[10] https://storm.apache.org/, 2018.

[11] https://wiki.apache.org/hadoop/PoweredBy, 2018.

[12] http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html, 2018.

[13] http://akka.io/, 2018.

[14] https://netty.io/, 2018.

[15] https://github.com/real-logic/Aeron, 2018.

[16] http://cgnal.com/blog/using-spark-with-hbase-and-salted-row-keys/, 2018.

[17] Stack overflow, 2018.

[18] https://flink.apache.org/, 2019.

[19] https://www.mongodb.com/, 2019.

[20] https://tikv.org/, 2019.

[21] http://cassandra.apache.org/, 2019.

[22] https://github.com/Netflix/dynomite, 2019.

[23] https://www.aerospike.com/, 2019.

[24] https://github.com/mongodb-partners/mongo-rocks/, 2019.

[25] https://pingcap.com/, 2019.

[26] https://myrocks.io/, 2019.

[27] https://dgraph.io/, 2019.

[28] https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage, 2019.

[29] https://github.com/rescrv/HyperLevelDB, 2019.

[30] https://en.wikipedia.org/wiki/List_of_file_systems, 2019.

[31] https://github.com/libfuse/libfuse/, 2019.

[32] http://www.pnfs.com/, 2019.

[33] https://github.com/jnr/jnr-ffi, 2019.

[34] http://www.wiredtiger.com/, 2019.

[35] https://github.com/mongodb/mongo/wiki/Sharding-Internals, 2019.

[36] https://docs.mongodb.com/manual/core/sharding-balancer-administration/, 2020.

[37] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[38] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Evaluating cassandra scalability with ycsb. In *International Conference on Database and Expert Systems Applications*, pages 199–207. Springer, 2014.

[39] Marcos K Aguilera, Joshua B Leners, and Michael Walfish. Yesquel: scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 245–262. ACM, 2015.

[40] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 61–74. ACM, 2012.

[41] Muhammad Yousuf Ahmad and Bettina Kemme. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment*, 8(8):850–861, 2015.

[42] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.

[43] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[44] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax.* " O'Reilly Media, Inc.", 2010.

[45] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[46] Alexey Andreyev. Introducing data center fabric, the next-generation facebook data center network. *74145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network*, 2014.

[47] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: managing datastore locality at scale with akkio. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 445–460, 2018.

[48] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[49] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *Proceedings of the conference on Networked Systems Design and Implementation.* USENIX Association, 2012.

[50] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. *Proc. of ATC*, 2017.

[51] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, Renton, WA, July 2019. USENIX Association.

[52] Cristina Băsescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolić, and Ido Zachevsky. Robust data sharing with key-value stores. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[53] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *Proceedings of the*

Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, pages 301–316. ACM, 2020.

[54] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock You Like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the 13th EuroSys Conference*, EuroSys '18, pages 1–15. ACM, 2018.

[55] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: it's time for a redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[56] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[57] Dhruba Borthakur. Under the hood: Building and open-sourcing rocksdb. *Facebook Engineering Notes*, 2013.

[58] Eric A Brewer. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 167–167, 2015.

[59] Martin C Brown. *Getting Started with Couchbase Server: Extreme Scalability at Your Fingertips.* " O'Reilly Media, Inc.", 2012.

[60] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[61] Josiah L Carlson. *Redis in action.* Manning Shelter Island, 2013.

[62] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining.* SIAM, 2004.

[63] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[64] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the European Conference on Computer Systems*, pages 1:1–1:15. ACM, 2015.

[65] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3):5–10, 2011.

[66] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3):5–10, 2011.

[67] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[68] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[69] 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. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[70] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[71] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 3–3. USENIX Association, 2012.

[72] Alex Davies and Alessandro Orsaria. Scale out with glusterfs. *Linux Journal*, 2013(235):1, 2013.

[73] 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 - Volume 6*, pages 10–10. USENIX Association, 2004.

[74] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[75] 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*, volume 41, pages 205–220. ACM, 2007.

[76] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference*, number EPFL-CONF-208856, pages 499–510. USENIX Association, 2015.

[77] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1357–1374, 2019.

[78] Thibault Dory, Boris Mejías, Peter Van Roy, and Nam Luc Tran. Comparative elasticity and scalability measurements of cloud databases. In *Proc of the 2nd ACM symposium on cloud computing (SoCC)*, volume 11. Citeseer, 2011.

[79] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414. USENIX Association, 2014.

[80] Paul DuBois and Michael Foreword By-Widenius. *MySQL*. New riders publishing, 1999.

[81] Khaled Elmeleegy, Christopher Olston, and Benjamin Reed. Spongefiles: Mitigating data skew in mapreduce using distributed memory. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 551–562. ACM, 2014.

[82] Elmootazbellah Nabil Elnozahy, David B Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 39–47. IEEE, 1992.

[83] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. Scalable and adaptive online joins. *Proceedings of the VLDB Endowment*, 7(6):441–452, 2014.

[84] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms 2007 (AofA07)*, pages 127–146, 2007.

[85] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified NP-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[86] Sanjay Ghemawat and Jeff Dean. Leveldb. *URL: https://github. com/google/leveldb,% 20http://leveldb. org*, 2011.

[87] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 29–43. ACM, 2003.

[88] Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson, Dushyanth Narayanan, Florin Dinu, and Antony IT Rowstron. Rhea: Automatic filtering for unstructured cloud storage. In *NSDI*, volume 13, pages 2–5, 2013.

[89] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 17–30. USENIX Association, 2012.

[90] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 599–613. USENIX Association, 2014.

[91] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev.*, 39(4):51–62.

[92] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 522–533. IEEE, 2012.

[93] Apache Hadoop. Hadoop, 2009.

[94] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 77–85. ACM, 2013.

[95] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[96] Kien A Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, pages 525–535. Citeseer, 1991.

[97] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9, 2010.

[98] Călin Iorgulescu, Florin Dinu, Aunn Raza, Wajih Ul Hassan, and Willy Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 97–109, 2017.

[99] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[100] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 29. ACM, 2016.

[101] Yekesa Kosuru and John Cohen. Method and apparatus for rebalancing data, May 31 2012. US Patent App. 12/956,917.

[102] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[103] Aapo Kyrola and Guy Blelloch. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the Conference on Operating Systems Design and Implementation*. USENIX Association, 2012.

[104] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[105] Justin Li and Florian Weingarten. Zero-downtime rebalancing and data migration of a mature multi-shard platform. Dublin, October 2019. USENIX Association.

[106] Jimmy Lin et al. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, volume 1, 2009.

[107] John D.C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, May 1961.

[108] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *Proceedings of Very Large Data Bases (PVLDB)*, 8 2012.

[109] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[110] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *HotOS*, 2015.

[111] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the International Conference on Management of Data*, pages 135–146. ACM, 2010.

[112] Jasmina Malicevic, Amitabha Roy, and Willy Zwaenepoel. Scale-up graph processing in the cloud: Challenges and solutions. In *Proceedings of the International Workshop on Cloud Data and Platforms*, pages 5:1–5:6. ACM, 2014.

[113] Holger Märtens. A classification of skew effects in parallel database systems. In *European Conference on Parallel Processing*, pages 291–300. Springer, 2001.

[114] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[115] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data: the management revolution. *Harvard business review*, 90(10):60–68, 2012.

[116] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[117] Michael Mitzenmacher. The power of two choices in randomized load balancing. *Trans. Parallel Distrib. Syst.*, 12(10), 2001.

[118] Daniel G Murray. *Tableau your data!: fast and easy visual analysis with tableau software.* John Wiley & Sons, 2013.

[119] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[120] Jaeseok Myung, Junho Shim, Jongheum Yeon, and Sang-goo Lee. Handling data skew in join algorithms using mapreduce. *Expert Systems with Applications*, 51:286–299, 2016.

[121] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[122] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the Usenix Annual Technical Conference*, pages 291–305. USENIX Association, 2015.

[123] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[124] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 1–15. USENIX Association, 2012.

[125] Karthik Nilakant, Valentin Dalibard, Amitabha Roy, and Eiko Yoneki. PrefEdge: SSD prefetcher for large-scale graph traversal. In *Proceedings of the International Conference on Systems and Storage*, pages 4:1–4:12. ACM, 2014.

[126] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, pages 39–50. ACM, 2009.

[127] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. The case for rackout: Scalable data serving using rack-scale systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 182–195. ACM, 2016.

[128] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 949–960. ACM, 2011.

[129] Kay Ousterhout, Aurojit Panda, Josh Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In *HotOS*, volume 13, pages 14–14, 2013.

[130] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. Making sense of performance in data analytics frameworks. In *NSDI*, volume 15, pages 293–307, 2015.

[131] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. Making sense of performance in data analytics frameworks. In *NSDI*, volume 15, pages 293–307, 2015.

[132] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[133] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[134] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[135] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[136] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[137] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the International conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[138] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[139] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[140] Smriti R Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 16. ACM, 2012.

[141] David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical (idc white paper), 2017.

[142] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, pages 1–5, 2012.

[143] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424. ACM, 2015.

[144] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[145] Lukas Rupprecht, William Culhane, and Peter Pietzuch. Squirreljoin: network-aware distributed join processing with lazy partitioning. *Proceedings of the VLDB Endowment*, 10(11):1250–1261, 2017.

[146] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 455–468, 2016.

[147] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 69–87, 2018.

[148] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[149] Alexander Tomic, Daniele Sciascia, and Fernando Pedone. Mosql: An elastic storage engine for mysql. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 455–462. ACM, 2013.

[150] Ehsan Totoni, Subramanya R Dulloor, and Amitabha Roy. A case against tiny tasks in iterative analytics. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 144–149. ACM, 2017.

[151] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[152] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[153] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[154] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Computer science and network technology (ICCSNT), 2011 international conference on*, volume 1, pages 601–605. IEEE, 2011.

[155] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement: Scalable and programmable analytics over very large graphs on a single PC. In *Proceedings of the Usenix Annual Technical Conference*, pages 387–401. USENIX Association, 2015.

[156] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[157] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling graph computation to the trillions. In *Proceedings of the Symposium on Cloud Computing*. ACM, 2015.

[158] Zhen Ye and Shanping Li. A request skew aware heterogeneous distributed storage system based on cassandra. In *Computer and Management (CAMAN), 2011 International Conference on*, pages 1–5. IEEE, 2011.

[159] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.

[160] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[161] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[162] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, page 7, 2008.

[163] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the Usenix Annual Technical Conference*, pages 375–386. USENIX Association, 2015.

Laurent **Bindschaedler**

PH.D. CANDIDATE IN COMPUTER SCIENCE · EPFL

EPFL IC IINFCOM LABOS, BC 166 (Bâtiment BC), Station 14, CH-1015 Lausanne

(+41) 21-693-6977 | laurent.bindschaedler@epfl.ch | binds.ch | Google Scholar: dt14rAQAAAAJ | OrcID: 0000-0003-0559-631X

Summary

I am a Ph.D. student at EPFL, Switzerland. My interests lie in big data, machine learning, and cloud computing. I have been obsessed with software architecture and systems design since I was 12. I also enjoy teaching, entrepreneurship, and improvisational theatre.

Education

EPFL (Ecole Polytechnique Fédérale de Lausanne) *Lausanne, Switzerland*

PH.D. IN COMPUTER SCIENCE *Sep. 2015 - May. 2020 (Exp.)*

- Ph.D. Thesis: An Architecture for Load Balance in Computer Cluster Applications. Supervised by Prof. Willy Zwaenepoel.
- Oral exam passed on March 19, 2020.
- 30 ECTS credits. Depth area: Systems.
- Classes: Principles of Computer Systems (CS-522), Applied Data Analysis (CS-401), Natural Language Processing (CS-431).

EPFL (Ecole Polytechnique Fédérale de Lausanne) *Lausanne, Switzerland*

M.S. IN COMPUTER SCIENCE *Sep. 2009 - Jun. 2011*

- GPA: 5.54 (out of 6). 125 ECTS credits.
- M.S. Thesis: Track Me If You Can. Hosted by: Nokia Research Center. Supervised by Prof. Jean-Pierre Hubaux.

EPFL (Ecole Polytechnique Fédérale de Lausanne) *Lausanne, Switzerland*

B.S. IN COMPUTER SCIENCE *Sep. 2006 - Jun. 2009*

- GPA: 5.59 (out of 6). 189 ECTS credits.
- B.S. Thesis: Secure SMS. Supervised by Prof. Jean-Pierre Hubaux.

Employment History

EPFL (Ecole Polytechnique Fédérale de Lausanne) *Lausanne, Switzerland*

DOCTORAL ASSISTANT *Sep. 2015 - Jun. 2020*

- Ph.D. Candidate in the Operating Systems Laboratory (LABOS), IC. Supervised by Prof. Willy Zwaenepoel.
- IT administrator for the laboratory (since 2018), in charge of support, maintenance, and purchases.

EPFL (Ecole Polytechnique Fédérale de Lausanne) *Lausanne, Switzerland*

RESEARCH ASSISTANT *Jul. 2013 - Aug. 2015*

- Operating Systems Laboratory (LABOS), IC. Prof. Willy Zwaenepoel.
- Technical coordinator for the Data Center Observatory, a small state-of-the-art data center for researchers at EPFL, ETHZ, and USI.
- Co-designer and developer on Chaos, a graph processing system that enables analytics on very large graphs using secondary storage.
- Maintainer of X-Stream, a single-machine graph processing system based on streaming partitions.

LakeMind *Lausanne, Switzerland*

CO-FOUNDER AND VP OF ENGINEERING *Aug. 2011 - Jun. 2013*

- LakeMind makes cloud services more reliable by automatically troubleshooting and repairing service outages. As a result, the duration of service downtimes is drastically reduced and developers spend less time fixing problems.
- LakeMind was an incubating company at EPFL in LABOS and NAL, supported by an Innovation Grant and Venture Kick.
- I designed and implemented the software stack for LakeMind consisting of an event processing pipeline, distributed in-memory dependency graph data structures, and several signal processing and machine learning algorithms. My responsibilities also involved coordinating the software development efforts involving several interns and business development in Europe and the US.
- Cloud service troubleshooting was too early to gain traction in its market segment and we did not attract the venture capital required to develop a minimum viable product. Recently, this market segment has seen significant activity.
- The two co-founders retain full ownership of the Intellectual Property and software developed.

Supervision of Students

Master Semester Projects

MARIO BUCEV *Spring 2019*

- Ph.D. Supervisor. Professor: Willy Zwaenepoel.
- Project: Graph Ingestion Engine for Evolving Graphs.

DIEGO ANTOGNINI *Spring 2016*

- Ph.D. Supervisor. Professor: Willy Zwaenepoel.
- Project: Scalable Decentralized Storage System Design.

VLAD HAPRIAN *Spring 2016*

- Ph.D. Supervisor. Professor: Willy Zwaenepoel.
- Project: Load Balancing Techniques for Chaos.

Summer Internship Project

JUNYAO ZHAO *Summer 2015*

- Supervisor. Professor: Willy Zwaenepoel.
- Project: HDFS Support for X-Stream.

Teaching Activities

EPFL, Real-time Systems (CS-321) *Lausanne, Switzerland*

GUEST LECTURER *Fall 2019*

- Undergraduate level, ~ 50 students. Taught in French.
- Gave a 1-hour lecture on ContikiOS and protothreads.

EPFL, Operating Systems Introduction (CS-323) *Lausanne, Switzerland*

GUEST LECTURER *Spring 2018, Spring 2019*

- Undergraduate level, ~ 100 students. Taught in English.
- Gave a 2-hour lecture on Virtual Machines.

EPFL, Real-time Systems (CS-321) *Lausanne, Switzerland*

TEACHING ASSISTANT *Fall 2019*

- Undergraduate level, ~ 50 students. Taught in French.
- Head Teaching Assistant.
- Designed and led lab sessions.

EPFL, Operating Systems Introduction and Implementation (CS-323 & CS-323a) *Lausanne, Switzerland*

TEACHING ASSISTANT *Spring 2017, Spring 2018, Spring 2019*

- Undergraduate level, ~ 100 students in Introduction class, ~ 30 students in Implementation class. Taught in English.
- Part of a team of 4 Ph.D. Teaching Assistants.
- Led exercise and answers sessions, graded exams, and held office hours in the Introduction class.
- Designed and graded 4 mini-projects in the Implementation class.

EPFL, Calculus III (MATH-203) *Lausanne, Switzerland*

TEACHING ASSISTANT *Fall 2018*

- Undergraduate level, ~ 300 students. Taught in French.
- Part of a team of 4 Ph.D. Teaching Assistants coordinating ~ 20 undergraduate TAs.
- Led exercise and answers sessions, graded exams, and held office hours.

EPFL, Information, Computation, Communication (CS-119) *Lausanne, Switzerland*

TEACHING ASSISTANT *Fall 2016, Fall 2017*

- Undergraduate level, ~ 200 students. Taught in French.
- Part of a team of 4 Ph.D. Teaching Assistants coordinating ~ 20 undergraduate TAs.
- Designed and graded exams, led exercise and answers sessions, held office hours.

EPFL, System Oriented Programming (CS-207) *Lausanne, Switzerland*

Teaching Assistant *Spring 2016*

- Undergraduate level, ~ 150 students. Taught in French.
- Head Teaching Assistant coordinating 8 undergraduate TAs.
- Designed and graded exercises, projects, and exams.

EPFL, Various Undergraduate Classes *Lausanne, Switzerland*

Student Teaching Assistant *Fall 2009 - Spring 2011*

- Information Technology (French), Systems Programming (English), and Software Engineering (English).
- Designed and graded exercises. Answered questions during exercise sessions.

Professional Service

Jul. 2020	**External Reviewer**, USENIX Annual Technical Conference	*Boston, USA*
Apr. 2020	**Organizer and PC Chair**, 1st European Workshop on Graph Processing Systems	*Heraklion, Greece*
Feb. 2018	**External Reviewer**, Principles and Practice of Parallel Programming	*Vienna, Austria*

Prizes, Awards, Fellowships

Dec. 2017	**Teaching Assistant Award**, EPFL IC Faculty
Dec. 2017	**Public Prize Winner**, Exposure Science Movie Hackathon
Sep. 2015	**Ph.D. Fellowship**, EPFL EDIC
Jan. 2012	**Venture Kick 1st Round**, LakeMind

Talks

Mar. 2020	**Hailstorm**, Disaggregated Compute and Storage for Distributed LSM-based Databases	*ASPLOS'20*
Oct. 2019	**Tesseract**, Fast, Scalable Graph Pattern Mining on Evolving Graphs	*U. Toronto*
May. 2019	**Tesseract**, Fast, Scalable Graph Pattern Mining on Evolving Graphs	*EcoCloud'19*
Jun. 2018	**Rock You Like a Hurricane**, Taming Skew in Large Scale Analytics	*EcoCloud'18*
Apr. 2018	**Rock You Like a Hurricane**, Taming Skew in Large Scale Analytics	*EuroSys'18*
Oct. 2015	**Chaos**, Scale-out Graph Processing from Secondary Storage	*SOSP'15*
Jan. 2012	**Track Me If You Can**, on the Effectiveness of Context-based Identifier Changes in Mobile Networks	*NDSS'12*
Sep. 2011	**ARLO**, Making Mobile Augmented Reality a Reality	*MobileHCI'11*

Language Skills

French	**Fluent**, Mother tongue
English	**Fluent (C2)**, Cambridge Certificate of Proficiency
German	**Basic Knowledge (B2)**, Zertifikat Deutsch

Personal Skills / Hobbies

Improvisional Theatre

Founding Member of L'Improsture, an improv team active in French-speaking Switzerland

Chess

Junior chess player in the Geneva chess club. ELO-rated. Hobbyist to this day.

Piano

Classical and pop player.