# Dynamic Resource Provisioning for Data Center Workloads with Data Constraints

BY

SHEN LI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Tarek F. Abdelzaher, Chair
Professor Indranil Gupta
Dr. Jie Liu, Microsoft Research
Professor Lui Sha

# Abstract

Dynamic resource provisioning, as an important data center software building block, helps to achieve high resource usage efficiency, leading to enormous monetary benefits. Most existing work for data center dynamic provisioning target on stateless servers, where any request can be routed to any server. However, the assumption of stateless behaviors no longer holds for subsystems that subject to data constraints, as a request may depend on a certain dataset stored on a small subset of servers. Routing a request to a server without the required dataset violates data locality or data availability properties, which may negatively impact on the response times.

To solve this problem, this thesis provides an unified framework consisting of two main steps: 1) determining the proper amount of resources to serve the workload by analyzing the schedulability utilization bound; 2) avoiding transition penalties during cluster resizing operations by deliberately design data distribution policies. We apply this framework to both storage and computing subsystems, where the former includes distributed file systems, databases, memory caches, and the latter refers to systems such as Hadoop, Spark, and Storm. Proposed solutions are implemented into MemCached, HBase/HDFS, and Spark, and evaluated using various datasets, including Wikipedia, NYC taxi trace, Twitter traces, etc.

# Acknowledgments

My Ph.D. journey was a joyful one. I would like to sincerely express my deepest gratitude to my advisor, Professor Tarek F. Abdelzaher, for his invaluable support. This thesis would not even be possible without him. It was his encouraging guidance thats lead me through the challenges I encountered. His passion and vision in research motivates me to pursue a career in the research community. I learnt a whole lot from him in every aspect, and will always look up to him as my lifetime role model.

I would like to cordially thank my thesis committee members, Professor Indranil Gupta, Dr. Jie Liu, and Professor Lui Sha. It is my honor to have chances to work with great scholars and researchers. Their constructive suggestions and comments helped me significantly improve my thesis.

I would like to extend my gratitude to the professors, researchers, and colleagues I collaborated with when working on my thesis. In particular, I would like to thank Professor Maria Kihl, Professor Anders Robertsson, Professor Marco Caccamo, Professor Gul Agha, Professor Lu Su, Richard Pace, Yong Yang, Shaohan Hu, Shiguang Wang, Yunlong Gao, Hengchang Liu, Md Tanvir Amin, Hongwei Wang, Yiran Zhao, Shuochao Yao, Huajie Shao, Chris Xiao Cai for valuable discussions and suggestions.

I am fortunate to have chances to work at IBM Research, Yahoo, and Facebook during my Ph.D. journey. Thus, I want to express my special thank to my mentors and managers including Raghu Ganti, Mudhakar Srivatsa, Richard Pace, and Lovro Puzar. I harvested a

whole lot of experiences working on real-world research projects and engineering products.

Also, I would like to thank various funding agencies and institutions, such as National Science Foundation, Army Research Laboratory, and IBM Research, for their financial support and assistance.

Lastly, and most importantly, I wish to thank my father Kunyu Li, my mother Wei Pan, and my wife Xiaoxiao Wang, for their continuous love, understanding, and support. They are with me all these years, backing me up through difficulties and sharing my happiness. To them I dedicate this thesis.

# Table of Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1: Introduction

Data centers have been serving as the foundation for the prospering Internet era for more than a decade. Explorations of digital data and applications drive data centers to rapidly grow in both its scale and population. Due to data centers' enormous monetary value, both industry and academia vigorously seek solutions to improve data center cost efficiency by increasing the amount of served workloads or reducing the operational expenses. Advances on both directions may benefit from resource proportionality which requires the data center to consume a proportional amount of resources (*e.g.*, power, number of servers, etc.) in accordance with the amount of workloads. Under shared circumstances, resource proportionality helps to pack a larger amount of heterogeneous workloads into a single data center in a staggering and opportunistic fashion [1–5], generating higher revenue. In dedicated clusters that serve a fluctuating amount of workloads, resource proportionality saves energy by allowing the cluster to adaptively power off a subset of servers at non-peak load [6–10], considerably reducing the operational expenses.

Data centers are complex systems composed of multiple subsystems, there is no single solution to arm all subsystems with the ability of resource proportionality. However, various solutions can be summarized into a common framework as shown in Figure 1.1. In this framework, the *system* is the target to enable dynamic provisioning. The *sensor* component monitors key performance metrics (*e.g.*, response time, CPU utilization, Free disk space, etc.) of the system, and reports system status to the policy. The scaling *policy* calculates the resource demand based on the given workload, and the scaling *manager* gracefully increases or decreases the amount of resources by taking care of transition penalties.

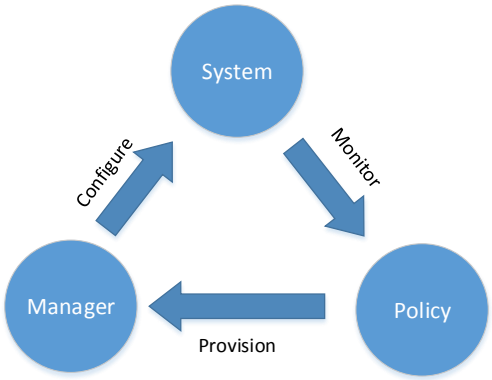



**Figure 1.1:** Dynamic Resource Provisioning Framework

Plenty of work have proposed dynamic resource provisioning solutions for stateless web servers [11–13], where any request can be routed to any web server. However, the assumption of stateless behaviors no longer holds for subsystems that subject to data constraints. We coarsely categorize those subsystems into two classes, storage and computing. The former includes distributed file systems, databases, memory caches, and the latter refers to systems such as Hadoop [14, 15], Spark [16–18], and Storm [19]. In these systems, properties such as data locality and availability significantly impact response times. Due to the huge disparities between these systems, each of them calls for specifically tailored solutions to achieve resource proportionality. As summarized in Table 1.1, chapters of this thesis design policies and managers for the storage and computing systems respectively. Below, we elaborate the characteristics and design requirements of storage and computing systems for the policy and manager components respectively.

| | Storage | | computing |
|---|---|---|---|
| | Randomly Partitioned | Ordered Partitioned | |
| Policy | Solved in existing solutions | | Section 3: Packing Servers |
| Manager | Section 4: Proteus | Section 5: Pyro | Section 6: Stark |

**Table 1.1:** Categorization and Sections Summary

## 1.1 Policy: Make Provisioning Decisions

Effective resource provisioning is the cornerstone for delivering resource proportionality. The policy collects monitoring information of the entire cluster, and makes estimations on what is the right amount of resources to reach given Quality of Services (QoS) objectives. Underestimation would overload the service cluster, leading to unacceptably long response times, while overestimation would under-utilize the cluster, unnecessarily wasting valuable resources. As the amount of workloads fluctuates over time, the resource provisioning solution should accommodate these dynamics adaptively. The decisions could be scaling CPU frequencies, workload scheduling constraints, power on/off servers, etc.

In systems enforcing response times as QoS objectives, one solution would be to monitor the system response time and increase (decrease) the number of running servers accordingly when the response time rises (drops). Though effective for storage systems that answer IO requests within milliseconds, this solution is inapplicable to distributed computing systems such as Hadoop and Spark. The reason is that the response times of jobs in these systems range from milliseconds to hours or even days, whose sheer value is inadequate to faithfully indicate cluster resource utilization. Hence, the solutions for the storage and the computing systems differ to a large degree. Therefore, policies need to be designed for the two different types of systems accordingly.

### 1.1.1 Distributed Storage Systems

Data center distributed storage systems usually aim at serving IO requests with low response times. Though distributed storage systems cover a wide spectrum of applications, such as memory cache, distributed file systems, NOSQL database, the request complexities within the same application vary within a small range. For example, getting a key-value pair from memcached [20] takes a few millisecond, while reading an HBase [21, 22] block (64KB by default) takes tens of milliseconds. The low latency and high throughput nature allows the system to actively monitor system status that positively relates to the response time (*e.g.*,

number of concurrent requests, queue length, etc.), and tune the cluster capacity accordingly in real time. Many existing solutions [11–13] have already solved this problem. So, this thesis skips exploring this area.

### 1.1.2 Distributed computing Systems

As mentioned before, jobs in the same distributed computing system may diverse to a large extend. Some of them may finish within hundreds of milliseconds, while others may last a few hours or even days. Therefore, this information can neither be collected in real time nor faithfully represent the cluster utilization. Moreover, as some jobs may occupy a huge amount of resources for a relatively long period of time, it is preferred to estimate the amount of required resources before executing the job, so that the system may provision resources proactively. This thesis tackles this problem in Section 3 by introducing the theory of *Packing Servers* that bridges the scheduling problem in distributed computing systems to the abundance set of theory advances in the Real-time community.

## 1.2 Manager: Handle Resizing Penalties

Apart from stateless web servers, resizing storage and computing systems subjects to considerable performance penalties. The reason is that distributed storage and computing systems usually closely bind with data, where resizing the cluster could destroy amiable properties such as data availability and data locality, negatively impacting the site performance.

### 1.2.1 Distributed Storage Systems

In distributed storage systems, data is associated with identifiers, where the identifier can be the path to a file, the ID of a block, a key for a key-value pair, the primary key for a row in a database, etc. Based on how the system distributes data into servers, they can be categorized into randomly partitioned and ordered partitioned data storage systems. The former randomly distributes data into servers. The latter divides the data into ranges based

on their identifiers such that all identifier in the same range are hosted by the same server. Disparities in key distribution policies lead to different types of performance penalties.

**Randomly Partitioned Data Storage**

Randomly partitioned data distribution policies work well for applications operating on individual data pieces. In this case, the key distribution policy directly affects cluster load balancing. Under static scenarios, one could achieve load balancing by randomly mapping data identifiers to a large integer range using a consistent random function, and then applying the modulo operator to wrap it to a valid cache server ID. However, this solution makes it painful when resizing a cluster. For example, in an $n$-server cache cluster, if one more server is added, this simple solution re-maps $\frac{n}{n+1}$ identifiers into different servers. Consistent hashing alleviates the problem to some degree. Nevertheless, how to balance load under dynamics and how to make dynamic server provisioning smooth enough still remain unsolved. Section 4 presents Proteus which creates an uneven amount of virtual nodes for physical servers, and optimally places those virtual nodes on the consistent hashing ring to balance loads.

**Ordered Partitioned Data Storage**

In ordered partitioned storage systems, each server handles a continuous range of identifiers. This design helps to reduce the overhead of range queries by avoiding sending every query to all servers in the cluster. The initial range boundaries are usually derived from historical data or estimations on data distributions. During run time, systems resize the cluster by splitting and merging ranges. Nevertheless, splitting and merging ranges destroy the data locality benefits, leading to larger response times. The Pyro system described in Section 5 tackles this problem by making use of data replications in lower layer distributed file systems.

### 1.2.2 Distributed Computing Systems

Unlike in storage systems where IO requests query for specific data items from a single dataset, computing system tasks often need to work on multiple datasets to perform opera-

tions such as join and cogroup. Therefore, besides data locality properties, maintaining data co-locality also significantly benefits the performance. In this case, splitting and merging operations affects multiple co-located datasets. This thesis discussed this problem using the in-memory computing system Spark as the context, and proposed an evolved system named Stark in Section 6 that solves the aforementioned problems.

# Chapter 2: Principles and Contributions

In this thesis, we apply three principles to achieve dynamic resource provisioning in data center:

- *Reducing Parallelism*: High parallelism is generally considered as a merit in distributed systems that allows job executions to embrace resources from a cluster of servers. While increasing parallelism usually helps to accelerate job execution, it may on the other hand drive job makespans to become unpredictable, due to resources contentions among multiple jobs. This unpredictability hinders the policy module from estimating the proper amount of resources to satisfy job deadlines. Therefore, in Chapter 3, we explore the trade-off between individual job parallelism and job makespan predictability.

  Our model considers job graphs with deadlines, where edges in the graph represent the precedence constraints among jobs. We develop the theory of packing server that limits the execution of each job into the minimum parallelism without violating the deadline. The reduced parallelism allows the scheduler to treat job graphs as independent jobs with bounded utilization penalty, and derives schedulability utilization bounds that beat best known results.

- *Grouping Replicas*: In distributed storage systems, data blocks are usually replicated and randomly distributed to achieve fault tolerance. Besides this original and common usage, we claim that data replicas may also help to deliver elasticity if grouped properly. Grouping replicas means that instead of randomly distributing data replicas into servers, replicas are organized into groups, such that all data within the same group

are stored in the same server. This design creates opportunities for higher layer applications to move computations to data when the computations require multiple data pieces, avoiding the overhead of moving data around. In Chapters 5 and 6, we discuss this principle in detail using HBase/HDFS stack and Spark as examples.

Chapter 5 presents the Pyro system developed based on the HBase/HDFS stack. Pyro organizes HDFS block replicas into groups, which helps the higher layer HBase to split/merge regions without losing the data locality benefits. Chapter 6 describes the Stark systems built based on Spark. Stark organizes RDD partitions into groups, which allows immutable RDDs to alter partitioning boundaries without re-construct the entire dataset.

- *Extending Hashes*: Hash functions are widely used in data center subsystems to efficiently and evenly allocate both computation and data (keys) into servers (hashes). To cope with the fluctuating volume of workload, hash functions need to be able to extend/shrink the base of their hashes (*i.e.*, turn on/off servers) based on the amount of received workload. This extendability can help the Manager module minimize the penalty when resizing the cluster. Chapters 4 and 6 evaluate two solutions that organize the hashes into a ring structure and a tree structure respectively.

  Chapter 4 introduces the Proteus memory cache system that tunes the virtual node placement policy on consistent hashing rings to guarantee that workloads are evenly distributed into all running servers during join and leave events. The Stark system presented in Chapter 6 organizes RDD partition hashes into a tree structure, such that leaf nodes represent partitions and internal nodes represent groups. In Stark, each task process all data of a group, where a group may split into two children or merge with its sibling into a parent group. The ability of extending/shrinking groups helps to avoid excessively large task input dataset induced by the inconsistency between partitioning scheme and dynamic input data distribution, keeping job makespans within a reasonable range.

We introduce and evaluate the above three principles into Hadoop, HBase, MemCached, and Spark, resulting in several novel software systems with improved dynamic resource provisioning performance. The contributions of this thesis can be summarized as follows:

- This thesis introduces the *generalized packing server*, which allows deriving schedulability utilization bound for precedence-constrained tasks using existing techniques for independent tasks. The utilization bound helps to efficiently determine the minimum amount of resources to meet deadlines of all tasks, which can be used as a policy module to make provisioning decisions.

- We propose and develop the first power proportional memory cache cluster by exploring the virtual node placement schemes in the consistent hashing ring, which guarantees minimum volume of data transferring during cluster resizing events.

- This thesis applies the same extendable hashing mechanism to both storage systems and computing systems, which allows data distribution mechanisms to adapt to data volume and request intensity dynamics, preserving data locality properties in the best effort manner.

- This thesis develops Pyro as ordered-partitioned spatial-temporal data storage system by introducing the extendable hashing mechanism to HBase/HDFS software stack. It may split and redistribute a dataset multiple times without moving data around, delivering both elasticity and data locality.

- This thesis also designs and implements Stark based on Spark, as an optimized in-memory batch-processing system for dynamic dataset collections. We inject the extendable hashing mechanism to Spark's data partitioning and caching modules, granting immutable RDDs the ability to react to data and request dynamics.

# Chapter 3: Policy For Distributed Computing Clusters

This chapter introduces the *generalized packing server*, which allows converting generalized parallel tasks into a corresponding set of independent tasks for the underlying scheduler. Hence, any schedulability results, previously derived for independent tasks on multiprocessors become applicable to analyze the schedulability of generalized parallel tasks.

The generalized parallel task model was studied extensively in recent literature [23–30]. It refers to tasks that are described by workflow graphs, where some subtasks can execute in parallel subject to precedence constraints. The new server packs computation time of these tasks into budgets, while respecting precedence constraints. The resulting budgets, however, can be treated by the underlying scheduler as *independent tasks*. To ensure that the packing server can always successfully pack the original task set into budgets, we show that the sum of the budgets should be larger by a certain factor than the sum of the original computation times. As a result, a conversion factor is derived between schedulability bounds derived in prior literature for independent tasks on multiprocessors (that therefore apply to the budgets) and the corresponding bounds that apply to the schedulability of generalized parallel tasks. The new bounds derived using the aforementioned conversion approach for the generalized parallel tasks are shown to be better in many cases than the corresponding best-known bounds for this task model.

The generalized packing server, described in this chapter, can use *any work-conserving scheduling policy* to pack tasks into server budgets. Hence, results derived for this server are

broadly applicable.

## 3.1 Task Model and Basic Concepts

Recent work introduced the generalized parallel task model, drawing an increasing amount of attention to its expressiveness and broad applications [26–29]. The chapter addresses the schedulability of generalized parallel tasks on multiprocessors, composed of a number of identical processing elements (or cores). We first review the task model, then formalize the notions of packing servers and conversion bounds, and finally outline the main result of the chapter.

### 3.1.1 The Generalized Parallel Task Model

We adopt the generalized parallel task model, proposed by Baruah et al. [31]. In this model, each task, $\tau_i$, is represented by a Directed Acyclic Graph (DAG), denoted by $G(N_i, E_i)$, where the set of nodes, $N_i$, represents a set of jobs comprising a single instance of the task, and the set of edges, $E_i$, represents their precedence constraints. Let the number of jobs $|N_i|$, in $\tau_i$, be denoted by $n_i$, and let the total number of recurrent tasks be denoted by $n$. The worst case execution time (WCET) of the $j^{th}$ job is $c_i^j$. A job can be preempted and then resumed on the same or a different core. The $j^{th}$ job of an instance of $\tau_i$ can only start after all its predecessor jobs on the DAG have finished. Every $T_i$ time units, a new instance of task $\tau_i$ is released with relative deadline $D_i$. In this chapter, we assume $D_i \leq T_i$. The utilization $u_i$ of task $\tau_i$ is thus the total amount of computation over its deadline (*i.e.*, $\frac{\sum_{j \in N_i} c_i^j}{T_i}$). The set of recurrent generalized parallel tasks, denoted by $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, runs on a platform of $m$ cores.

Let $L_i$ denote the critical path length of task $\tau_i$. The critical path is the longest execution path in the DAG. Hence, $L_i = \max_{path_k \in DAG} \sum_{j \in path_k} c_i^j$. We define the stretch of a generalized parallel task, $\tau_i$, denoted by $\varphi_i$, as the ratio of its relative deadline $D_i$ over its critical path length $L_i$. Let $\varphi$ denote the minimum stretch of all tasks (*i.e.*, $\varphi = \min\{\varphi_1, \varphi_2, ..., \varphi_n\}$).

| Notation | Description |
|---|---|
| $n$, $m$ | number of tasks and number of cores |
| $\tau_i$ | task graph $i$ |
| $N_i$ | job set of $\tau_i$ |
| $D_i$ | deadline of $\tau_i$ |
| $L_i$ | critical path length of $\tau_i$ |
| $u_i$ | utilization of $\tau_i$ |
| $\overline{u}_i$ | total utilization of budgets converted from $\tau_i$ |
| $\varphi_i$ | the ratio of $D_i$ over $L_i$ (stretch) |
| $\beta$ | reverse of max individual budget utilization |
| $c_i^j$ | WCET of job $j$ in $\tau_i$ |
| $D_i'$ | $D_i$ over $\beta$ |
| $x_i$ | the number of budgets for task graph $\tau_i$ |
| $l_i$ | the size of individual budget for task graph $\tau_i$ |

**Table 3.1:** Packing Server Notation

### 3.1.2 Assumptions

As the first approach to pack precedence-constrained tasks to independent tasks, this chapter makes three assumptions to simplify the problem:

- Tasks can be preempted and resumed with no penalties.
- The WCET of each individual task can be estimated a priori to its execution.
- Data locality does not affect task execution times.

These assumptions help us to derive neat theorems with concise proofs. Future works of relaxing these assumptions will be presented in the discussion section.

### 3.1.3 Packing Servers

A packing server is a construct that allows the underlying scheduler to ignore precedence constraints among jobs in the above model and focus on scheduling server budgets as independent entities. Each task, $\tau_i$, in the above model, is assigned its own server that is characterized by two parameters; a budget size, $l_i$, and a concurrency level, $x_i$. The server appears to the underlying scheduler as a set of $x_i$ independent tasks, each of computation

time, $l_i$. These tasks (i.e., server budgets) can be scheduled independently on the multiprocessor (on the same or different cores) according to the underlying multiprocessor scheduling policy. When the underlying scheduler invokes any of the server budgets, the server looks for jobs of task, $\tau_i$, whose predecessors are finished and runs them within its budget(s). When a budget expires, the underlying scheduler stops executing it. This chapter shows that if $x_i$ and $l_i$ are appropriately chosen, the packing server always manages to fit all jobs of the original task, $\tau_i$, within its budgets, while respecting precedence constraints, regardless of how the budgets are scheduled by the underlying scheduler. Hence, as long as the budgets finish before the deadline of $\tau_i$, the task is schedulable. Note that, analyzing schedulability of server budgets is an independent task scheduling/schedulability analysis problem. Thus, the server allows converting the original problem of scheduling generalized parallel tasks on multiprocessors to the problem of scheduling independent tasks (the server budgets).

An analogy can be made between packing servers and servers for aperiodic tasks. Since aperiodic tasks were harder to analyze than periodic ones, real-time scheduling literature introduced the idea of aperiodic task servers [32–34], such that aperiodic tasks can be scheduled inside these servers, while the servers themselves appear to be periodic tasks to the underlying scheduler. Scheduling and analysis techniques for periodic tasks could then be extended to aperiodic task servers.

By analogy, the generalized packing server is introduced because scheduling tasks with precedence constraints is more difficult than scheduling independent tasks. The server allows scheduling and analysis techniques for independent tasks to be applied to generalized parallel tasks.

### 3.1.4 Conversion Bounds

The “catch" in converting generalized parallel tasks to independent server budgets is that, as we show in this chapter, the sum of server budgets assigned to task $\tau_i$ needs to be appropriately larger, in total, than the sum of execution times of the jobs in $\tau_i$. If this condition is met, then we show that any work-conserving scheduling policy will *always succeed* at fit-

ting all jobs of a task within the corresponding server budgets, while respecting precedence constraints, *regardless* of how these budgets are scheduled by the underlying multiprocessor scheduler.

We call the ratio of the sum of execution times of jobs in $\tau_i$ to the sum of resulting budgets, the *conversion bound*. The lower the bound, the less efficient the packing server. This chapter introduces the simplest packing server and its conversion bound. Possible improvements that may lead to better bounds are mentioned in the discussion section.

### 3.1.5 The Main Result

We show that, for the generalized packing server introduced in this chapter, the conversion bound is equal to $\frac{\varphi-\beta}{\varphi}$, where $\varphi$ is as defined earlier in Section 3.1.1 and $\beta$ is a tunable parameter that controls the upper bound of the maximum individual server budget utilization $u_{\max} = l_i/T_i$, such that $u_{\max} \leq \frac{1}{\beta}$. We prove that if the utilization bound for scheduling of independent tasks by the underlying multiprocessor scheduling policy, is $U_B$, then the generalized parallel task set is schedulable when its utilization does not exceed $U_B \cdot \frac{\varphi-\beta}{\varphi}$. The observation allows us to map bounds for independent tasks to those for generalized parallel tasks, essentially discovering new schedulability bounds for the latter in several cases. As mentioned in the introduction, the chapter generalizes our prior work [35] which derived a similar result for a special-case packing server that required a clairvoyant underlying scheduler.

## 3.2 Generalized Packing Server

Figure 3.1 illustrates the high-level idea of the generalized packing server. The user must create a server for every generalized parallel task. The underlying scheduler runs some independent task scheduling algorithm, such as Global EDF (GEDF) [36] or EDF First Fit (EDF-FF) [37]. When the underlying scheduler executes a budget, much like the case with aperiodic task servers, the corresponding packing server executes jobs of the original

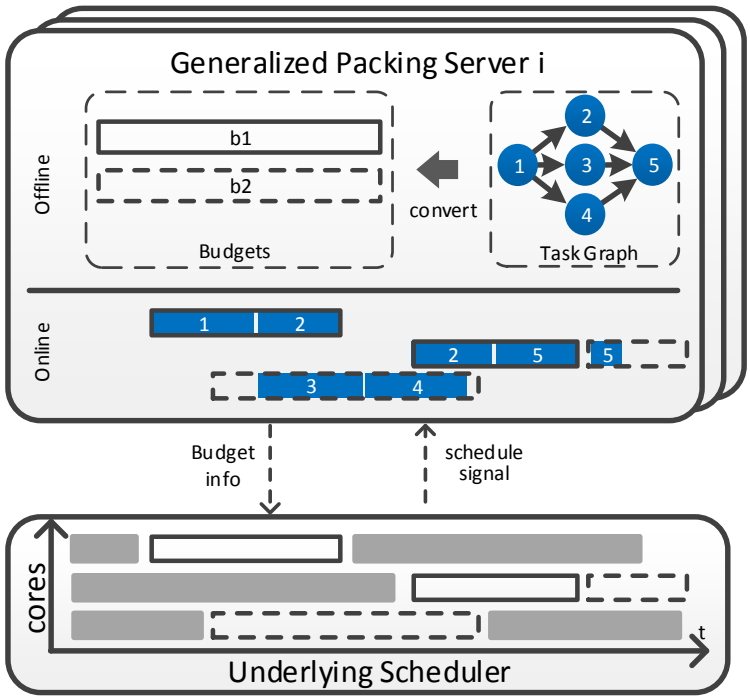



**Figure 3.1:** Packing Server Overview

workflow task within that budget using *any* work-conserving scheduling policy. The policy respects precedence constraints by assigning jobs whose predecessors are finished to the running budgets. We show that this scheduler always succeeds at fitting all jobs into the budgets, as long as the budgets are sized as described in this chapter.

When a server budget of task $\tau_i$ is invoked, if no job of $\tau_i$ is ready to execute in it (which can happen if the number of budgets invoked concurrently exceeds the level of concurrency of $\tau_i$), the budget remains unutilized. We say that the budget, in this case, runs *idle*. In this chapter, we analyze a very simple version of the packing server, where budgets that run idle continue to occupy the CPU (i.e., the budget busy-waits) and continue to be decremented, just as if they were executing jobs of $\tau_i$. When a budget is depleted, it stops running. Should a job become ready at any point prior to budget depletion, it can use the leftover. If a job finishes before the underlying budget depletes, the server picks another ready job to feed into that budget.

Section 3.2.1 discuss how to set server parameters such that any work-conserving sched-

uler in the server can always finish the input task graph inside the budgets. Section 3.2.2 derives the resulting conversion bound achieved by the generalized packing server. Notations are introduced near their first usage, and are also summarized in Table 3.1.

### 3.2.1 Deriving Server Parameters

Many existing utilization bounds for schedulability of independent tasks are a function of granularity of individual tasks, often measured by the maximum individual task utilization. When individual tasks have a lower maximum utilization, a higher utilization bound can be derived for schedulability of various scheduling algorithms (*e.g.*, EDF-FF [37] and GEDF [29]). Translating to the context of server budgets, the above observation suggests that it is preferable to have a smaller budget size, $l_i$. Therefore, we introduce a parameter $\beta$ to restrict budget size by imposing the constraint: $l_i \leq \min\{D_i, \frac{T_i}{\beta}\}$. The constraint bounds the maximum budget utilization at $\frac{1}{\beta}$. Later we shall determine the optimal value of $\beta$. Hence, it is useful to define a modified deadline, $D'_i$ as:

$$D'_i = \min\{D_i, \frac{T_i}{\beta}\} \leq \frac{D_i}{\beta}. \tag{3.1}$$

The following theorem states how to set the budget parameters of a packing server such that any work-conserving scheduling policy, executed by the server, will always succeed to pack the original task graphs within the available budgets, no matter how these budgets are invoked by the underlying scheduler.

**Theorem 1.** *Any work-conserving application scheduler can always fit $\tau_i$ into its $x_i$ identical budgets of size $l_i$, regardless of how the underlying scheduler schedules those budgets, if:*

$$x_i = \left\lceil \frac{\sum_{j \in N_i} c_i^j - L_i}{D'_i - L_i} \right\rceil \tag{3.2}$$

$$l_i = L_i + \frac{1}{x_i} \left( \sum_{j \in N_i} c_i^j - L_i \right). \tag{3.3}$$

***Proof.*** To prove the theorem, consider an execution schedule that maps jobs of task $\tau_i$ onto $x_i$ budgets of length $l_i$ each. We do not make any assumptions on when individual budgets are executed by the underlying scheduler. We merely assume that whenever a budget becomes available, the server assigns jobs of task $\tau_i$ to it, in a work-conserving fashion, subject to their precedence constraints. Let $t_{start}$ be the first time that a job of $\tau_i$ started executing in a budget, and let $t_{finish}$ be the time that $\tau_i$ finished. Let us compute the maximum amount of idle server budgets accrued between $t_{start}$ and $t_{finish}$. Recall that a budget is said to be "running idle" if it is invoked by the underlying scheduler but not used by the packing server (to run a job of $\tau_i$). In our implementation, such a budget simply busy-waits. This happens if the underlying scheduler invokes more budgets than the number of jobs of $\tau_i$ that are currently eligible (i.e., whose predecessors finished). Let us divide the time $[t_{start}, t_{finish}]$ into three types of intervals:

- (A) Intervals where no budgets belonging to $\tau_i$ are running.
- (B) Intervals where budgets are running and at least one budget is idle. Let the sum of those intervals be denoted by $I_1$.
- (C) Intervals where budgets are running and no budget is idle.

Note that, idle budgets exist only during intervals of type (B) above. The question is: what is the maximum amount of idle budget execution that may be incurred?

To answer this question observe that when a budget is idle, it must be that the number of available budgets exceeds the number of eligible jobs of $\tau_i$ (i.e., those jobs whose predecessors in the task graph are finished). Accordingly, *all* eligible jobs of $\tau_i$ must be running at that time. This condition is indistinguishable from running $\tau_i$ on a machine of infinite parallelism (one that has infinite cores). Thus, the total time interval for which this condition can hold (i.e., the sum $I_1$) cannot be longer than $L_i$ (the length of the critical path of $\tau_i$). This is true because $\tau_i$ would always finish execution on a machine of infinite parallelism in time $L_i$ or less.

Clearly, in our schedule, intervals where a budget is idle (i.e., intervals of type (B)) are interleaved with intervals of type (A) and (C). However, these other intervals do not increase the workload and do not change the graph. Hence, the sum $I_1$ remains bounded by what would be enough for a machine with infinite parallelism to finish the original task graph.

In summary, the total duration of time during which at least one budget is idle cannot exceed $L_i$. Also, since the schedule used by the packing server is work conserving, at least one budget must be busy as long as the task has not terminated. Hence, the number of simultaneously idle budgets cannot exceed $x_i - 1$. The maximum total idle budget time is thus:

$$(x_i - 1)\, L_i. \tag{3.4}$$

Therefore, by the time task $\tau_i$ terminates, it would have consumed budgets for a total duration $\sum_{j \in N_i} c_i^j$ (to run all its jobs), and a total budget time of no more than $(x_i - 1)L_i$ would have been left idle. This adds up to the following maximum total needed budget time:

$$x_i l_i = \sum_{j \in N_i} c_i^j + (x_i - 1)\, L_i. \tag{3.5}$$

By dividing both sides with $x_i$, we arrive at the $l_i$ formula shown in Equation (3.3).

As suggested by Equation (3.5), there are multiple pairs of valid $x_i$ and $l_i$ combinations. However, as shown in Equation (3.4), the value of $x_i$ directly affects the amount of utilization penalties introduced during the budget construction. Larger $x_i$ leads to larger penalties. On the other hand, excessively small $x_i$ leads to larger $l_i$, which might violate the deadline $D_i'$ constraints. Given the above two considerations, $x_i$ needs to be set to the smallest value that respects the deadline $D_i'$. Quantitatively, it means that using $x_i$ budgets respects $D_i'$, while using $x_i - 1$ budgets violates $D_i'$. Therefore, we have:

$$L_i + \frac{1}{x_i}\left(\sum_{j \in N_i} c_i^j - L_i\right) \leq D_i' \tag{3.6}$$

$$L_i + \frac{1}{x_i - 1}\left(\sum_{j \in N_i} c_i^j - L_i\right) > D_i' \tag{3.7}$$

Expressing $x_i$ using Equations (3.6) and (3.7) respectively, we have:

$$\frac{\sum_{j \in N_i} c_i^j - L_i}{D_i' - L_i} \leq x_i < \frac{\sum_{j \in N_i} c_i^j - L_i}{D_i' - L_i} + 1. \tag{3.8}$$

As $x_i$ has to be an integer, Equation (3.2) follows. ■

Next, we prove the resulting conversion bound.

### 3.2.2 The Conversion Bound

Task graphs can always fit within their budgets, when the budgets are parameterized as described above. Hence, if the underlying scheduler can schedule all budgets by the original task deadlines, then all tasks will be schedulable. The budget schedulability problem is one of analyzing schedulability of independent tasks, for which utilization bounds have been derived in literature. Let the utilization bound for schedulability independent tasks (by the underlying scheduler) be $U_B$. The utilization of the original task set must be smaller than $U_B$ because the sum of budgets is larger than the sum of the original job execution times. In this section, we use this observation to derive a utilization bound for the original task set, such that the resulting budgets remain under the utilization bound, $U_B$. We prove the following theorem:

**Theorem 2.** *If the underlying scheduler achieves a utilization bound of $U_B$, the generalized packing server guarantees that $U_B \cdot \frac{\varphi - \beta}{\varphi}$ is a valid utilization bound for task graphs.*

***Proof.*** Remember that in computing $x_i$, we found the smallest value that allows meeting

the modified deadline, $D'_i$. Hence, by construction, decreasing budgets to $(x_i - 1)$ will violate the modified deadline $D'_i$. This results in:

$$L_i + \frac{1}{x_i - 1}\left(\sum_{j \in N_i} c_i^j - L_i\right) > D'_i \geq \frac{\varphi}{\beta} \cdot L_i \tag{3.9}$$

By reorganizing Equation (3.9), we have:

$$\begin{aligned} \frac{\sum_{j \in N_i} c_i^j}{x_i - 1} &> \frac{\sum_{j \in N_i} c_i^j - L_i}{x_i - 1} \\ &> \left(\frac{\varphi}{\beta} - 1\right) L_i \end{aligned} \tag{3.10}$$

Then, we can represent the lower bound on the total execution time of the original task graph as:

$$\sum_{j \in N_i} c_i^j \geq (x_i - 1)(\frac{\varphi_i}{\beta} - 1) L_i \tag{3.11}$$

As the amount of idle budget contributed by a task graph is upper bounded by $(x_i - 1)\, L_i$, we can compute the upper bound ratio of idle budget over original task graph total WCET:

$$\begin{aligned} \frac{x_i l_i - u_i}{u_i} &= \frac{(x_i - 1)\, L_i}{\sum_{j \in N_i} c_i^j} \\ &\leq \frac{(x_i - 1) L_i}{(x_i - 1)(\frac{\varphi_i}{\beta} - 1) L_i} \qquad \text{Inequality (3.11)} \\ &= \frac{\beta}{\varphi_i - \beta} \end{aligned} \tag{3.12}$$

Finally we have:

$$\sum_i u_i \geq \frac{\varphi - \beta}{\varphi} \sum_i x_i l_i, \quad \varphi = \min\{\varphi_1, \varphi_2, ..., \varphi_n\} \tag{3.13}$$

Equation (3.13) agrees with the conversion bound claimed in Theorem 2. ■

This conversion bound matches the one derived in literature [35]. However, compared to literature [35], we completely remove the claivoyance requirement on the underlying scheduler, thereby promoting the packing server technique to a broader spectrum of applications.

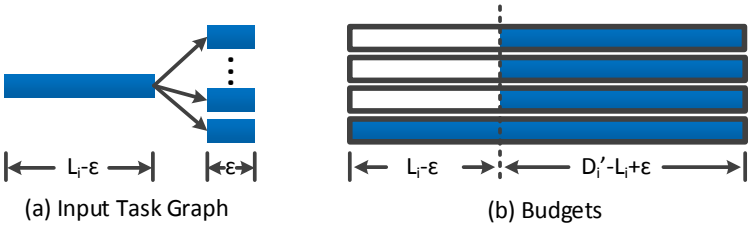



**Figure 3.2:** Bound Tightness

The conversion bound $\frac{\varphi-\beta}{\varphi}$ applies to all work-conserving schedulers used by the packing server, with deadline equals to its period ($D_i = T_i$). One natural question to ask is whether the conversion bound is tight. For example, when proving the conversion bound, we do not make use of proactive preemptions at all, where *Proactive Preemption* is the job preemption issued by the application-level scheduler when the corresponding budget is not preempted by the underlying scheduler. Hence, it is meaningful to know whether the conversion bound can be further improved if the application scheduler is allowed to proactively preempt job executions using some judicious algorithms.

Intuitively, proactive preemptions may help reduce the amount of required idle budget $(x_i - 1)\, L_i$. For example, by enforcing a PFair-like scheduling policy in the application, packing servers no longer need to worry about depleting a job too soon to preserve enough parallelism to occupy all budgets. However, it turns out that the bound is still tight even if the application scheduler uses unlimited proactive preemptions.

Consider a task graph $\tau_i$ as depicted in Figure 3.2 (a). The WCET of the first job is $c_i^1$. The remaining $\frac{\left(D_i'-c_i^1\right)x}{\epsilon}$ jobs share the same WCET $\epsilon$, and are all successors of the first job. After converting task graph $\tau_i$ into $x_i$ budgets, we compute the ratio of the additional computation time introduced as:

$$\begin{aligned}
\frac{x_i l_i - u_i}{u_i} &= \frac{(x-1)c_i^1}{(D_i' - c_i^1)\,x + c_i^1} \\
&\approx \frac{(x-1)L_i}{\left(\frac{\varphi_i}{\beta} - 1\right) L_i x + L_i} \text{ as } L_i = c_i^1 + \epsilon \approx c_i^1 \\
&\approx \frac{\beta}{\varphi_i - \beta}. \qquad \text{when } x \text{ is large}
\end{aligned} \tag{3.14}$$

In this example, proactive preemptions cannot help, as the conversion bound in this case is still $\frac{\varphi-\beta}{\varphi}$. Hence, the bound shown in Equation (3.13) is the optimal conversion bound regardless of how the application scheduler fits jobs into budgets.

### 3.2.3 Computing the Optimal Beta

Using the conversion bound, shown in Equation (3.13), requires one to determine the value of $\beta$ that preferably yields the highest utilization bound. We demonstrate this process by using GEDF-FF [37] and GEDF [29] as two examples.

Generalized packing servers may use any work-conserving independent task scheduling policy in the underlying scheduler. As proven above, they achieve a utilization bound of $U_B \cdot \frac{\varphi-\beta}{\varphi}$ for task graphs, where $U_B$ is the utilization bound of the underlying independent task scheduler. The survey [38] summarizes $U_B$ for both GEDF and EDF-FF. The value of $\beta$ affects the utilization bound in two competing directions: 1) Schedulability bounds for the underlying scheduling algorithms usually favor smaller individual task utilization, and hence larger $\beta$; 2) The conversion bound prefers a smaller $\beta$. Therefore, it is desired to find an optimal $\beta$ that balances these two requirements. Substituting for $U_B$ with the utilization bound for schedulability of GEDF, the optimal $\beta$ maximizes:

$$U_B \cdot \frac{\varphi - \beta}{\varphi} = \frac{(\varphi - \beta)(m\beta - m + 1)}{m\varphi\beta}. \tag{3.15}$$

By taking derivatives with respect to $\beta$, and setting the derivative to 0, the highest utilization

bound for the packing server over GEDF is achieved at:

$$\beta = \sqrt{\frac{\varphi(m-1)}{m}}. \tag{3.16}$$

Similarly, one can show that the highest utilization bound for the packing server over EDF-FF is achieved at:

$$\beta = \sqrt{\frac{(\varphi+1)(m-1)}{m}} - 1. \tag{3.17}$$

The derived value of $\beta$ determines the modified deadline $D_i'$, which in turn affects the budget size as shown in Equation (3.3). Packing servers use the optimal $\beta$ to set $D_i'$ and configure budgets accordingly.

### 3.2.4 Impact on Schedulability Analysis

The generalized packing server can cooperate with various independent task schedulers, which not only improves task graph utilization bounds for some algorithms, but also unlocks task graph utilization bounds for many more algorithms. We compare the utilization bounds achieved with and without generalized packing servers. Table 3.2 presents the utilization bounds ($U_B$) for algorithms summarized in literature [38]. Assume the configurations satisfy high platform parallelism ($m \gg 1$) and large task graph stretch ($D_i \gg L_i$). For simplicity, instead of computing the optimal $\beta$ and presenting task graph utilization bound formulas, we further assume that $\beta$ is set to 5 for packing servers, which allows direct comparisons against existing results. Let $U_K$ and $U_P$ denote the best Known utilization bounds [29] and the bounds achieved by Packing servers respectively.

Table 3.2, shows that packing server significantly improves the best known utilization scheduling bound for G-EDF and G-RM. Moreover, with packing server, schedulers such as EDF-FF, EDF-FFD, RM-ST, and EDZL embrace task graph utilization bounds which were not available before.

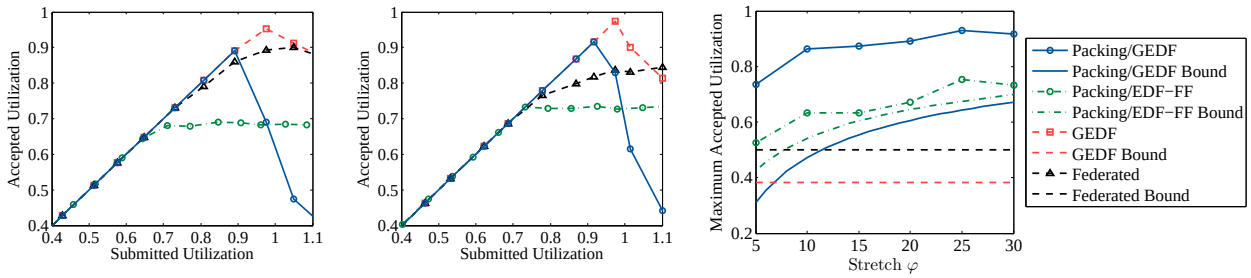
| Task Scheduler | $U_B$ Equation | $U_K$ | $U_P$ |
|---|---|---|---|
| G-EDF [29] | $\frac{m\left(1-\frac{1}{\beta}\right)+\frac{1}{\beta}}{m}$ | 38.2% | 80% |
| G-RM [29] | $\frac{\frac{m}{2}\left(1-\frac{1}{\beta}\right)+\frac{1}{\beta}}{m}$ | 25.8% | 40% |
| Federated [29] | NA | 50% | NA |
| EDF-FF [39] | $\frac{\frac{m\beta+1}{\beta+1}}{m}$ | Unknown | 80% |
| EDF-FFD [40] | $\frac{m-\frac{m-1}{\beta}}{m}$ | Unknown | 80% |
| RM-ST [41] | $\frac{(m-2)\left(1-\frac{1}{\beta}\right)+1-\ln 2}{m}$ | Unknown | 80% |
| EDZL [42] | $1-\frac{1}{e}$ | Unknown | 63% |

**Table 3.2:** Examples of Improved Bounds ($m, \varphi \gg 1$, $\beta = 5$)

## 3.3 Evaluation

This section compares existing task graph schedulers, namely GEDF and Federated, to packing servers using GEDF (independent tasks) and EDF-FF as the underlying schedulers. Moreover, we evaluate how the stretch $\varphi$, the reverse of maximum individual task utilization $\beta$, and task graph topology affect the accepted utilizations when using packing servers. The evaluated schedulers are described as below:

- **Packing/GEDF**: The generalized packing server over GEDF as the underlying scheduler.
- **Packing/EDF-FF**: The generalized packing server over EDF First Fit (EDF-FF) [37] as the underlying scheduler.
- **GEDF**: Global EDF is a well-known and widely used scheduler. It assigns the highest priority to the workflow with the earliest absolute deadline. Its best-known utilization bound is $\frac{2}{3+\sqrt{5}} \approx 38.2\%$ [29].
- **Federated**: Federated scheduling is the state-of-the-art generalized parallel task scheduling algorithm [29]. It achieves the highest-known utilization bound of 50% for gener-




(a) $\varphi = 20$ (b) $\varphi = 30$ (c) Summary

**Figure 3.3:** Varying Stretch $\varphi$

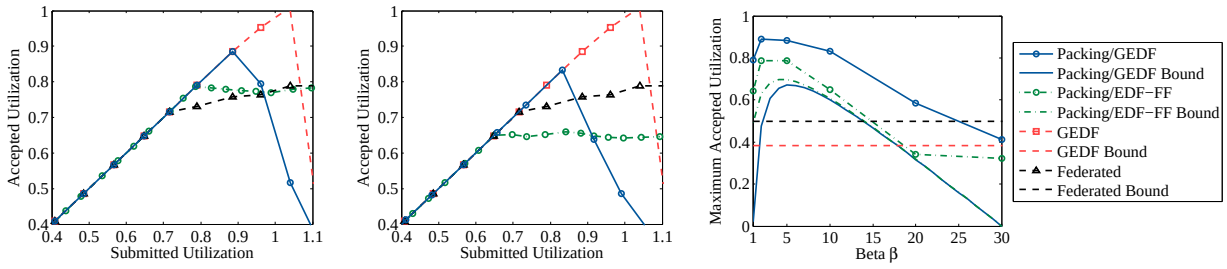
alized parallel tasks, when the stretch $\varphi$ surpasses 2. The algorithm partitions tasks based on their utilizations. For every task $\tau_i$ with utilization at least one (*i.e.*, $u_i \geq 1$), the algorithm allocates $\lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ dedicated cores to $\tau_i$, where $C_i = \sum_j c_i^j$. All other lower-utilization tasks share the remaining cores.

The simulations emulate 50 cores. In all figures, the shown utilization is divided by the number of cores in the system to normalize it to a per-core value.

The DAG generation program generates task graphs using parameters that include the number of hops on the critical path, the job WCET, the number of jobs in each DAG, and the tightness of deadline. Detailed task graph generation policies will be described close to corresponding experiments.

A large family of DAGs are generated first for each experiment. When load is increased on the horizontal axis, more DAGs are drawn from the set. If there are unfinished jobs when a DAG reaches its deadline, all remaining jobs are preempted and discarded, and the experiment records the deadline-violation event. The experiments enforce no admission controls.

The following experiments use the optimal $\beta$ values to configure the packing server according to the underlying scheduling policy, unless otherwise stated.




(a) $\beta = 5$ (b) $\beta = 10$ (c) Summary

**Figure 3.4:** Varying Budget Utilization Cap

### 3.3.1 Effect of $\varphi$

This section evaluates how the key parameter $\varphi$ in the conversion bound affects the performance of the generalized packing servers. During the task graph generation process, the program first generates a random topology. Then, the deadline of the task graph is set to the product of its critical path length $L_i$ and the given stretch $\varphi$ (*i.e.*, $D_i = L_i \cdot \varphi$). In this case, varying the parameter $\varphi$ also affects the task utilization, which we shall show later in Section 3.3.3 is an important parameter for the *Federated* scheduling policy. Tuning two key parameters in the same experiment would cause confusion about which one is the true driving factor of observed results. Therefore, we compensate by adding more jobs to keep individual task around the same utilization.

We vary the value of $\varphi$ from 5 to 30 with step 5, and compute optimal $\beta$ for each different $\varphi$ using Equation (3.16) and (3.17). The results show that, although packing server does not dominate other algorithms for every $\varphi$ value, it beats the best-known generalized parallel task utilization bound when $\varphi$ is larger than 8. Moreover, to the best of our knowledge, Packing/EDF-FF introduces the first known bound of 70% for EDF-FF algorithm with $\varphi = 30$.

Figure 3.3 (a) depicts the $\varphi = 20$ case, where the utilization bounds for Packing/EDF-ff and Packing/GEDF are 64% and 60% respectively, beating the best-known bound of 50%. The generalized packing servers over EDF-FF and GEDF start to miss deadlines when the submitted utilization surpasses 68% and 90% respectively. When $\varphi$ is set to 30 as shown

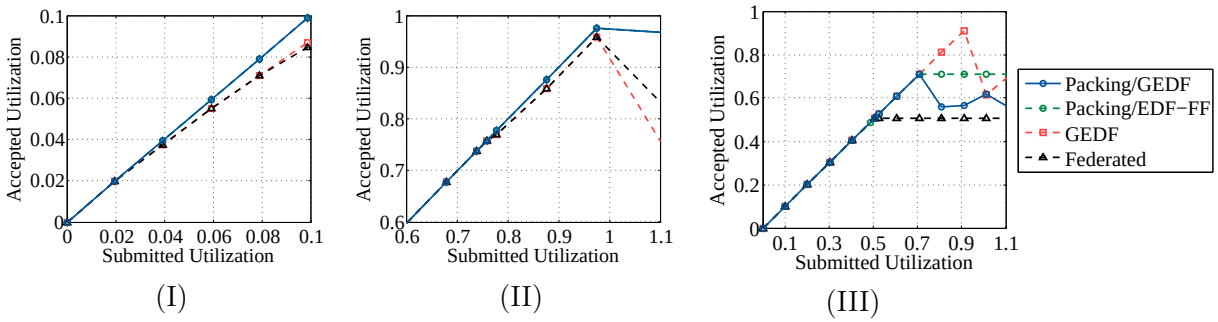



**Figure 3.5:** Unfavorable DAGs for GEDF and Federated Scheduling

in Figure 3.3 (b), the theoretical schedulability utilization bound for packing servers over EDF-FF and GEDF further improve to 70% and 67% respectively.

The curves using GEDF-related schedulers dip at higher utilization to a value below the bound. This is because no admission control is used. In the absence of admission control, EDF suffers a domino effect when the system gets overloaded causing a sharp decline in the utilization of tasks that actually meet deadlines. If admission control was used, all tasks would meet their deadlines as is clear from the point on the x-axis where the submitted utilization is equal to the bound.

Figure 3.3 (c) summarizes, in general, how the parameter $\varphi$ affects the maximum observed accepted utilization. The curves without markers indicate the theoretical utilization bound for Packing/EDF-FF and Packing/GEDF respectively. For randomly generated task graphs, both generalized packing servers accept higher utilizations at larger $\varphi$ values, following the same trend shown by the theoretical bound. Additionally, comparing the bound computed to the empirically obtained maximum schedulable utilization, we can conclude that Packing/EDF-FF leads to the least degree of pessimism when scheduling randomly generated workflows.

### 3.3.2 Effect of $\beta$

The other key parameter in the conversion bound is $\beta$. Although the optimal $\beta$ can be derived using Equations (3.16) and (3.17), it is still important to know how generalized packing

servers behave when using different $\beta$ values. Figure 3.4 plots how $\beta$ influences accepted utilizations, with $\beta$ varying from 1 to 30 and $\varphi$ fixed at 30. The optimal $\beta$ values are 4.56 and 5.47 for Packing/EDF-FF and Packing/GEDF respectively. In Figure 3.4 (a) where $\beta = 5$, Packing/EDF-FF and Packing/GEDF both achieve considerably high utilization bounds, as $\beta$ is close to the optimal value. Empirically, they also accepts more than 80% utilization. The accepted utilization of Packing/EDF-FF dramatically drops to 65% when $\beta$ increases to 10, as shown in Figure 3.4 (b). The results confirm that $\beta$ fundamentally impacts accepted utilizations. As summarized in Figure 3.4 (c), there is a clear trend of ascending followed by descending when $\beta$ increases from 1 to 30. The pinnacle in the bound curve conforms to the optimal beta value of 4.56 and 5.42. Again, Figure 3.4 shows that the utilization bound for Packing/EDF-FF achieves the least degree of pessimism.

### 3.3.3 Effect of DAG Topology

In experiments using randomly generated task graphs, GEDF performs well, and accepts more than 90% utilization in most cases. However, there are also cases where the generalized packing servers significantly outperforms GEDF and Federated scheduling policies. Consider three sets of implicit-deadline parallel tasks.

- Task set (**I**): Task $\tau_1$ consists of 100 parallel jobs , each with length $c_1^i = 1$. Task $\tau_2$ contains a single job of length $c_2^1 = 100$. The deadlines for the two workflows are $D_1 = 101$, and $D_2 = 102$ respectively.
- Task set (**II**): Compared to Task set **I**, the only difference is that $c_2^i$ shrinks to 25.
- Task set (**III**): Task $\tau_1$ consists of 27 parallel jobs, each with WCET $c_3^i = 3$. The deadline $D_1$ is set to 80.

In the above three task sets, $\tau_1$ can be copied an arbitrary number of times to achieve a desired task set utilization.

Task sets (I) and (II) represent unfavorable task graphs for the GEDF scheduler. Figure 3.5 (I) plots the simulation results for task set (I). In this case, the GEDF scheduler starts to miss deadline at a utilization of 4%, while generalized packing server servers accept up to 99% task set utilizations. In this task set, $\tau_2$ contains a large job of length $c_2^1 = 100$, and subjects to a slightly larger deadline of $D_2 = 102$, compared to other tasks. So, GEDF schedules other tasks prior to $\tau_2$, forcing $\tau_2$ to miss its deadline at a very low task set utilization. Packing servers handle these input tasks well due to the result of consolidating jobs into less parallelism. By setting $\beta = 1$, every task is packed into a single budget, allowing $\tau_2$ to enjoy its execution opportunity when the task set utilization is below 99%. Task set (I) is out of the scope considered in literature [29], as the stretch $\varphi$ is smaller than $\frac{3+\sqrt{5}}{2}$. Task set (II) presents tasks with larger stretches where the $\frac{2}{3+\sqrt{5}} \approx 38.2\%$ utilization bound is applicable. Figure 3.5 (II) elucidates the result. GEDF and Federated schedulers miss deadlines when task set utilization reaches 76%, whereas packing servers accept all tasks when the total utilization stays below 97%.

Task set (III) is unfavorable for the Federated scheduler. The simulation results are presented in Figure 3.5 (III). The accepted utilization of the Federated scheduling policy stays at 50%, which is its theoretical schedulability utilization bound. It is because, under this configuration, $\frac{C_i - L_i}{D_i - L_i} = \frac{81-3}{80-3} = \frac{78}{77}$ is slightly larger than 1, forcing the Federated scheduler to allocate 2 cores to each task. As a result, half of the resources in the platform are wasted. From this perspective, Federated scheduling algorithm prefers higher individual task utilizations. Generalized packing servers accept 72% task set utilization. GEDF does not miss any deadline when the task set utilization is smaller than 90%.

These measurements convey that no single scheduler dominates all others. Therefore, systems without prior knowledge of the DAG topology can only turn to the theoretically utilization bound for schedulability guarantees.

## 3.4 Discussion

The generalized packing server is a novel solution to schedule task graphs using independent job scheduling algorithms. This chapter presents a simple packing server. It can be improved along several directions.

- **Non-Identical Budgets**: The generalized packing server described in this chapter creates *identical* budgets. However, this requirement may not be necessary. Non-identical budgets may potentially help reduce the amount of idle budget during execution, which in turn improves the conversion bound. This is because the degree of concurrency in the task graphmay differ from one point in execution to another.

- **Budget Priority Exchange**: In our model, every budget is dedicated to one task. If a budget is invoked when no eligible job is left, the budget is completely wasted. One possible improvement is to borrow the idea of *Priority Exchange* [33, 43, 44] from earlier aperiodic server literature, which allows servers to loan unused capacity to others. Similarly, an improved packing server could loan idle budgets to another packing server, which would help reduce the total amount of idle budget over the entire task set.

- **Optimized Application Scheduler**: This chapter presents a general result that allows the server to use *any* work-conserving scheduling algorithm to determine which jobs to run next. We prove that the bound is tight in the worst case if budgets are constructed using our theorems. However, outside the worst case, or if budgets are configured differently (e.g., are not identical), some scheduling algorithms might be superior to others. In this case, it would be meaningful to develop scheduling algorithms that help the packing server further improve the conversion bound.

- **Distinguish Large and Small Stretch**: As suggested by the conversion bound, the stretch $\varphi$ is a critical parameter that significantly affects the resulting conversion bound. For simplicity, we use the minimum stretch of all task graphs to calculate the conversion

bound. Inspired by the Federated scheduling algorithm [29], treating tasks with large and small utilization differently would increase the schedulability bound. Similarly, the conversion bound could also be improved by categorizing stretch into tight and loose classes, and distinguish them during scheduling. The above improvements are left for future work.

## 3.5 Related Work

The generalized parallel task scheduling problem has been recently studied on multiprocessor platforms. Baruah *et al.* [27] prove that EDF can achieve a 2X speedup bound for a single recurrent workflow. Saifullah *et al.* [26] propose to arrange a workflow into stages, and then the workflow's deadline is split and assigned to each stage. If some optimal algorithm can successfully schedule the original workflow, their solution is guaranteed to satisfy the same deadline with 4X (speedup bound) speed processors. When the workflow is restricted to a fork-join model [45, 46], Lakshmanan *et al.* [47] improve the speedup bound to 3.42. Li *et al.* [28] develop a capacity augmentation bound of $4 - \frac{2}{m}$ for workflows, which immediately leads to a simple and effective schedulability test. More recently, Li *et al.* [29] improve the capacity augmentation bound to 2 using the federated scheduling algorithm for implicit deadline workflows. They then prove the same result for stochastic parallel tasks [48]. Analysis from literature [23] shows a speedup bound for the federated scheduling algorithm to be $\left(4 - \frac{2}{m}\right)$ when using arbitrary-deadline sporadic DAG models. Fonseca *et al.* [24] further introduce conditional executions into the generalized parallel task models to capture the conditional constructs, such as *if-then-else* clauses. Baruah *et al.* [25] later prove a speedup factor of $\left(2 - \frac{1}{m}\right)$ for the conditional sporadic generalized parallel tasks. These efforts study the generalized parallel task scheduling problem by directly analyzing its behavior on multiprocessor platforms.

Compared to above approaches, packing servers take a different path to solve the generalized parallel task scheduling problem. Namely, they offer a mechanism for converting the

original task set into independent tasks. Unlike other conversion-based approaches, in packing servers the resulting independent tasks (server budgets) are subject to the same original end-to-end deadlines. Since the deadlines are not broken into per-segment deadlines, no artificial deadline constraints are introduced, which improves schedulability. Our prior work [35] analyzes the workflow scheduling problem under the assumption of a clairvoyant scheduler. Specifically, the previous approach required that the packing server know the exact future task execution timeline, in order to determine the exact time intervals when one of the budgets will be scheduled on one of the resources in the future. This was accomplished by simulation, which made the previous approach of limited applicability. In particular, it applied to Map/Reduce scheduling [49], where all execution it batched (and hence predictable) and where schedulers are heavy-weight application-layer entities burdened by layers of cloud middleware (making it possible to accommodate the overhead of simulating the execution of the workload batch into the future to compute future start and end times of tasks). Clearly, in the context of embedded computing, schedulers must be lighweight and efficient. Moreover, models such as sporadic tasks, or tasks that can finish early (before expiration of their worst-case execution time) cannot be accommodated by prior work because they make it impossible to predict exact future start and end times of tasks.

The generalized packing server, described in this chapter, no longer requires scheduler clairvoyance. Instead, it can use *any work conserving scheduling policy* to pack tasks into server budgets. Hence, results derived for this server are much more broadly applicable. With the generalized packing server techniques, the generalized parallel task scheduling problem will continuously benefit from the rich body of existing and future advances of independent task scheduling algorithms and analysis.

# Chapter 4: Manager For Randomly Partitioned Storage Systems

The previous chapter assumes that the workload can be distributed to any server with no penalty. Though this assumption is valid in statless web servers, the memory cache cluster cannot enjoy this simplicity.

The memory cache cluster usually sits in front of the database or distributed file system tier to offer fast in-memory data access by running Memcached instances. The cache layer is playing a very important role. For example, Facebook reported that their cache hit ratio is higher than 95% [50], which significantly reduces the database workload. (Below, we use cache and Memcached interchangeably.) In industry data centers, workload seen by the cluster varies over time, and the peak workload can be as much as twice the valley workload [51]. This property offers us golden opportunities to apply dynamic server provisioning policies, such that when the workload is light, a subset of the servers is turned off to save energy. As the cache cluster may consist of hundreds or thousands of servers[52], the monetary benefits brought by energy management could be large.

Dynamic server provisioning, a common energy management methodology, is widely used for both stateless web servers and distributed replicated file systems. Unfortunately, due to distinct characteristics of cache clusters, directly applying dynamic server provisioning will suffer from severe performance degradation during provisioning dynamics (i.e., when servers are turned on or off). For stateless web servers, a reasonable assumption is that one request can be handled by any server without much performance penalty. Nevertheless, this

assumption does not hold for the cache cluster. If one request is directed to some cache server that does not have the requested data, the request will have to reach the database tier (or DFS), which induces high response time. For distributed File systems, requests are usually directed by using deterministic load distribution policies. Many current designs employ master server [53], name node [54], or meta servers [55] to store the data chunk location information. One request will first reach the master server to look up the corresponding chunk server address, and then communicate with the chunk server to fetch data. However, given that the cache servers are designed for fast in-memory data access for a large number of data items, similar look-up operations, which involve one or more disk seeks, are too slow to serve the cache tier.

Under static scenarios, achieving load balancing is trivial: The web server can simply hash the requested data ID to a large integer range, and then apply the modulo operator to wrap it to a valid cache server ID. Reddit, a popular social news website, did use this scheme before. However, they soon experienced the pain of expanding cache clusters for capacity upgrades [56]. This is because, in an $n$-server cache cluster, if one more server is added, this simple solution will re-map $\frac{n}{n+1}$ data IDs to cache servers where the requested data is not in-memory. Therefore, in expectation, $\frac{n}{n+1}$ of the requests will reach the database tier simultaneously, and the databases are easily overloaded. The same problem happens when dynamic server provisioning is used to turn servers on or off. Consistent hashing alleviates the problem to some degree. Nevertheless, how to balance load under dynamics and how to make dynamic server provisioning smooth enough are still problems that are yet to be solved.

## 4.1 Problem Formulation and Assumptions

Along this direction, we aim at eliminating the performance penalties caused by dynamic cache cluster provisioning in a 3-tier server cluster as shown in Fig. 4.1. Time is divided into slots. Let $N$ denote the total number of cache servers, and $n(t)$ denote the number of

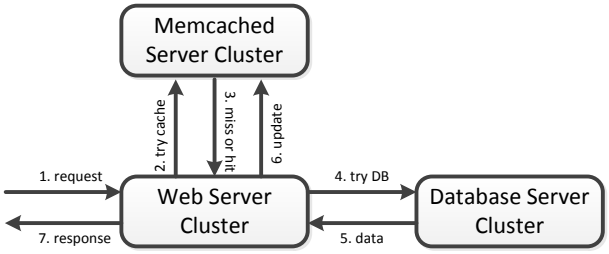



**Figure 4.1:** Simplified Information Flow

running cache servers in the $t^{th}$ time slot. The data key $key^d$ can be a page title in Wikipedia, a user ID in Facebook, etc. With $key^d$, the web server cluster fetches corresponding data $d$ from either the cache server $m^d$ or the database tier, and then presents the response to users. Cache servers store data in the form of $(key, data)$ pairs. We define a piece of data as "hot" if it is touched at least once during the past $TTL$ seconds. Please note that, we do not make any assumptions on the cache eviction strategy (LRU, fixed expiration duration, etc.). At one time instance, a cache server is called *active* if it hosts "hot" data and serves requests. Otherwise, it is called *inactive*, and operates in low power status.

**Assumptions**

- Each object in cache is of the same size. Even though the size of pages or user accounts would vary considerably, they can be divided into fixed-size pieces. One piece is considered as the basic unit of objects in cache. Actually, modern storage clusters already employ such idea [53–55].
- The load of requests have temporal behavior, and the gap between the peak and the nadir load is huge. Our study of Wikipedia's traces supports this assumption.

**Objectives**

- *Balance load distribution in cache tier under dynamics*: We define the load as the amount of data objects served by the cache cluster. In static environments, the load can be easily balanced by using hash and modular operations. We pursue the same level of load balance in face of dynamic server provisioning.

- *Minimize data movements during re-balancing transitions*: Proteus should achieve minimum data migration, such that only at most $\frac{|n(t+1)-n(t)|}{\max\{n(t+1),n(t)\}}$ of the in-cache data is remapped when the number of active servers changes from $n(t)$ to $n(t+1)$.

- *Eliminate re-balancing transition delay spikes*: Dynamic server provisioning should never hurt the performance too much. We aim at managing energy with no delay penalty.

Generally, our goal is to design a provisioning actuator that executes decisions according to server provisioning policy without degrading the system performance in terms of response time. Please note that designing the best provisioning policy is *not* our focus. Different policies [11, 57, 58] can cooperate with Proteus with no confliction.

## 4.2 Load Balance under Dynamics

In this section, we describe an algorithm that not only deterministically balances load distribution under provisioning dynamics but also guarantees migrating minimum amount of data during each re-balancing transition. Our algorithm inherits ideas of consistent hashing and virtual nodes [59, 60], and focuses on generating a virtual node placement strategy.

Load balance under dynamics requires each server handles equal amount of objects even if the number of active servers dynamically changes. As already used in many Memcached clusters [61], consistent hashing and virtual nodes help to balance load among multiple cache servers to some degree [59, 60]. In consistent hashing, data keys and servers are hashed into the same key space, which forms a hashing ring. The ring acts as an indirect layer of index for the hash. Each obejct will be stored in the first server that succeeds the data key on the ring. When one server is turned off, its direct successor takes care of its workload. One physical server may have multiple direct successors by placing multiple virtual nodes on the hashing ring. Therefore, the load of one server can be spread out to more physical servers when necessary. However, without careful design, consistent hashing and virtual nodes alone

do not guarantee load balance in face of provisioning dynamics. In this section, we borrow the idea of consistent hashing, and improve it by designing and analyzing a virtual node placement algorithm that deterministically balances workload among all active Memcached servers.

### 4.2.1 Fixed Provisioning Order

In data centers, every individual server is under control. Hence, it is possible to turn on/off servers according to any fixed order. We argue that, a provisioning scheme with a fixed order is not any weaker than the one that adapts to arbitrary orders. The reason is that, when no failure presents, the fixed order can be maintained easily. If some server crashes, we have already lost the data in cache, and both schemes need some fault tolerant solutions for reconstructing the cache or directing the requests to some redundant caches.

Fixing provisioning order eliminates one dimension of freedom of the load balance problem, thus simplifies the algorithm design. Well designed order further improves power savings. For example, the decreasing order of server efficiency should be better than a random order, where server efficiency is defined as the amount of workload served per unit of energy. The system administrators should be responsible for choosing a reasonable order, which is not the focus of this chapter. Our solutions are able to cooperate with any fixed order.

Define the list $(s_1, s_2, ..., s_N)$ as the fixed order for dynamic provisioning, where $N$ is the number of servers. Let $n(t)$ denote the number of active servers in the $t^{th}$ time slot. In other words, servers in set $\{s_i \mid 1 \leq i \leq n(t)\}$ are active in the $t^{th}$ time slot.

### 4.2.2 Optimal Number of Virtual Nodes

Let $\mathcal{V}_i = \{v_{i1}, v_{i2}, v_{i3}, ...\}$ denote the set of virtual nodes of the server $s_i$. Any request within the key range between the virtual node $v_{ij}$ and its direct predecessor on the hashing ring will be served by virtual node $v_{ij}$, and thus, physical node $s_i$. We call this key range the *host range* of the virtual node $v_{ij}$. As the provisioning algorithm turns on and off physical nodes

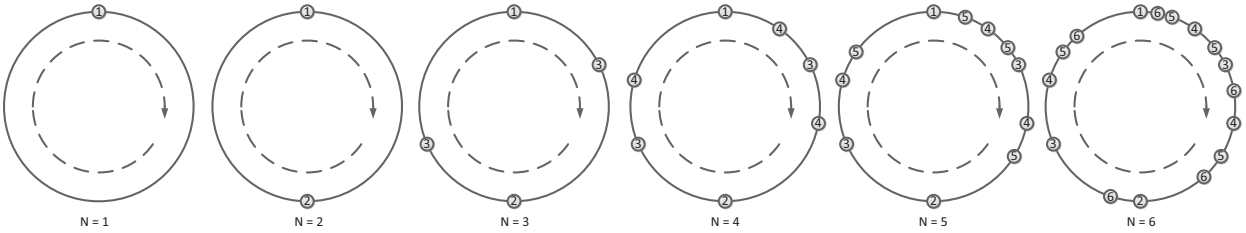



**Figure 4.2:** An example of virtual node placement

dynamically, the direct successor of $v_{ij}$ varies as well. However, recall that the order for provisioning is static, $v_{ij}$ always precedes the same direct successor if the number of active servers is the same. We define $v_{ij}$'s *final successor* as the direct successor when $i-1$ servers are on. Denote $v_{ik_i} \to v_{jk_j}$ as the relation between two virtual nodes $v_{ik_i}$ and $v_{jk_j}$, such that $v_{jk_j}$ is the final successor of $v_{ik_i}$ on the consistent hashing ring. Let $\mathcal{P}_i^s$ denote the set of final successor servers of server $s_i$ (as illustrated in Fig. 4.2), *i.e.*,

$$\mathcal{P}_i^s = \{s_j \mid \exists v_{jk_j} \in \mathcal{V}_j, \exists v_{ik_i} \in \mathcal{V}_i,\ s.t.\ v_{ik_i} \to v_{jk_j}\}.$$

In order to achieve load balance under provisioning dynamics, the virtual node placement policy should satisfy the following conditions: (1) when one physical node is turned off, objects served by this node should be evenly migrated to all other running physical nodes, and (2) when one physical node is turned on, it should take an identical amount of objects from all other active physical nodes. We call this condition the *Balance Condition (BC).*

Obviously, among all solutions that satisfies BC, the one with minimum number of virtual nodes is preferred, since less virtual nodes introduce less overhead in terms of both space consumption and query complexity. We now prove that $\frac{N^2-N}{2}+1$ is the lower bound on the number of virtual nodes in order to meet BC.

**Theorem 3.** *At least* $\frac{N^2-N}{2}+1$ *virtual nodes are required to satisfy BC.*

*Proof.* We first introduce a necessary condition of BC, which we call the *pseudo BC*:

$$\mathcal{P}_i^s \supset \{s_j \mid 1 \le j \le i-1\}.$$

The pseudo BC states that to achieve load balance, the final successor set $\mathcal{P}_i^s$ should at least cover $s_j, \forall j < i$. Otherwise, if $\exists j < i, s_j \notin \mathcal{P}_i^s$, $s_i$'s workload will not be directed to $s_j$ when $s_i$ is turned off. Hence the load is not balanced. Please note that the pseudo BC does not guarantee the feasibility of host ranges. [1].

One virtual node can only have one final successor on the hashing ring. Since $|\mathcal{P}_i^s| \geq i-1$, $s_i$ needs at least $i-1$ virtual nodes to precede $i-1$ final successors. The corner case is $i = 1$, where $s_1$ needs to have at least 1 virtual node. Therefore, altogether, at least $1 + \sum_{i=2}^{N}(i-1) = \frac{N^2-N}{2} + 1$ virtual nodes are required. □

### 4.2.3 Virtual Node Placement

In this section, we elaborate the virtual node placement algorithm. Assume the key space size is $\mathcal{K}$. When generating the placement solution, physical nodes are served one by one according to the fixed provisioning order. For $s_i$ $(i > 1)$, the algorithm places $i-1$ virtual nodes on the consistent hashing ring, denoted by $\{v_{ij} | 1 \leq j \leq i-1\}$. When placing $v_{ij}$, the algorithm borrows $\frac{\mathcal{K}}{i(i-1)}$ continues host range from one feasible virtual node of $s_j$, and assigns it to $v_{ij}$. Fig. 4.2 shows an illustrative example with 6 physical nodes. In Section 4.2.4, we prove that the algorithm is correct and the host ranges are balanced among all physical nodes.

As each virtual node can be identified by its host range, placing virtual node is equivalent to assigning an unique host range to each virtue node. Below, we use host range and virtual node interchangeably. The pseudo code is shown in Algorithm 4.1. The algorithm takes 2 inputs: the number of physical nodes $N$, and the consistent hashing ring range $\mathcal{K}$. Line $2-3$ adds $s_1$'s only virtual node $v_{1,0}$ into $s_1$'s host range set $R[1]$. Initially, $v_{1,0}$ covers the entire consistent hashing ring. The loop starting on line 4 enumerates all $s_i, i > 1$ to add their virtual nodes. Line $5-15$ iterates to place each of $s_i$'s virtual node $v_{ij}$. For $v_{ij}$, the inner loop on line 6 searches for one feasible virtual node $r$ of $s_j$ whose host range is larger than

[1] Counter Example: Place the virtual nodes clockwise on the consistent hashing ring with the following order: $1, 2, 3, ..., n, 2, 3, 4, ..., n, ..., (n-3), (n-2), (n-1), n, (n-2), (n-1), n, (n-1), n$. The numbers are corresponding to physical server ID. When $n > 6$, it is not possible to have all responsible ranges positive.

$\frac{\mathcal{K}}{i(i-1)}$. When found, $v_{ij}$ borrows $\frac{\mathcal{K}}{i(i-1)}$ from $r$, and the algorithm inserts $v_{ij}$ into $s_i$'s virtual node set $R[i]$.

**Algorithm 4.1** Virtual Node Placement

**Input:** Number of physical nodes $N$, consistent hashing ring range $\mathcal{K}$.
**Output:** Virtual node placement strategy $vps$.

1: $R \leftarrow$ an array of $N$ empty set
/* Host range set array for all physical vertices. */
2: $v_{1,0}.start \leftarrow 0$, $v_{1,0}.len \leftarrow \mathcal{K}$
3: $R[1] \leftarrow \{v_{1,0}\} \bigcup R[1]$
/* Initially, the host range of $s_1$'s only virtual node starts at 0 with length $\mathcal{K}$*/
4: **for** $i \leftarrow 2$ **to** n **do** /* enumerate all $s_i$ */
5: **for** $j \leftarrow 1$ **to** i-1 **do** /* compute host range for $v_{ij}$ */
6: **for** $r$ **in** $R[j]$ **do**
/* borrow host range from one feasible virtual node of $s_j$*/
7: **if** $r.len > \frac{\mathcal{K}}{i(i-1)}$ **then**
8: $v_{ij}.start \leftarrow r.start$, $v_{ij}.len \leftarrow \frac{\mathcal{K}}{i(i-1)}$
9: $r.len \leftarrow r.len - \frac{\mathcal{K}}{i(i-1)}$
10: $r.start \leftarrow r.start + \frac{\mathcal{K}}{i(i-1)}$
11: $R[i] \leftarrow \{v_{ij}\} \bigcup R[i]$
12: **Break**
13: **end if**
14: **end for**
15: **end for**
16: **end for**
17: $vps \leftarrow \emptyset$ /*serialize host ranges in a sorted array for fast access*/
18: **for** i $\leftarrow 1$ **to** $N$ **do**
19: **for** $r$ **in** $R[i]$ **do**
20: $vps \leftarrow vps \bigcup \{r\}$
21: **end for**
22: **end for**
23: sort $vps$ based on $vps[\cdot].start$
24: **Return** $vps$

### 4.2.4 Algorithm Analysis

We prove the virtual node placement solution generated by Algorithm 4.1 balances the sum of host ranges assigned to each physical server.

*Proof.* **basis:** If $N = 1$, there is only one node. Hence the host range is trivially balanced.

**Inductive Step:** Assume the algorithm is correct for $N = k$ (*i.e.*, the host range of each physical node is $\frac{\mathcal{K}}{k}$). For $N = k + 1$, the host range of each physical node should be $\frac{\mathcal{K}}{k+1}$. Hence, to achieve load balance, the $(k+1)^{th}$ node needs to deduct $\frac{\mathcal{K}}{k(k+1)}$ from at least one virtual node of each physical node with smaller ID, which is done in line 6 $\sim$ 14 in Algorithm 4.1. We now show that it is always possible to find one feasible virtual node from $R[j]$ whose

host range length is at least $\frac{\mathcal{K}}{k(k+1)}$. By contradiction, we assume that for some $s_i$, $(i < k+1)$, the host range length of all $v_{ij}, (j < i)$ is smaller than $\frac{\mathcal{K}}{k(k+1)}$. Therefore, the sum of host range length deducted from $s_i$ by physical nodes set $\{s_l | i < l < k+1\}$ is at least at least

$$(i-1)\left(\frac{\mathcal{K}}{i(i-1)} - \frac{\mathcal{K}}{k(k+1)}\right). \tag{4.1}$$

According to the algorithm, each $s_l, l > i$ borrows $\frac{\mathcal{K}}{l(l-1)}$ from $s_i$. So, we also have, $\sum_{l=i+1}^{k} \frac{\mathcal{K}}{l(l-1)} = \frac{\mathcal{K}}{i} - \frac{\mathcal{K}}{k}$, which is smaller than formula (4.1),

$$(i-1)\left(\frac{\mathcal{K}}{i(i-1)} - \frac{\mathcal{K}}{k(k+1)}\right) - \left(\frac{\mathcal{K}}{i} - \frac{\mathcal{K}}{k}\right) = \mathcal{K}\frac{ik - i^2}{ik(k-1)} > 0. \tag{4.2}$$

Hence, the assumption does not hold. It is always possible to borrow $\frac{\mathcal{K}}{k(k+1)}$ host range length from at least one virtual node of each physical node $s_i$, $(i < k)$. Therefore, the host ranges are balanced in $N = k+1$ case. □

### 4.2.5 Fault Tolerance

Given the scale of data centers, server failure is not an exception, instead, it occurs frequently. For the sake of fault tolerance, Proteus can easily extend to embrace redundancies. For example, if Proteus is set to keep $r$ replications for each piece of $(key, data)$ pair, it just needs to construct $r$ consistent hashing rings with $r$ different hash functions. Different hashing rings share the same virtual nodes placement policy. If a $key$ falls in the host range of any virtual node of $s_i$ on any hashing ring, $s_i$ will store one copy of the $(key, data)$ pair. This strategy does not guarantee that the $r$ replications will be stored in $r$ different servers. However, the confliction probability will be low. If the hash function distributes the keys evenly and randomly. The probability that no confliction occurs is:

$$P_{nc} = \prod_{i=0}^{r-1} \frac{n(t) - i}{n(t)}. \tag{4.3}$$

As $r$ is usually a small number (*e.g.*, 2 or 3), and $n(t)$ is much larger (*e.g.*, a few thousand), $P_{nc}$ for each data piece should be close to 1.

## 4.3 Smooth Provisioning Transition

In this section, we elaborate an efficient solution for smooth provisioning transition, which is executed before turning on/off any cache server.

Each Memcached server may host tens or hundreds of gigabyte "hot" data [62]. If we turn off the Memcached servers brutally, we will lose a considerable amount of in-cache data. As consequences, some users might see delay spikes. We propose *Smooth Provisioning Transition* to avoid such spikes. Our high level idea is that when turning off $s_i$, $s_i$ postpones this operation by $TTL$ seconds to migrate "hot" data to its final successor on demand. Define $\mathcal{H}_t$ as the consistent hash in $t^{th}$ time slot. Let $key^d$ and $m_t^d$ denote the data key and the corresponding Memcached server ID for data $d$ in $t^{th}$ time slot. Since ($key^d$, $d$) may reside in both $m_t^d$ and $m_{t+1}^d$ during the transition stage, web servers need to know which cache server they should query.

Three objectives must be satisfied. First, the transition must be unblocking. Memcached is designed as a fast key value store. The performance of the Memcached tier should not be interfered too much, otherwise the solution is useless. Second, only the "hot" data should be transferred, otherwise bandwidth and computational resources are wasted. Third, the transition delay should be small and bounded. The number of Memcached servers is tuned to catch up with load dynamics. Thus, long transition delay harms the system agility.

To achieve these objectives, we propose to use one Counting Bloom Filter [63, 64] as the content digest for each cache server. Below, we elaborate the details of maintaining and configuring the counting Bloom filter.

### 4.3.1 MemCached Digest

Bloom filter is a data structure that stores a set of elements and supports membership queries. Counting Bloom filter is a variant of Bloom filter that supports both element insertion and deletion. In our solution, each Memcached server maintains one counting Bloom filter for

in-cache keys. We call the counting Bloom filter the *digest*. Let $\mathcal{F}_i$ denote the *digest* of $s_i$. When $key^d$ and its data are inserted into (or deleted from *resp.*) $s_i$, $\mathcal{F}_i$ will also be updated accordingly. We do not assume any cache replacement policy, as long as $\mathcal{F}_i$ is consistent with $s_i$'s content, our solution works.

At the beginning of the transition stage, digests (a few $KB$ each) will be broadcasted to all web servers. Then, the web server knows what is in-cache and what is not. The algorithm for data retrieval is described in Algorithm 4.2. Line $2 \sim 4$ checks whether the data is in $s_{m_{t+1}^d}$. If hit, the algorithm returns the data and does not go any deeper. If miss, line 6 checks whether the data resides in $s_{m_t^d}$. If yes, it retrieves the data. Since Bloom filter may have false positives, line 8 further checks if the data is indeed retrieved. If and only if both attempts miss, will the request reach the database tier. Therefore, the database tier will not realize transition dynamics is taking place. If the data is retrieved from either $s_{m_t^d}$ or the database tier, the algorithm also updates $s_{m_{t+1}^d}$.

**Algorithm 4.2** Data Retrieval

```
 1: procedure FETCH_DATA(key^d)
 2:     data ← s_{m_{t+1}^d}.get(key^d)
 3:     if NULL ≠ data then
 4:         return data                                   /* found in new server. */
 5:     else
 6:         if F_{m_t^d}.check(key^d) then                /* data is "hot". */
 7:             data ← s_{m_t^d}.get(key^d)
 8:         end if
 9:         if NULL = data then                           /* false positive or "cold" data. */
10:             data ← database.get(key^d)
11:         end if
12:         s_{m_{t+1}^d}.put(key^d, data)
13:         return data
14:     end if
15: end procedure
```

The algorithm has two important properties. First, for "hot" data $d$, only the first request will reach $s_{m_t^d}$, all subsequent requests will find the data in $s_{m_{t+1}^d}$. Therefore, no bandwidth and computational resources are wasted. Second, Memcached servers can be safely turned off after $TTL$ seconds. Because, if one piece of data $d$ is touched in the past $TTL$ second, it has already been transferred to $s_{m_{t+1}^d}$. If it is not touched in the past $TTL$ second, it is no longer "hot", and can be safely discarded.

| Symbol | Description |
|---|---|
| $h$ | Number of different hash functions |
| $\kappa$ | Number of inserted keys |
| $l$ | Number of counters in Bloom filter |
| $b$ | Number of bits in each counter |

**Table 4.1:** Bloom Filter Parameters and Descriptions I

### 4.3.2 Bloom Filter Configuration

The Bloom filter employs a bit array with $h$ hash functions. All bits in the array are initialized to 0. When an element $e_i$ is inserted, bits at position $hash_1(e_i)$, $hash_2(e_i)$, ..., $hash_h(e_i)$ will be set to 1. When querying the membership of $e_i$, the Bloom filter checks the bits at position $hash_1(e_i)$, $hash_2(e_i)$, ..., $hash_h(e_i)$. If all of them are 1, the Bloom filter answers "yes". Otherwise, it answers "no". Bloom filter may have false positives.

Counting Bloom filter is a variant of Bloom filter, which uses counters rather than bits. When one element is inserted/deleted, the corresponding counters will increase/decrease by 1. The Counting Bloom filter suffers from both false positive and false negative issues. False negative is caused by either deleting an absent element (due to false positive) or counter overflow. In our scenario, the first case will never happen. The deletion from *digest* is only triggered by the deletion from Memcached. The Memcached deterministically knows whether an element is in-cache or not. Therefore, deleting absent element from *digest* will never happen. However, counter overflow may occur. In order to achieve low false negative rate, the Bloom filter should curb its counter overflow probability. Otherwise, if counter overflows, underflow will also be possible, which triggers false negatives. The counter overflow probability decreases with the increase of the counter size and the number of counters. However, more and larger counters lead to more memory consumption and higher overhead when broadcasting *digest*s. We now compute Bloom filter configurations to achieve minimum memory consumption subject to given false positive and false negative rate constraints ($p_p$, $p_n$). Table 4.1 shows the descriptions of symbols.

The probability that one counter remains 0 after inserting $\kappa$ keys into $l$ counters with $h$

hash functions is $(1-1/l)^{\kappa h}$. Hence, the false positive rate is,

$$(1-(1-1/l)^{\kappa h})^{\kappa} \approx (1-e^{-\frac{\kappa h}{l}})^{h}. \tag{4.4}$$

As Memcached is designed as a high performance software, fewer hash functions are preferred. Therefore, we take $h$ as a parameter.

As we have stated above, the counter overflow (and hence, underflow) is the only reason of false negatives in our system. The probability that any counter is greater than $2^b$ is [63, 64],

$$\mathbf{Pr}\left(\max(counter) \geq 2^b\right) \leq l\binom{\kappa h}{2^b}\frac{1}{l^{2^b}} \leq l\left(\frac{e\kappa h}{2^b l}\right)^{2^b} \tag{4.5}$$

Let $G_p(l) = (1-e^{-\frac{\kappa h}{l}})^h$ and $G_n(l,b) = l(\frac{e\kappa h}{2^b l})^{2^b}$ represent the false positive and false negative rate respectively. Then, our objective is:

$$\begin{aligned} &Minimize && lb \\ &s.t. && G_p(l) \leq p_p \quad G_n(l,b) \leq p_n \end{aligned} \tag{4.6}$$

where $p_p$ and $p_n$ are given false positive and false negative probability bounds.

Take partial derivatives of $G_n(l,b)$ with respect to $l$, and $b$ respectively, we have

$$\begin{aligned} \frac{\partial G_n(l,b)}{\partial b} &= \mathcal{C}\left(\frac{b}{2}\ln\frac{e\kappa h}{2^b l} - \ln 2\right) < \mathcal{C}\left(\frac{b}{2}\ln\frac{e\kappa h}{2^b l}\right) \\ \frac{\partial G_n(l,b)}{\partial l} &= G_n(l,b)\frac{1-2^b}{l} > -\mathcal{C}\frac{1}{l} \end{aligned} \tag{4.7}$$

where $\mathcal{C} = G_n(l,b)2^b$. When

$$\frac{bl}{2} > \left(\ln\frac{2^b l}{e\kappa h}\right)^{-1} \tag{4.8}$$

| Scenario | Server Provisioning | Workload Distribution |
|---|---|---|
| Static | All servers are on | simple hash with modular |
| Naive | Dynamically tuned | simple hash with modular |
| Consistent | Dynamically tuned | Consistent hashing |
| Proteus | Dynamically tuned | Proteus's algorithms |

**Table 4.2:** Bloom Filter Parameters and Descriptions II

we have

$$\frac{\partial G_n(l,b)}{\partial b} < \frac{\partial G_n(l,b)}{\partial l}. \tag{4.9}$$

Please note that 4.8 is almost always true, since $2^b l$ need to be much larger than $e\kappa h$ to achieve low false positive and false negative rate. Therefore, fixing $lb$, $G_n(l,b)$ decreases with the decrease of $l$. Hence, the optimal point is reached at the minimum possible $l$, which can be derived from the inequality $G_p(l)$. Hence, the optimal solution is

$$l = \frac{-\kappa h}{\ln\left(1 - p_p^{\frac{1}{h}}\right)}, \quad b = \ln\left(\beta e^{\mathcal{W}\left(-\frac{\ln\gamma}{\beta}\right)}\right) \tag{4.10}$$

where $\beta = \frac{e\kappa h}{l}, \gamma = \frac{p_n}{l}$, and the function $\mathcal{W}(x)$ is the inverse of $xe^x$ (Lambert function [65]).In practical, $b$ is an integer with a very small range. Therefore, we can enumerate all possible values of $b$ and pick the optimal one. For example, with ($\kappa = 10^4$, $h = 4$, $p_p = p_n = 10^{-4}$), ($l = 4 \times 10^5, b = 3$) is more than enough, which takes about $150KB$ memory per digest.

## 4.4 Performance Evaluation

In the evaluation part, we compare four different scenarios: Static, Naive, Consistent, and Proteus. The detailed description is shown in Table 4.2.

In this chapter, we attack the performance penalty when applying dynamic cluster provisioning. For the sake of fairness, we use the same cluster provision feedback loop. Due the

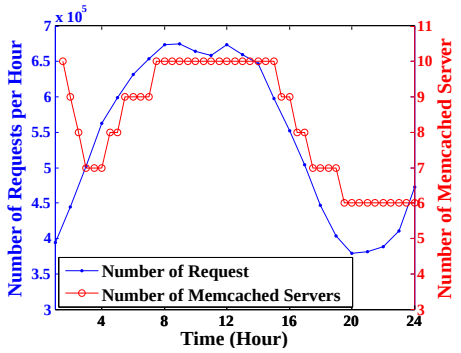



**Figure 4.3:** WikiPedia Workload

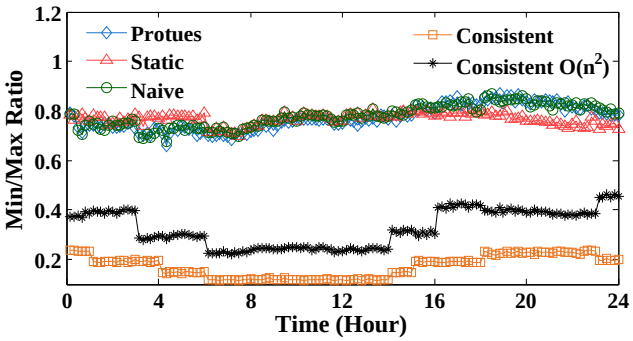



**Figure 4.4:** Load Balancing

limit of space, the details of the feedback loop is omitted here. We run the feedback control algorithm along with Proteus with the delay bound set to 0.5 second. The feedback loop reference point is set to 0.4 second to tolerate overshot. The loop updates its status every 30 minutes. After this experiment, we know the number of running cache servers assigned to each 30-minute time slot, which is shown as the curve with small circles in Fig. 4.3. Here we use the number of requests as the workload based on which we do dynamic provisioning. It is true that the number of data being visited should be the real workload, because the bottleneck resource in Memcached cluster is the main memory. However, although the number of requests is not strictly linear proportional with the number of data being visited, it is a reasonable estimation and it is also easy to get. Please also note that our major focus is not designing optimal provisioning algorithm but the load balancing and smoothness of transition during provisioning dynamics. Then we apply the same cluster provisioning result, Wikipedia data and Wikipedia workload to all 4 different scenarios. In this way, the only differences of 4 scenarios are the load balancing algorithm and their behavior in face of provisioning transition.

### 4.4.1 Load Balancing under Dynamics

We first evaluate how our load balancing algorithm works with real Wikipedia workload [66]. The trace contains timestamp and requested URL for every single user request. We first do some preliminaries to distill the requests that hit English Wikipedia. Then we sort all the

request by timestamp, and count the number of requests inside every 1-hour time window. The result workload is depicted as the curve with dots in Fig. 4.3. Due to the lack of Wikipedia image data, our system cannot serve the real workload, and hence the response time for real workload is not measured. However, given that we are evaluating whether the load is balanced or not under dynamics in Memcached tier, we do not actually need the real image data. All we need is to measure whether the number of requests handled by each Memcached server is roughly the same. Again, as we stated before, the chapter does not focus on how to build an efficient and accurate feedback loop for server provisioning. Here, we use the provisioning result as shown as the curve with circles in Fig. 4.3.

In Fig. 4.4, we study the performance of load balancing of the algorithms by depicting the ratio of $min\{L_i^t | i < n(t)\}$ over $\max\{L_i^t | i < n(t)\}$ for all $t$, where $L_i^t$ is the workload on Memcached server $i$ in $t^{th}$ time slot. The curve with dimonds shows the load distribution by using Proteus, while the curves with small circles and triangles show the load distribution by using naive and static solutions (hash and modular) respectively. Clearly, Proteus achieves as good performance as the static and naive solutions. The curve with squares depicts the load distribution using consistent hashing with $\mathcal{O}(\log n)$ randomly placed virtual nodes. The result is much worse than Proteus in face of real Wikipedia workload. Then we increase the number of virtual nodes for Consistent to $\frac{n^2}{2}$, and the result is shown by the curve with stars. It is better than the $\mathcal{O}(\log n)$ case but is still much worse than Proteus.

### 4.4.2 False Positive and False Negative

The counting Bloom filter is the key feature used as a digest for the Memcached data in the smooth transition phase. Both Memcached server and Bloom filter should be configured carefully, so that the cache achieves high hitting rate and the Bloom filter has low false positive and false negative rate. We apply the real Wikipedia trace [66] to evaluate Bloom filter settings. Fig. 4.5 shows how does the Memcached cache size affect hit ratio. When each Memcached server uses 1GB memory (with 4KB data per page), the hit ratio reaches above 80% (*i.e.*, roughly 2,560,000 pages in cache). With this setting, we proceed to tune

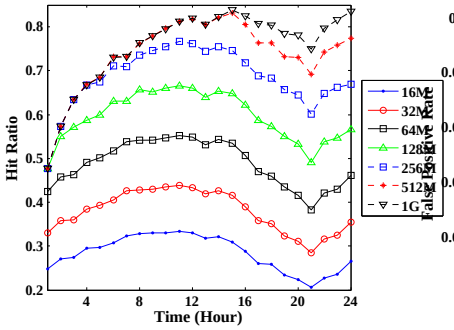



**Figure 4.5:** Hit Ratio vs Cache Size

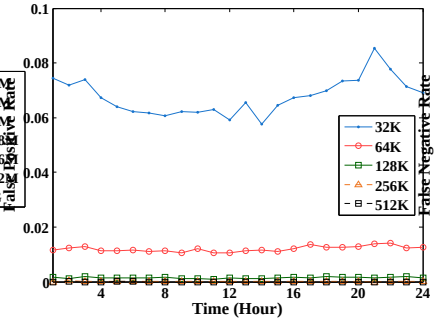



**Figure 4.6:** False Positive vs Bloom Filter Size

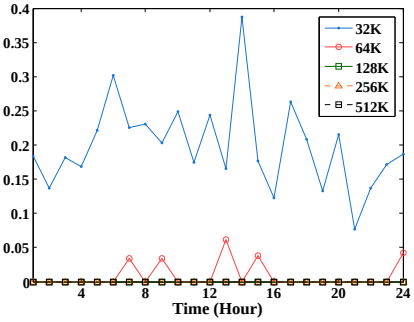



**Figure 4.7:** False Negative vs Bloom Filter Size

the Bloom filter. As we stated before, using more hash functions in the Bloom filter induces higher overhead. Therefore, we choose to use only 4 non-encryption hash functions and tune the Bloom filter size instead. Fig. 4.6 and Fig. 4.7 present the relationship between the Bloom filter size and the false positive/negative ratio. As shown, with 512KB memory, the Bloom filter achieves negligible false positive and false negative rate. So, we set the Bloom filter to use 512KB memory when doing other evaluations.

### 4.4.3 Response Time

In this section, we will evaluate the service response time by applying synthetic workload to the server cluster. The response time is measured on the RBE side, which records the time duration from submitting one HTTP request to receiving the corresponding response. Each RBE server simulates hundreds of independent users with think time 0.5 second. As we have 10 RBE servers, altogether, there will be approximately a few thousand user requests per second. Each user has an independent page set of 50 pages. Every time generating one request, the user thread will choose one page from her page set, and contact the web server to perform server side logic. The user requests will be uniformly randomly directed to all web servers.

Fig. 4.8 shows the experiment result. The recorded response time data are grouped into 480 slots according to physical time. We look at the response time locating at 99.9% percentile. The delay is plotted logarithmically. The curve with squares shows the response

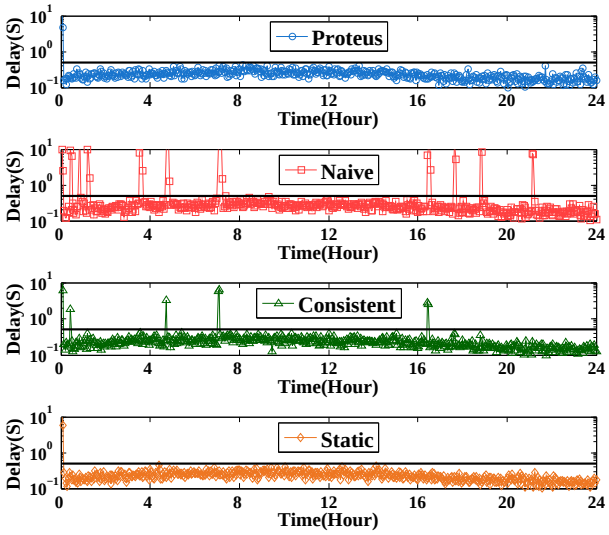



**Figure 4.8:** Response Time

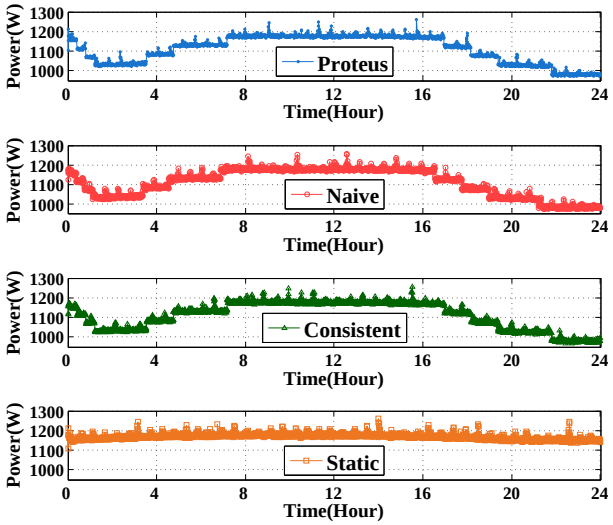



**Figure 4.9:** Power Consumption

time for the Naive solution where the requests are directed by using simple hash and modular operation. Obviously, there is a huge response time spike when the number of running Memcached servers changes. As we explained before, it is because, when the number of running Memcached server changes, the mapping between data key and Memcached server ID changes significantly. A large number of requests will see misses in cache tier and will reach the database tier, hence induce a spike in response time. The curve with triangles depicts the response time when using consistent hashing with exactly $\frac{n^2}{2}$ virtual nodes. The virtual nodes are placed randomly using Java Random class. All web servers share the same random seed (0) to generate virtual node positions. Therefore, the view of all web servers are consistent. The consistent hashing solution shows much better performance during dynamics. But there are still considerable performance degradation during transition. The curve with circles demonstrates the response time when using Proteus. The delay spike is clearly removed, and users see almost no difference during the transition stages. Proteus's performance match what the static solution achieves as shown by the curve with diamonds.

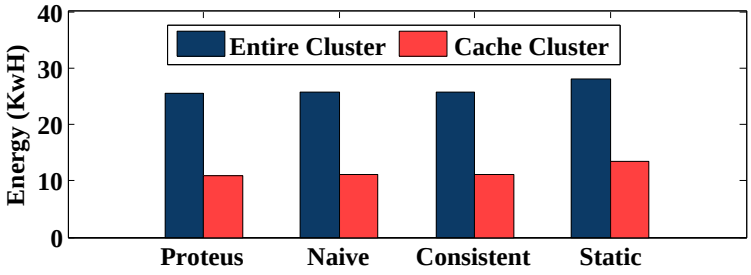



**Figure 4.10:** Total Energy

### 4.4.4 Power Consumption

In previous sections, we have shown that, by using Proteus, the performance degradation induced by dynamic server provisioning is almost eliminated. Now, we compare the power consumption of the four different scenarios. We measure the real power reading by using the Avocent 3000 Power Distribution Unit. The data is sampled every 15 seconds. We take the entire cluster (web servers, cache server, and database servers) into consideration to see total power saving, rather than considering only the Memcached tier. Fig. 4.9 shows the power consumption over time. The Static scenario consumes roughly the same level of power during the whole experiment. The power consumption actually decreases slightly as the workload decreases. However, when compared with the power saving induced by server provisioning (as shown in the other three scenarios), the difference is almost unnoticeable. The curve with small dots shows the power draw of Proteus. It is clear that, Proteus not only eliminates performance degradations, but also saves the same amount of energy compared to Naive and Consistent cases. The total energy consumption during the experiments is shown in Fig. 4.10. The final result is that, with Proteus, we are able to save roughly 10% energy over the entire cluster, and 23% over the cache cluster without delay penalty.

# Chapter 5: Manager For Ordered Partitioned Storage Systems

The popularity of mobile devices is growing at an unprecedented rate. According to the report published by the United Nations International Telecommunication Union [67], mobile penetration rates are now about equal to the global population. Thanks to positioning modules in mobile devices, a great amount of information generated today is tagged with geographic locations. For example, users can share tweets and Instagram images with location information with family and friends; taxi companies collect pick-up and drop-off events data with geographic location information as well. The abundances of geo-tagged data give birth to a whole range of applications that issue spatial-temporal queries. These queries, which we call geometry queries, request information about moving objects within a user-defined geometric area. Despite the urgent need, no existing systems manage to meet both the scalability and efficiency requirements for spatial-temporal data. For example, geospatial databases [68] are optimized for spatial data, but usually fall short on scalability on handling big-data applications, whereas distributed data stores [69–72] scale well but quite often yield inefficiencies when dealing with geometry queries.

Distributed data stores, such as HBase [69], Cassandra [70], and DynamoDB [71], have been widely used for big-data storage applications. Their key distribution algorithms can be categorized into two classes: random partitioning and ordered partitioning. The former randomly distributes keys into servers, while the latter divides the key space into subregions such that all keys in the same subregion are hosted by the same server. Compared to random

partitioning, ordered partitioning considerably benefits range scans, as querying all servers in the cluster can then be avoided. Therefore, existing solutions for spatial-temporal big-data applications, such as MD-HBase [73], and ST-HBase [74], build index layers above the ordered-partitioned HBase to translate a geometry query into a set of range scans. Then, they submit those range scans to HBase, and aggregate the returned data from HBase to answer the query source, inheriting scalability properties from HBase. Although these solutions fulfill the semantic level requirement of spatial-temporal applications, moving hotspots and large geometry queries still cannot be handled efficiently.

Spatial-temporal applications naturally generate moving workload hotspots. Imagine a million people simultaneously whistle taxis after the New Year's Eve at NYC's Times Square. Or during every morning rush hour, people driving into the city central business district search for the least congested routes. Ordered partitioning data stores usually mitigate hotspots by splitting an overloaded region into multiple daughter regions, which can then be moved into different servers. Nevertheless, as region data may still stay in the parent region's server, the split operation prevents daughter regions from enjoying data locality benefits. Take HBase as an example. Region servers in HBase usually co-locate with HDFS datanodes. Under this deployment, one replica of all region data writes to the region server's storage disks, which allows get/scan requests to be served using local data. Other replicas spread randomly in the entire cluster. Splitting and moving a region into other servers disable data locality benefits, forcing daughter regions to fetch data from remote servers. Therefore, moving hotspots often lead to performance degradation.

In this chapter, we present Pyro, a holistic spatial-temporal big-data storage system tailored for high resolution geometry queries and moving hotspots. Pyro consists of PyroDB and PyroDFS, corresponding to HBase and HDFS respectively. This chapter makes three major contributions. First, PyroDB internally implements Moore encoding to efficiently translate geometry queries into range scans. Second, PyroDB aggregately minimizes IO latencies of the multiple range scans generated by the same geometry query using dynamic programming. Third, PyroDFS employs a novel DFS block grouping algorithm that allows

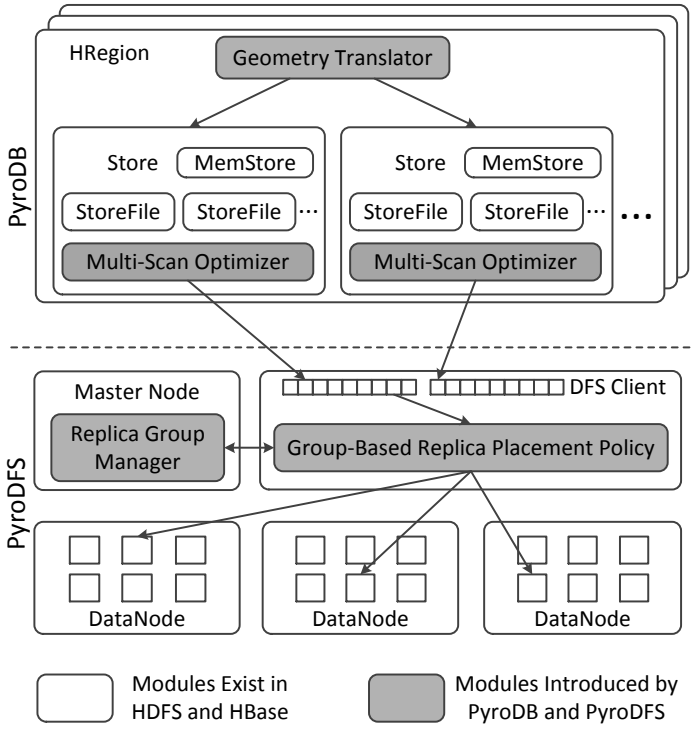



**Figure 5.1:** Pyro Architecture

Pyro to preserve data locality benefits when PyroDB splits regions during hotspots dynamics. Pyro is implemented by adding 891 lines of code into Hadoop-2.4.1, and another 7344 lines of code into HBase-0.99. Experiments using NYC taxi dataset [75, 76] show that Pyro reduces the response time by 60X on $1km \times 1km$ rectangle geometries. Pyro further achieves 10X throughput improvement on $100m \times 100m$ rectangle geometries.

## 5.1 Design Overview

Pyro consists of PyroDB and PyroDFS. The design of PyroDB and PyroDFS are based on HBase and HDFS respectively. Figure 5.1 shows the high-level architecture, where shaded modules are introduced by Pyro.

### 5.1.1 Background

HDFS [14] is an open source software based on GFS [53]. Due to its prominent fame and universal deployment, we skip the background description.

HBase is a distributed, non-relational database running on top of HDFS. Following the design of BigTable [77], HBase organizes data into a 3D table of rows, columns, and cell versions. Each column belongs to a column family. HBase stores the 3D table as a key-value store. The key consists of row key, column family key, column qualifier, and timestamp. The value contains the data stored in the cell.

In HBase, the entire key space is partitioned into regions, with each region served by an HRegion instance. HRegion manages each column family using a *Store*. Each Store contains one MemStore and multiple StoreFiles. In the write path, the data first stays in the MemStore. When the MemStore reaches some pre-defined flush threshold, all key-value pairs in the MemStore are sorted and flushed into a new StoreFile in HDFS. Each StoreFile wraps an HFile, consisting of a series of data blocks followed by meta blocks. In this chapter, we use meta blocks to refer to all blocks that store meta, data index, or meta index. In the read path, a request first determines the right HRegions to query, then it searches all StoreFiles in those regions to find target key-value pairs.

As the number of StoreFiles increases, HBase merges them into larger StoreFiles to reduce the overhead of read operations. When the size of a store increases beyond a threshold, its HRegion splits into two daughter regions, with each region handles roughly half of its parent's key-space. The two daughter regions initially create reference files pointing back to StoreFiles of their past parent region. This design postpones the overhead of copying region data to daughter region servers at the cost of losing data locality benefits. The next major compaction materializes the reference files into real StoreFiles.

HBase has become a famous big-data storage system for structured data [78], including data for location-based services. Many location-based services share the same request primitive that queries information about moving objects within a given geometry, which

we call geometry queries. Unfortunately, HBase suffers inefficiencies when serving geometry queries. All cells in HBase are ordered based on their keys in a one-dimensional space. Casting a geometry into that one-dimensional space inevitably results in multiple disjoint range scans. HBase handles those range scans individually, preventing queries to be aggregately optimized. Moreover, location-based workloads naturally create moving hotspots in the backend, requiring responsive resource elasticity in every HRegion. HBase handles workload hotspots by efficiently splitting regions, which sacrifices data locality benefits for newly created daughter regions. Without data locality, requests will suffer increased response time after splits. Above observations motivate us to design Pyro, a data store specifically tailored for geometry queries.

### 5.1.2 Assumptions

This chapter focuses on spatial-temporal applications, such that each data entry associates with a geo-tag and a timestamp. Data entries are ordered-partitioned into a cluster of servers based on their geographic location tags. The amount of data and requests falling in a certain server varies over time. Based on our observations, we make the assumption that, the request ratio of peak time over nadir time is within four times.

### 5.1.3 Architecture

Figure 5.1 shows the high-level architecture of Pyro. Pyro internally uses Moore encoding algorithm [79–82] to cast two-dimensional data into one-dimensional Moore index, which is enclosed as part of the row key. For geometry queries, the *Geometry Translator* module first applies the same Moore encoding algorithm to calculate scan ranges. Then, the *Multi-Scan Optimizer* computes the optimal read strategy such that the IO latency is minimized. Sections 5.2.1 and 5.2.2 present more details.

Pyro relies on the group-based replica placement policy in PyroDFS to guarantee data locality during region splits. To achieve that, each StoreFile is divided into multiple shards

based on user-defined pre-split keys. Then, Pyro organizes DFS replicas of all shards into elaborately designed groups. Replicas in the same group are stored in the same physical server. After one or multiple splits, each daughter region is guaranteed to find at least one replica of all its region data within one group. To preserve data locality, Pyro just need to move the daughter region into the physical server hosting that group. The details of group-based replica placement are described in section 5.2.3.

Pyro makes three major contributions:

- Geometry Translation: Apart from previous solutions that build an index layer above HBase, Pyro internally implements efficient geometry translation algorithms based on Moore encoding. This design allows Pyro to optimize a geometry query by globally processing all its range scans together.
- Multi-Scan Optimization: After geometry translation, the multi-scan optimizer aggregately processes the generated range scans to minimize the response time of the geometry query. By using storage media performance profiles as inputs, the multi-scan optimizer employs a dynamic programming algorithm to calculate the optimal HBase blocks to fetch.
- Block Grouping: To deal with moving hotspots, Pyro relies on a novel data block grouping algorithm in the DFS layer to split a region quickly and efficiently, while preserving data locality benefits. Moreover, by treating meta block and data block differently, block grouping helps to improve Pyro's fault tolerance.

## 5.2 System Design

We first present the geometry translation and multi-scan optimization in Sections 5.2.1 and 5.2.2 respectively. These two solutions help to efficiently process geometry queries. Then, Section 5.2.3 describes how Pyro handles moving hotspots with the block grouping algorithm.

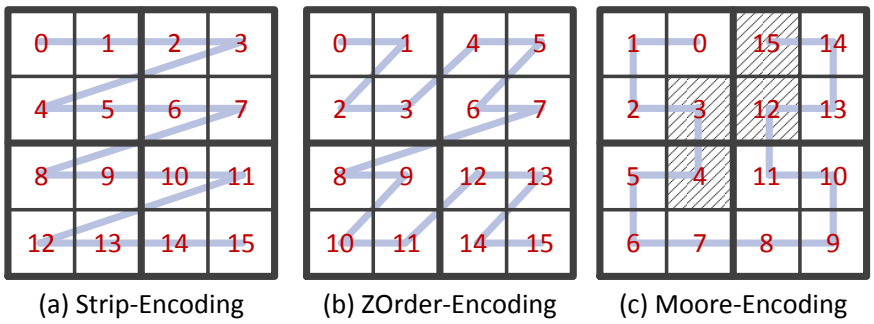



**Figure 5.2:** Spatial Encoding Algorithms of Resolution 2

### 5.2.1 Geometry Translation

In order to store spatial-temporal data, Pyro needs to cast 2D coordinates $(x, y)$ into the one-dimensional key space. A straightforward solution is to use a fixed number of bits to represent $x$, and $y$, and append $x$ after $y$ to form the spatial key. This leads to the *Strip*-encoding as shown in Figure 5.2 (a). Another solution is to use ZOrder-encoding [73] that interleaves the bits of $x$ and $y$. An example is illustrated in Figure 5.2 (b). These encoding algorithms divide the 2D space into $m \times m$ tiles, and index each tile with a unique ID. The tile is the spatial encoding unit as well as the unit of range scans. We define the resolution as $\log_2(m)$, which is the minimum number of bits required to encode the largest value of $x$ and $y$.

In most cases, encoding algorithms inevitably break a two-dimensional geometry into multiple key ranges. Therefore, each geometry query may result in multiple range scans. Each range scan requires a few indexing, caching, and disk operations to process. Therefore, it is desired to keep the number of range scans low. We carry out experiments to evaluate the number of range scans that a geometry query may generate. The resolution ranges from 25 to 18 over the same set of randomly generated disk-shaped geometry queries with $100m$ radius in a $40,000,000m \times 40,000,000m$ area. The corresponding tile size ranges from 1.2m to 153m. Figure 5.3 shows the number of range scans generated by a single geometry query under different resolutions. It turns out that Strip-encoding and ZOrder-encoding translate a single disk geometry to a few tens of range scans when the tile size falls under 20m.

To reduce the number of range scans, we developed the Geometry Translator module. The

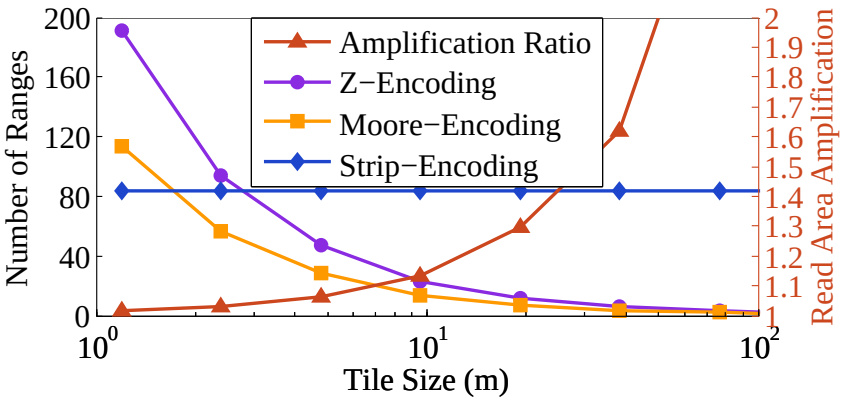



**Figure 5.3:** Translate Geometry to Key Ranges

module employs the *Moore*-Encoding algorithm which is inspired by the Moore curve from the space-filling curve family [79–82]. A simple example is shown in Figure 5.2 (c). A Moore curve can be developed up to any resolution. As shown in Figure 5.4 (a), resolutions 1 and 2 of Moore encoding are special cases. The curve of resolution 1 is called a unit component. In order to increase the resolution, the Moore curve expands each unit component according to a fixed strategy as shown in Figure 5.5. Results plotted in Figure 5.3 show that Moore-Encoding helps to reduce the number of range scans by 40% when compared to ZOrder-Encoding. Moore curves may generalize to higher dimensions [83], Figure 5.4 (b) depicts the simplest 3D Moore curve of resolution 1. Implementations of the Moore encoding algorithm are presented in Section 5.3.

### 5.2.2 Multi-Scan Optimization

The purpose of multi-scan optimization is to reduce read amplification. Below, we first describe the phenomenon of read amplification, and then we present our solution to this problem.

#### 5.2.2.1 Read Amplification

When translating geometry queries, range scans are generated respecting tile boundaries at the given resolution. But, tile boundaries may not align with the geometry query boundary.

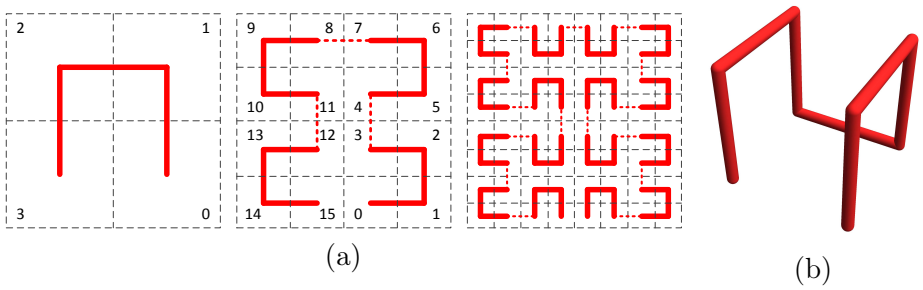



**Figure 5.4:** Moore Curve

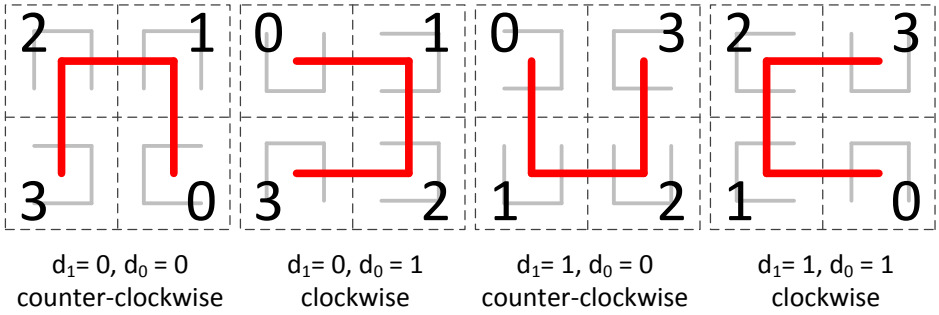



**Figure 5.5:** Moore Encoding Unit

In order to cover the entire geometry, data from a larger area is fetched. We call this phenomenon *Read Area Amplification*. Figure 5.3 plots the curve of read area amplification ratio, which is quantitatively defined as the total area of fetched tiles over the area of the geometry query. The curves show that, solely tuning the resolution cannot achieve both a small number of range scans and a low ratio of read area amplification. For example, as shown in Figure 5.3, restricting each geometry query to generate less than 10 scans forces Pyro to fetch data from a 22% larger area. On the other hand, limiting the area amplification ratio to less than 5% leads to more than 30 range scans per geometry query. The problem gets worse for larger geometries.

Moreover, encoding tiles are stored into fixed-size DB blocks on disks, whereas DB blocks ignore the boundaries of encoding tiles. An entire DB block has to be loaded even when there is only one requested key-value pair fallen in that DB Block, which we call the *Read Volume-Amplification*. Please notice that, DB blocks are different from DFS blocks. DB blocks are the minimum read/write units in PyroDB (similar to HBase). One DB block is usually only a few tens of KiloBytes. In contrast, a DFS block is the minimum replication

unit in PyroDFS (similar to HDFS). DFS blocks are orders of magnitudes larger than DB blocks. For example, the default PyroDFS block size is 64MB, which is 1024 times larger than the default PyroDB block size.

Besides read area and volume amplifications, using a third-party indexing layer may also force the data store to unnecessarily visit a DB block multiple times, especially for high resolution queries. We call it the *Redundant Read Phenomenon*. Even though a DB block can be cached to avoid disk operations, the data store still needs to traverse DB block's data structure to fetch the requested key-value pairs. Therefore, Moore encoding algorithm alone is not enough to guarantee the efficiency.

For ease of presentation, we use the term *Read Amplification* to summarize the read area amplification, read volume amplification, and redundant read phenomena. Read amplification may force a geometry query to load a significant amount of unnecessary data as well as visiting the same DB block multiple times, leading to a much longer response time. In the next section, we present techniques to minimize the penalty of read amplification.

#### 5.2.2.2 An Adaptive Aggregation Algorithm

According to Figure 5.3, increasing the resolution helps to alleviate read area amplification. Using smaller DB block sizes reduces read volume amplification. However, these changes require Pyro to fetch significantly more DB blocks, pushing disk IO to become a throughput bottleneck. In order to minimize the response time, Pyro optimizes all range scans of the same geometry query aggregately, such that multiple DB blocks can be fetched within fewer disk read operations. There are several reasons for considering IO optimizations in the DB layer rather than relying on asynchronous IO scheduling in the DFS layer or the OS layer. First, issuing a DFS read request is not free. As a geometry query may potentially translate into a large number of read operations, maintaining those reads alone elicits extra overhead in all three layers. Second, performance of existing IO optimizations in lower layers depend on the timing and ordering of request submissions. Enforcing the perfect request submission ordering in the Geometry Translator is not any cheaper than directly performing the IO

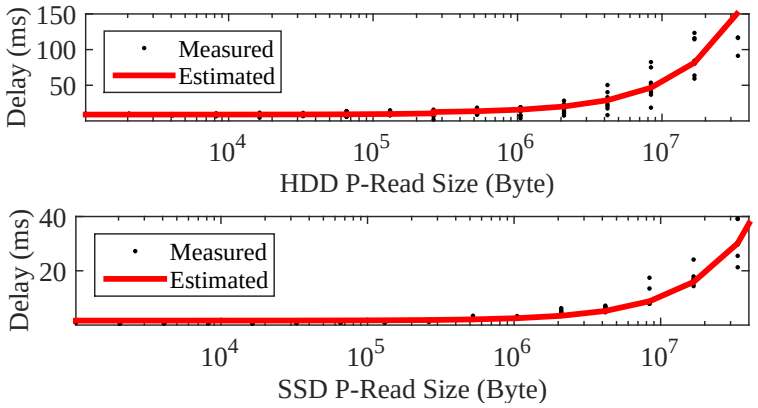



**Figure 5.6:** Storage Media Profile

optimization in PyroDB. Third, as PyroDB servers have the global knowledge about all p-reads from the same geometry request, it is the natural place to implement IO optimizations.

Pyro needs to elaborately tune the trade-off between unnecessarily reading more DB blocks and issuing more disk seeks. Figure 5.6 shows the profiling results of Hadoop-2.4.1 position read (p-read) performance on a 7,200RPM Seagate hard drive and a Samsung SM0256F Solid State Drive respectively. In the experiment, we load a 20GB file into the HDFS, and measure the latency of p-read operations of varies sizes at random offsets. The disk seek delay dominates the p-read response time when reading less than 1MB data. When the size of p-read surpasses 1MB, the data transmission delay starts to make a difference. A naïve solution calculates the disk seek delay and the per-block transmission delay, and directly compares whether reading the next unnecessary block helps to reduce response time. However, the system may run on different data storage media, including hard disk drives, solid state drives, or even remote cloud drives. There is no guarantee that all media share the same performance profile. Such explicit seek delay and transmission delay may not even exist.

In order to allow the optimized range scan aggregation to work for a broader scenarios, we propose the Adaptive Aggregation Algorithm ($A^3$). $A^3$ uses the p-read profiling result to estimate delay of p-read operations. The profiling result contains the p-read response time of various sizes. $A^3$ applies interpolation to fill in gaps between profiled p-read sizes. This

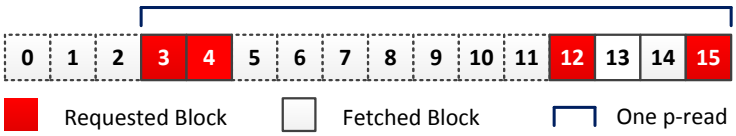



**Figure 5.7:** Block Layout in a StoreFile

design allows the $A^3$ algorithm to work for various storage media.

Before diving into algorithm details, we present the abstraction of the block aggregation problem. Suppose a geometry query hits shaded tiles (3, 4, 12, 15) in Fig 5.2 (c). For the sake of simplicity, assume that DB blocks align perfectly with encoding tiles, one block per tile. Figure 5.7 shows the block layout in the StoreFile. $A^3$ needs to determine what block ranges to fetch in order to cover all requested blocks, such that the response time of the geometry query is minimized. In this example, let us further assume each block is 64KB. According to the profiling result shown in Figure 5.6, reading one block takes about 9 ms, four blocks takes 14 ms, while reading thirteen blocks takes 20 ms. Therefore, the optimal solution reads blocks 3-15 using one p-read operation.

$A^3$ works as follows. Suppose a geometry query translates to a set $\mathbf{Q}$ of range scans. Block indices help to convert those range scans into another set $\mathbf{B}'$ of blocks, sorted in the ascending order of their offsets. By removing all cached blocks from $\mathbf{B}'$, we get set $\mathbf{B}$ of $n$ requested but not cached blocks. Define $\mathbf{S}[i]$ as the estimated minimum delay of loading the first $i$ blocks. Then, the problem is to solve $\mathbf{S}[n]$. For any optimal solution, there must exist a $k$, such that blocks $k$ to $n$ are fetched using a single p-read operation. In other words, $\mathbf{S}[n] = \mathbf{S}[k-1]+\textsc{Estimate}(k,n)$, where $\textsc{Estimate}(k,n)$ estimates the delay of fetching blocks from $k$ to $n$ together based on the profiling result. Therefore, starting from $\mathbf{S}[0]$, $A^3$ calculates $\mathbf{S}[i]$ as $\min\{\mathbf{S}[k-1]+\textsc{Estimate}(k,i)|1 \leq k \leq i\}$. The pseudo code of $A^3$ is presented in Algorithm 5.1.

In Algorithm $A^3$, the nested loop between line $3-7$ leads to $\boldsymbol{O}(|\mathbf{B}|^2)$ computational complexity. If $\mathbf{B}$ is large, the quadratic computational complexity explosion can be easily mitigated by setting an upper bound on the position read size. For example, for the hard drive profiled in Figure 5.6, fetching $10^7$ bytes result in about the same delay as fetching

**Algorithm 5.1** $A^3$ Algorithm

**Input:** blocks to fetch sorted by offset **B**
**Output:** block ranges to fetch **R**

```
1: S ← an array of size |B|; initialize to ∞
2: P ← an array of size |B|; S[0] ← 0
3: for i ← 1 ∼ |B| do
4:     for j ← 0 ∼ i − 1 do
5:         k = i − j;    s ←ESTIMATE(k, i) +S[k − 1]
6:         if s < S[i] then S[i] ← s;    P[i] ← k
7:         end if
8:     end for
9: end for
10: i ← |B|; R ← ∅
11: while i > 0 do
12:     R ← R ∪ (P[i], i);    i ← P[i] − 1
13: end while
14: return R
```

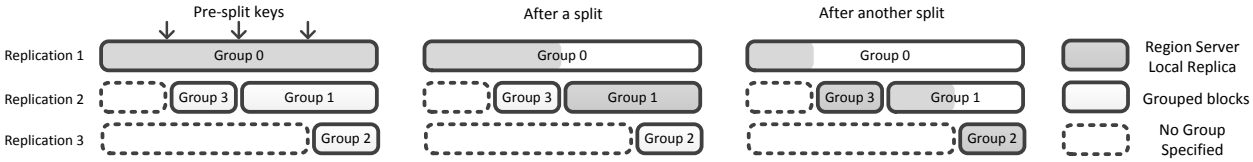



**Figure 5.8:** Split Example

$5 \times 10^6$ bytes twice. Therefore, there is no need to issue position read larger than $5 \times 10^6$ bytes. If block size is set to $64KB$, the variable $j$ on the $5^{th}$ line in Algorithm 5.1 only needs to loop from 0 to 76, resulting in linear computational complexity.

### 5.2.3 Block Grouping

Moore encoding concentrates range scans of one geometry query into fewer servers. This may lead to performance degradation when spatial-temporal hotspots exist. To handle moving hotspots, a region needs to gracefully split itself to multiple daughters to make use of resources on multiple physical servers. Later, those daughter regions may merge back after their workloads shrink.

In HBase, the split operation creates two daughter regions on the same physical server, each owning reference files pointing back to StoreFiles of their parent region. Daughter regions are later moved onto other servers during the next cluster balance operation. Using reference files on one hand helps to keep the split operation light, but on the other hand

prevents daughter regions from taking advantage of data locality benefits. Because, after leaving the parent region's server, the two daughter regions can no longer find their region data in their local disks until daughters' reference files are materialized. HBase materializes reference files during the next major compaction, which executes at a very low frequency (*e.g.*, once a day). Forcing earlier materialization does not solve the problem. It could introduce even more overhead to the already-overwhelmed region, as materialization itself is a heavy operation.

An ideal solution should keep both split and materialization operations light, allowing the system to react quickly when a hotspot emerges. Below, we present our design to achieve such ideal behaviors.

#### 5.2.3.1 Group Based Replica Placement

Same as HBase, Pyro suggests users to perform pre-split based on expected data distribution to gain initial load balancing among region servers. Pyro relies on the expected data distribution to create more splitting keys for potential future splits. Split keys divide StoreFiles into shards, and help to organize DFS block replicas into replica groups. PyroDFS guarantees that DFS blocks respect predefined split keys. To achieve that, PyroDFS stops writing into the current DFS block and start a new one as soon as it reaches a predefined split key. This design relies on the assumption that, although moving hotspots may emerge in spatial-temporal applications, the long-round popularity of different geographic regions changes slowly. Results presented in evaluation Section 5.4.1 confirm the validity of this assumption.

Assume blocks are replicated $r$ times and there are $2^{r-1}-1$ predefined split keys within a given region. Split keys divide the region key space into $2^{r-1}$ shards, resulting in $r \cdot 2^{r-1}$ shard replicas. Group 0 contains one replica from all shards. Other groups can be constructed following a recursive procedure:

1 Let $\Psi$ be the set of all shards. If $\Psi$ contains only one shard, stop. Otherwise, use the

median split key $\kappa$ in $\Psi$ to divide all shards into two sets $A$ and $B$. Keys of all shards in A are larger than $\kappa$, while keys of all shards in B are smaller than $\kappa$. Perform step 2, and then perform step 3.

2 Create a new group to contain one replica from all shards in set A. Then, let $\Psi \leftarrow A$, and recursively apply step 1.

3 Let $\Psi \leftarrow B$, and then recursively apply step 1.

Replicas in the same group are stored in the same physical server, whereas different groups of the same region are placed into different physical servers. According to the construction procedure, group 1 starts from the median split key, covering the bottom half of the key space (*i.e.*, $2^{r-2}$ shards). Group 1 allows half of the regions workload to be moved from group 0's server to group 1's server without sacrificing data locality. Figure 5.8 demonstrates an example of $r = 3$. PyroDFS is compatible with normal HDFS workload whose replicas can be simply set as no group specified. Section 5.2.3.2 explains why group 1 and 2 are placed at the end rather than in the beginning of the StoreFile.

Figure **??** also shows how Pyro makes use of DFS block replicas. The shaded area indicates which replica serves workloads falling in that key range. In the beginning, there is only one region server. Replicas in group 0 take care of all workloads. As all replicas in group 0 are stored locally in the region's physical server, data locality is preserved. After one split, the daughter region with smaller keys stays in the same physical server, hence still enjoys data locality. Another daughter region moves to the physical server that hosts replica group 1, which is also able to serve this daughter region using local data. Subsequent splits are carried out under the same fashion.

To distinguish from the original split operation in HBase, we call the above actions the *soft* split operation. Soft splits are designed to mitigate moving hotspots. Daughter regions created by soft splits eventually merge back to form their parent regions. The efficiency of the merge operation is not a concern as it can be performed after the hotspot moves out of that region. Please notice that the original split operation, which we call the *hard* split,

is still needed when a region grows too large to fit in one physical server. As this chapter focuses on geometry query and moving hotspots, all splits in the following sections refer to soft splits.

#### 5.2.3.2 Fault Tolerance

As a persistent data store, Pyro needs to preserve high data availability. The block grouping algorithm presented in the previous section affects DFS replica placement schemes, which in turn affects Pyro's fault tolerance properties. In this section, we show that the block grouping algorithm allows Pyro to achieve higher data availability compared to the default random replica placement policy in HDFS.

Pyro inherits the same HFile format [69] from HBase to store key-value pairs. According to HFile Format, meta blocks are stored at the end of the file. Losing any DFS block of the meta may leave the entire HFile unavailable, whereas the availability of key-value DFS blocks are not affected by the availability of other key-value DFS blocks. This property makes the last shard of the file more important than all preceding shards. Therefore, we choose two different objectives for their fault tolerance design.

- Meta shard: Minimize the probability of losing any DFS block.
- Key-value shard: Minimize the expectation of the number of unavailable DFS blocks.

Assume there are $n$ servers in the cluster, and $f$ nodes are unavailable during a cluster failure event. For a given shard, assume it contains $b$ blocks, and replicates $r$ times, where $g$ out of $r$ replications are grouped. PyroDFS randomly distributes the grouped $g$ replications into $g$ physical servers. The remaining $(r-g)b$ block replicas are randomly and exclusively distributed in the cluster. If the meta fails, it must be the case that the $g$ servers hosting the $g$ grouped replications all fail (*i.e.*, $\binom{f}{g}/\binom{n}{g}$), and at least one block in each $r-g$ ungrouped replications fails. Hence, the probability of meta failure is

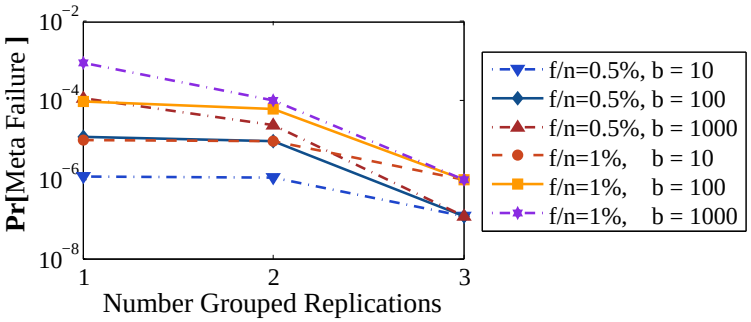



**Figure 5.9:** Unavailability Probability

$$\mathbf{Pr}\left[\text{meta failure}\right] = \frac{\binom{f}{g}}{\binom{n}{g}}\left(1-\left(1-\frac{\binom{f-g}{r-g}}{\binom{n-g}{r-g}}\right)^{b}\right). \tag{5.1}$$

Figure 5.9 plots how the number of grouped replications $g$ affects the failure probability. In this experiment, $n$ and $r$ are set to 10,000, and 3 respectively. According to [84–86], after some power outage, 0.5%-1% of the nodes fail to reboot. Hence, we vary $f$ to be 50, and 100. The results show that the meta failure probability decreases when $g$ increases. Pyro sets $g$ to the maximum value for the meta shard, therefore achieves higher fault tolerance compared to default HDFS where $g$ equals 1.

For key-value shards, transient and small-scale failures are tolerable, as they do not affect most queries. It is more important to minimize the scale of the failure (*i.e.*, the number of unavailable DB blocks). The expected failure scale is,

$$\mathbf{E}\left[\text{failure scale}|\text{failure occurs}\right] = \frac{b\binom{f-g}{r-g}}{\binom{n-g}{r-g}}. \tag{5.2}$$

The failure scale decreases with the increase of grouped replication number $g$. Therefore, placing replica groups 1 and 3 at the end of the StoreFile minimizes both the meta shard failure probability and the failure scale of key-value shards.

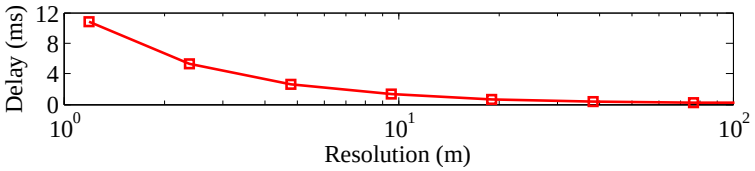



**Figure 5.10:** Geometry Translation Delay

## 5.3 Implementation

PyroDFS and PyroDB are implemented based on HDFS-2.4.1 and HBase-0.99 respectively.

### 5.3.1 Moore Encoding

As previously shown in Figure 5.4 and Figure 5.5, each unit of Moore curve can be uniquely defined by the combination of its orientation (north, east, south, and west) and its rotation (clockwise, counter-clockwise). Encode the orientation with 2 bits, $d1$ and $d0$, such that 00 denotes north, 01 east, 10 south, and 11 west. With more careful observations, it can be seen that the rotation of a Moore curve component unit completely depends on its orientation. Starting from the direction shown in Figure 5.4 (a), the encodings in east and west oriented units rotate clockwise, and others rotate counter-clockwise.

With a given integer coordinate $(x, y)$, let $x_k$ and $y_k$ denote the $k^{th}$ lowest bits of $x$ and $y$ in the binary presentation. Let $d_{k,1}d_{k,0}$ be the orientation of the component unit defined by the highest $r-k-1$ bits in $x$, and $y$. Then, the orientation $d_{k-1,1}d_{k-1,0}$ can be determined based on $d_{k,1}$, $d_{k,0}$, $x_k$, and $y_k$ [79–82].

$$
\begin{aligned}
d_{k-1,0} &= \bar{d}_{k,1}\bar{d}_{k,0}\bar{y}_k \;\mid\; \bar{d}_{k,1}d_{k,0}x_k \\
&\mid\; d_{k,1}\bar{d}_{k,0}y_k \;\mid\; d_{k,1}d_{k,0}\bar{x}_k && (5.3)\\
&= \bar{d}_{k,0}\left(d_{k,1}\oplus\bar{y}_k\right) \;\mid\; d_{k,0}\left(d_{k,1}\oplus x\right) && (5.4)\\
d_{k-1,1} &= \bar{d}_{k,1}\bar{d}_{k,0}x_k\bar{y}_k \;\mid\; \bar{d}_{k,1}d_{k,0}\bar{x}_k y_k \\
&\mid\; d_{k,1}\bar{d}_{k,0}\overline{\bar{x}_k y_k} \;\mid\; d_{k,1}d_{k,0}\overline{x_k\bar{y}_k} && (5.5)\\
&= d_{k,1}\left(\bar{x_k}\oplus y_k\right) \;\mid\; \left(x_k\oplus y_k\right)\left(d_0\oplus x_k\right) && (5.6)
\end{aligned}
$$

The formula considers all situations where $d_{k-1,0}$ and $d_{k-1,1}$ should equal to 1, and uses a

logic *or* to connect them all. For example, the term $\bar{d}_{k,1}\bar{d}_{k,0}\bar{y}_k$ states that when the previous orientation is north ($\bar{d}_{k,1}\bar{d}_{k,0}$), the current unit faces east or west ($d_{k-1,0} = 1$) if and only if $y_k = 0$. The same technique can be applied to determine the final Moore encoding index $\omega$.

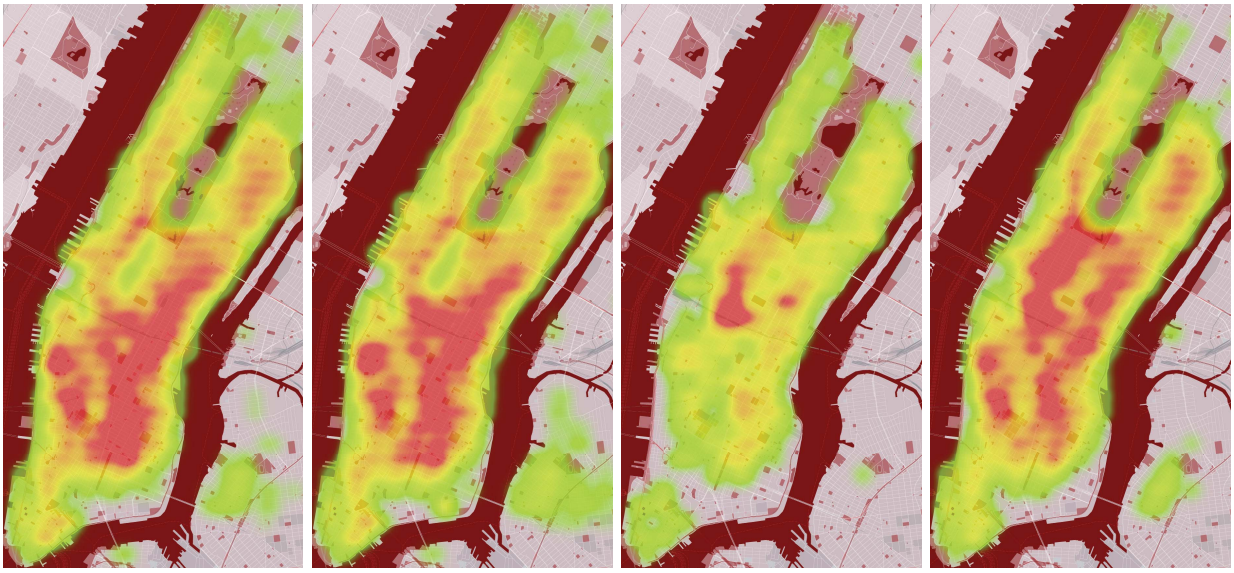
$$\begin{aligned}
\omega_{2k+1} &= \bar{d}_{k,1}\bar{d}_{k,0}\bar{x}_k \mid \bar{d}_{k,1}d_{k,0}\bar{y}_k \\
&\mid d_{k,1}\bar{d}_{k,0}x_k \mid d_{k,1}d_{k,0}y_k && (5.7) \\
&= \bar{d}_{k,0}\left(d_{k,1} \oplus \bar{x}_k\right) + d_{k,0}\left(d_{k,1} \oplus \bar{y}_k\right) && (5.8) \\
\omega_{2k} &= \bar{x}_k \oplus y_k && (5.9)
\end{aligned}$$

Then, each geometry can be translated into range scans using a quad tree. Each level in the quad tree corresponds to a resolution level. Each node in the tree represents a tile, which is further divided into four smaller tiles in the next level.

The translating algorithm only traverses deeper if the geometry query partially overlaps with that area. If an area is fully covered by the geometry, there is no need to go further downwards. Figure 5.10 shows the delay of translating a $5km \times 5km$ square geometry. The delay stays below 11ms even using the finest resolution.

### 5.3.2 Multi-Scan Optimization

After converting a geometry query into range scans, the multi-scan optimizer needs two more pieces of information to minimize the response time: 1) storage media performance profiles, and 2) the mapping from key ranges to DB blocks. For the former one, an administrator may specify an HDFS path under the property name *hbase.profile.storage* in the hbase-site.xml configuration file. This path should point to a file containing multiple lines of (p-read size, p-read delay) items, indicating the storage media performance profile result. Depending on storage media types in physical servers, the administrator may set the property *hbase.profile.storage* to different values for different HRegions. The file will be loaded during HRegion initialization phase. For the latter one, HBase internally keeps indices of DB blocks. Therefore, Pyro can easily translate a range scan into a serious of block starting offsets and



(a) (b) (c) (d)

**Figure 5.11:** Manhattan Taxi Pick-up/Drop-off Hotspots

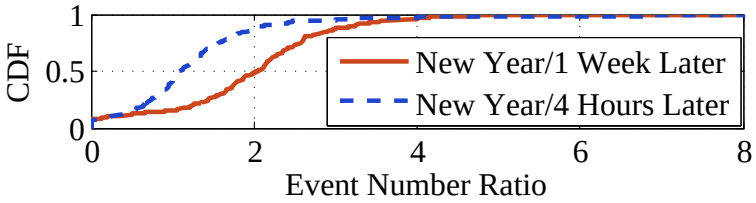



**Figure 5.12:** Workload Heat Range

block sizes. Then, those information will be provided as inputs for the $A^3$ algorithm.

#### 5.3.2.1 Block Grouping

Distributed file systems usually keep replica placement policies as an internal logic, maintaining a clean separation between the DFS layer and higher layer applications. This design, however, prevents Pyro from exploring opportunities to make use of DFS data replications. Pyro carefully breaks this barrier by exposing a minimum amount of control knobs to higher layer applications. Through these APIs, applications may provide *replica group* informa-


**Figure 5.13:** Row Key

tion when writing data into DFS. It is important to choose the right set of APIs such that PyroDFS applications do not need to reveal too much about details in the DFS layer. At the same time, applications are able to fully make use of data locality benefits of all block replicas.

In our design, PyroDFS exposes two families of APIs which help to alter its internal behavior.

- *Sealing a DFS Block:* PyroDB may force PyroDFS to seal the current DFS block and start writing into a new DFS block, even if the current DFS block has not reached its size limit yet. This API is useful because DFS block boundaries may not respect splitting keys, especially when there are many StoreFiles in a region and the sizes of StoreFiles are about the same order of magnitude of the DFS block size. The *seal* API family will help StoreFiles to achieve full data locality after splits.

- *Grouping Replicas:* PyroDB may specify *replica namespace* and *replica groups* when calling the *write* API in PyroDFS. This usually happens during MemStore flushes and StoreFile compactions. Under the same namespace, replicas in the same replica group will be placed into the same physical server, and replicas in different groups will be placed into different physical servers. If there are not enough physical servers or disk spaces, PyroDFS works in a best effort manner. The mapping from the replica group to the physical server and corresponding failure recovery is handled within PyroDFS. PyroDB may retrieve a physical server information of a given replica group using *grouping* APIs, which allows PyroDB to make use of data locality benefits.

## 5.4 Evaluation

Evaluations use NYC taxi dataset [75, 76] that contains GPS pickup/dropoff location information of 697,622,444 trips from 2010 to 2013. The experiments run on a cluster of 80 Dell servers (40 Dell PowerEdge R620 servers and 40 Dell PowerEdge R610 servers) [35, 87–94].

The HDFS cluster consists of 1 master node and 30 datanodes. The HBase server contains 1 master node, 3 zookeeper [95] nodes, and 30 region servers. Region servers are co-located with data nodes. Remaining nodes follow a central controller to generate geometry queries and log response times, which we call Remote User Emulators (RUE).

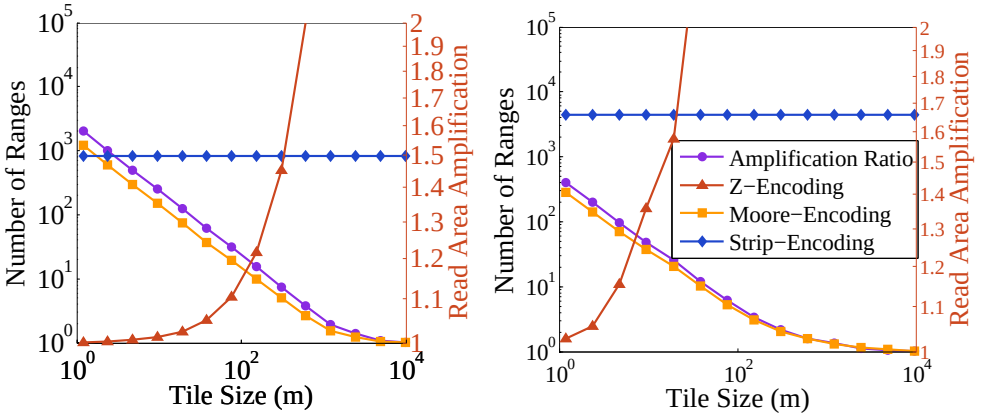
We first briefly analyze the NYC taxi dataset. Then, Sections 5.4.2, 5.4.3, and 5.4.4 evaluate the performance improvements contributed by Geometry Translator, Multi-Scan Optimizer, and Group-based Replica Placement respectively. Finally, in Section 5.4.5, we measure the overall response time and throughput of Pyro.

### 5.4.1 NYC Taxi Data Set Analysis

Moving hotspot is an important phenomenon in spatial-temporal data. Figure 5.11 (a) and (b) illustrate the heat maps of taxi pick-up and drop-off events in the Manhattan area during a 4 hour time slot starting from 8:00PM on December 31, 2010 and December 31, 2012 respectively. The comparison shows that the trip distribution during the same festival does not change much over the years. Figure 5.11 (c) plots the heat map of the morning (6:00AM-10:00AM) on January 1st, 2013, which drastically differs from the heat map shown in Figure 5.11 (b). Figure 5.11 (d) illustrates the trip distribution from 8:00PM to 12:00AM on July 4th, 2013, which also considerably differs from that of the New Year Eve in the same year.

Figures 5.11 (a)-(d) demonstrate the distribution of spatial-temporal hotspots. It is important to understand by how much hotspots cause event count to increase in a region. We measure the increase as the ratio, $\frac{\text{event count during peak hours}}{\text{event count during normal hours}}$.The CDF on 16X16 Manhattan area is shown in Figure 5.12. Although hotspots move over time, the event count of a region changes within a reasonably small range. During New Year midnight, popularity of more than 97% regions grow within four folds.

When loading the data into HBase, both spatial and temporal information contribute to the row key. The encoding algorithm translates the 2D location information of an event into




(a) radius = $1000m$ (b) $50m \times 628m$

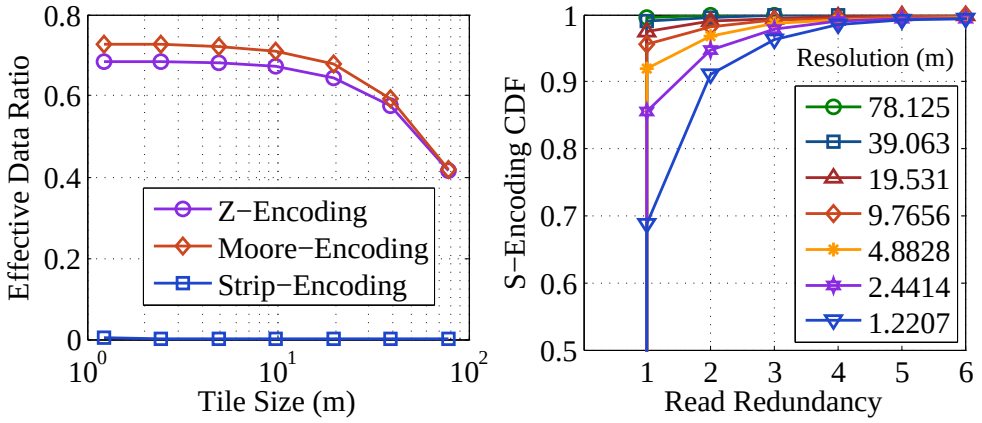
**Figure 5.14:** Reducing the Number of Range Scans

a 32-bit spatial-key, which acts as the suffix of the row key. Then, the temporal strings are parsed to Linux 64-bit timestamps. We use the most significant 32 bits as the temporal-key. Each temporal key represents roughly a 50-day time range. Finally, as shown in Figure 5.13, the temporal-key is concatenated in front of the spatial key to form the complete row key.

### 5.4.2 Moore Encoding

Figure 5.14 shows how much Moore encoding helps to reduce the number of range scans at different resolutions when translating geometry queries in a $40,000,000m \times 40,000,000m$ area. Figures 5.14 (a) and (b) uses disk geometry and rectangle geometries respectively. The two figures share the same legend. For disk geometries, Moore encoding generates 45% fewer range scans when compared to ZOrder-encoding. When a long rectangle is in use, Moore encoding helps to reduce the number of range scans by 30%.

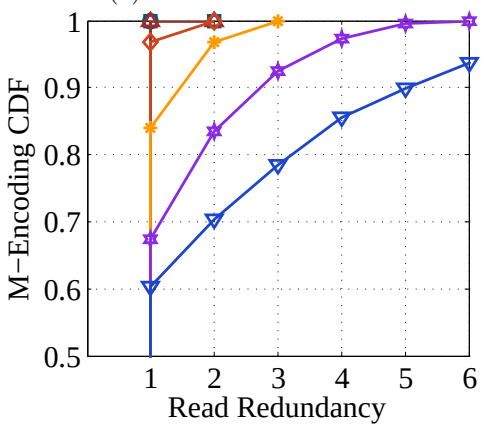
To quantify the read volume amplification, we encode the dataset coordinates with Moore encoding algorithm using the highest resolution shown in Figure 5.3, and populate the data using 64KB DB Blocks. Then, the experiment issues $1Km \times 1Km$ rectangle geometries. Figure 5.15 (a) shows the ratio of fetched key-value pairs volume over the total volume of accessed DB Blocks, which is the inverse of read volume amplification. As the Strip-encoding results in very high read volume amplification, using the inverse helps to limit



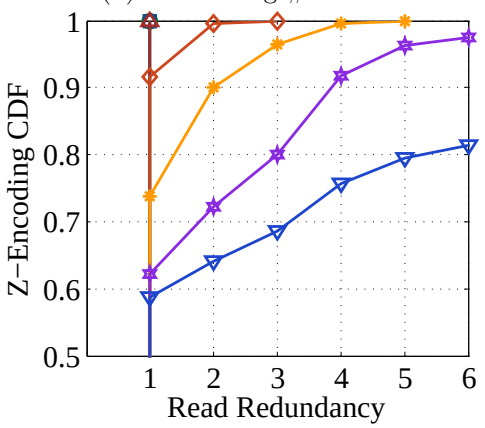

(a) Effective Data Ratio




(b) S-Encoding # of Hits




(c) Z-Encoding # of Hits




(d) M-Encoding # of Hits

**Figure 5.15:** Read Amplification Phenomenon

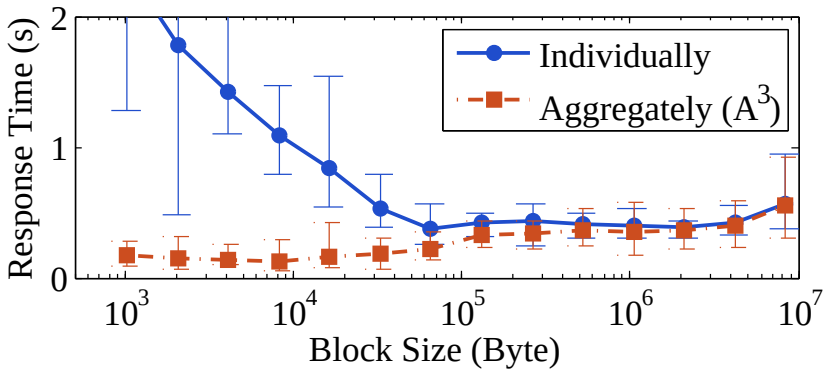



**Figure 5.16:** Block Read Aggregation

the result in interval [0, 1]. Therefore, readers can easily distinguish the difference between Moore-encoding and ZOrder-encoding. We call the inverse metric the effective data ratio. As Moore encoding concentrates a geometry query into fewer range scans, and hence fewer range boundaries, it also achieves higher effective data ratio.

Figures 5.15 (b)-(d) plot the CDFs of redundant read counts when processing the same geometry query.It is clear that the number of redundant reads increases when using higher resolutions. Another observation is that, Moore-encoding leads to large read redundancy. Thanks to the multi-scan optimization design, this will not be a problem, as all redundant reads will be accomplished within a single DB block traverse operation.

### 5.4.3 Multi-Scan Optimization

In order to measure how $A^3$ algorithm works, we load data from the NYC taxi cab dataset using Moore encoding algorithm, and force all StoreFiles of the same store to be compacted into one single StoreFile. Then, the RUE generates $1Km \times 1Km$ rectangle geometry queries with the query resolution set to 13. We measure the internal delay of loading requested DB blocks individually versus aggregately.

The evaluation results are presented in Figure 5.16. The curves convey a few interesting observations. Let us look at the $A^3$ curve first. In general, this curve rises as the block size increases, which agrees with our intuition as larger blocks lead to more severe *read volume amplification*. The minimum response time is achieved at 8KB. Because the minimum data

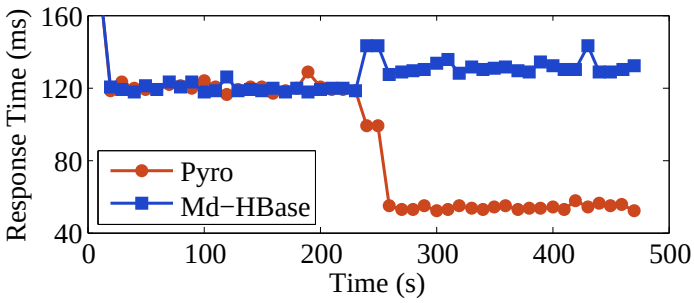



**Figure 5.17:** Response Time at Splitting Event

unit of the disk under test is 4KB, further decreasing block size does not help any more. On the computation side, using smaller block size results in larger input scale for the $A^3$ algorithm. That explains why the response time below 8KB slightly goes up as the block size decreases. The "*individually*" curve monotonically decreases when the block size grows from 1KB to 100KB. It is because increasing block size significantly reduces the number of disk seeks when the block is small. When the block size reaches between 128KB and 4MB, two facts become true: 1) key-value pairs hit by a geometry query tend to concentrate in less blocks; 2) data transmission time starts to make impacts. The benefits of reducing the number of disk seeks and the penalties of loading DB blocks start to cancel each other, leading to a flat curve. After 4MB, the data transmission delay dominates the response time, and the curve rises again. Comparing the nadirs of the two curves concludes that $A^3$ helps to reduce the response time by at least 3X.

### 5.4.4 Soft Region Split

To measure the performance of *soft* splitting, this experiment uses normal scan queries instead of geometry queries, excluding the benefits of Moore encoding and multi-scan optimization. A table is created for the NYC's taxi data, which initially splits into 4 regions. Each region is assigned to a dedicated server. The HBASE_HEAPSIZE parameter is set to 1GB, and the MemStore flush size is set to 256MB. Automatic region split is disabled to allow us to manually control the timing of splits. Twelve RUE servers generate random-sized small scan queries.

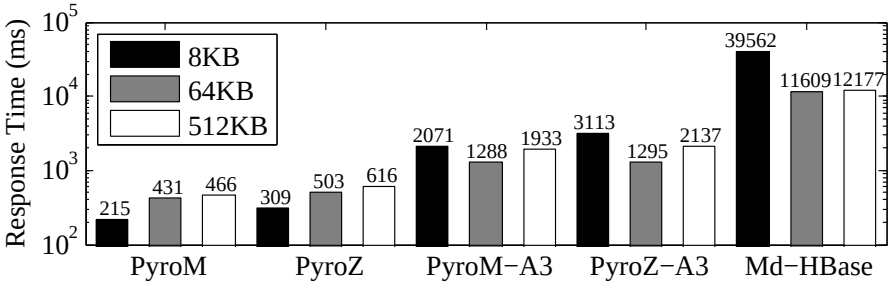



**Figure 5.18:** 1Km×1Km Geometry Response Time

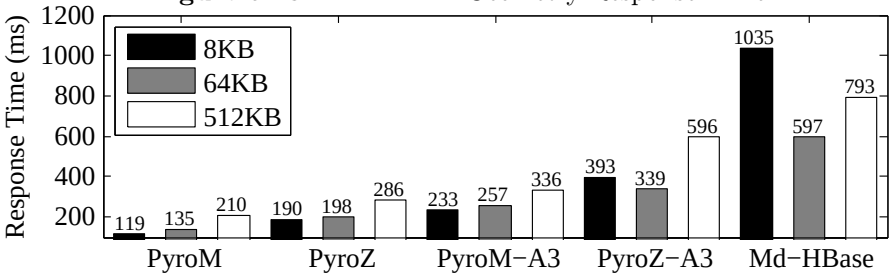



**Figure 5.19:** 100m×100m Geometry Response Time

Figure 5.17 shows the result. The split occurs at the $240^{th}$ second. After the split operation, HBase suffers from even longer response time. It is because daughter region B does not have its region data in its own physical server, and has to fetch data from remote servers, including the one hosting daughter region A. When the group based replication is enabled, both daughter regions read data from local disks, reducing half of the pressure on disk, cpu, and network resources.

### 5.4.5 Response Time and Throughput

We measure the overall response time and throughput improved by Pyro compared to the state-of-the-art solution MD-HBase. Experiments submit rectangle geometry queries of size $1km \times 1km$ and $100m \times 100m$ to Pyro and MD-HBase. The request resolutions are set to 13 and 15 respectively for two types of rectangles. The block sizes vary from 8KB to 512KB. When using MD-HBase, the remote query emulator initiates all scan queries sequentially using one thread. This configuration tries to make the experiment fair, as Pyro uses a single thread to answer each geometry query. Besides, experiments also show how Pyro performs when using ZOrder-encoding or/and $A^3$ algorithm. Figures 5.18 and 5.19 plot experiment

results. The legend on the upper-left corner shows the mapping from colors to block sizes. PyroM and PyroZ represent Pyro using Moore- and ZOrder- encoding respectively. PyroM-$A^3$ and PyroZ-$A^3$ correspond to the cases with the $A^3$ algorithm disabled.

When using PyroM and PyroZ, the response times grow with the increase of block size regardless of whether the rectangle geometry is large or small. It is because larger blocks weaken the benefits of block aggregation and force PyroM and PyroZ to read more data from disk. After disabling $A^3$, the response time rises by 6X for $1km \times 1km$ rectangles, and 2X for $100m \times 100m$ rectangles. MD-HBase achieves the shortest response time when using 64KB DB blocks, which is 60X larger compared to PyroM and PyroZ when handling $1km \times 1km$ rectangle geometries. Reducing the rectangle size to $100m \times 100m$ shrinks the gap to 5X. An interesting phenomenon is that using 512KB DB blocks only increases the response time by 5% compared to using 64KB DB blocks, when the request resolution is set to 13. However, the gap jumps to 33% if the resolution is set to 15. The reason is that, higher resolution leads to more and smaller range scans. In this case, multiple range scans are more likely to hit the same DB block multiple times. According to HFile format, key-value pairs are chained together as a linked-list in each DB block. HBase has to traverse the chain from the very beginning to locate the starting key-value pair for every range scan. Therefore, larger DB block size results in more overhead on iterating through the key-value chain in each DB block.

Figure 5.20 shows the throughput evaluation results of the entire cluster. Pyro regions are initially partitioned based on the average pick up/drop off event location distribution over the year of 2013. Literature [75] presents more analysis and visualizations of the dataset. During the evaluation, each RUE server maintains a pool of emulated users who submit randomly located $100m \times 100m$ rectangle geometry queries. The reason of using small geometries in this experiment is that MD-HBase results in excessively long delays when handling even a single large geometry. The distribution of the rectangle geometry queries follows the heat map from 8:00PM to 11:59PM on December 31, 2013. The configuration mimics the situation where an application only knows the long-term data distribution, and is unable to predict hotspot

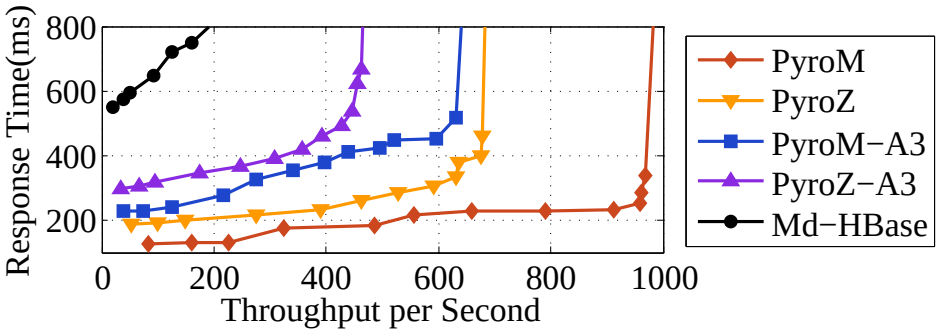



**Figure 5.20:** 100m×100m Geometry Throughput

bursts. When setting 600ms to be the maximum tolerable response time, Pyro outperforms MD-HBase by 10X.

## 5.5 Related Work

As the volume of spatial-temporal data is growing at an unprecedented rate, pursing a scalable solution for storing spatial-temporal data has become a common goal shared by researchers from both the distributed system community and the database community. Advances on this path will benefit a great amount of spatial-temporal applications and analytic systems.

Traditional relational databases understand high dimensional data well [81, 82, 96, 97] due to extensively studied indexing techniques, such as $R$-Tree [98], $Kd$-Tree [99], $UB$-Tree [100, 101], etc. Therefore, researchers seek approaches to improve the scalability. Wang *et al.* [102] construct a global index and local indices using Content Addressable Network [103]. The space is partitioned into smaller subspaces. Each subspace is handled by a local storage. The global index manages subspaces, and local indices manage data points in their own subspaces. Zhang *et al.* [104] propose a similar architecture using $R$-tree as global index and $Kd$-tree as local indices.

From another direction, distributed system researchers push scalable NoSQL stores [69–72, 77, 105–107] to better understand high dimensional data. Distributed key-value stores can be categorized into two classes. One class uses random partition to organize keys. Such

systems include cassandra [70], DynamoDB [71], etc. Due to the randomness on key distribution, these systems are immune to dynamic hotspots concentrated in a small key range. However, spatial-temporal data applications and analytic systems usually issue geometry queries, which translate to range scans. Random partitioning cannot handle range scans efficiently, as it cannot extract all keys within a range with only the range boundaries. Consequently, each range scan needs to query all servers. Other systems, such as BigTable [77], HBase [69], couchDB [108], use ordered partitioning algorithms. In this case, the primary key space is partitioned into regions. The benefits are clear. As data associated with a continuous primary key range are also stored consecutively, sorted partitioning helps to efficiently locate the servers that host the requested key range.

The benefits of ordered partitioning encouraged researchers to mount spatial-temporal application onto HBase. Md-HBase [73] builds an index layer on top of HBase. The index layer encodes spatial information of a data point into a bit series using ZOrder-encoding. Then, a row using that bit series as key is inserted into HBase. The ST-HBase [74] develops a similar technique. However when serving geometry queries, the index layer inevitably translates each geometry query into multiple range scans, and prevents data store from aggregately minimizing the response time.

As summarized above, existing solutions either organize multiple relational databases together using some global index, or build a separate index layer above some general purpose distributed data stores. This chapter, however, takes a different path by designing and implementing a holistic solution that is specifically tailored for spatial-temporal data.

# Chapter 6: Manager For Distributed Computing Clusters

In the past few years, in-memory computing frameworks [16–18, 109–116] have made big-data analytics fast, supercharging a rapidly increasing number of applications in the fields of social networks, e-commerce, finance, and telecommunications [117]. Spark [18], as the state-of-the-art in-memory computing system, attracts a tremendous amount of attention from both academia and industry. Many Spark applications operate on dynamic collections of datasets. For example, an advertising optimization system [118] may store user browsing histories into a dataset every hour, and execute algorithms using data of the past few hours. An IT administrator may dynamically load and evict various system log datasets for diagnosis [119, 120], and run interactive queries on subsets of those datasets. Another example is Spark Streaming [16], which divides stream data into timesteps, and relies on batch-process functionality of Spark core to operate on multiple timesteps within a time window. These applications require Spark core to not only process a single dataset efficiently, but also excel at computations across a dynamic collection of datasets.

Spark delivers high efficiency when tasks are scheduled to servers with all input data cached in local RAM. Violating this data locality condition would force Spark to access disk and network to construct and load data into memory, leading to deteriorated delay. Default Spark relies on the delay scheduling policy [121] to preserve data locality, which allows a task at front of the queue to wait for a small amount of time if its data-local servers are all busy. Although this policy may achieve high data locality probability for applications

dealing with a single dataset, data *co-locality* may still be a far less likely contingency when applications work on a collection of datasets. Data co-locality refers to the property that multiple input datasets are partitioned using the same scheme and cached in the same set of servers correspondingly. Without proper management, a data-local execution slot may not even exist, as multiple input data partitions of the same task can fall into different servers, leaving no chance for the scheduling policy to chase the co-locality property. Moreover, even if the system preserves the co-locality property, time-varying data volume and distribution may result in excessive data re-partitions and transfers, which would inevitably slow down the entire job.

In this chapter, we present Stark, a system specifically designed for optimizing in-memory computing on dynamic dataset collections. Stark achieves data co-locality by judiciously managing partitioning strategies and data placement policies. There are three major contributions. First, Stark allows applications to preserve data partitioning strategy across a collection of datasets, and arranges corresponding partitions into the same set of physical servers (*i.e.*, co-locality), avoiding huge data shuffling overheads when processing multiple datasets. Second, Stark handles time-varying data volume and distribution by delivering elasticity into partitions, such that partitions may split or merge without re-partitioning the entire dataset collection. Third, Stark achieves bounded failure recovery delay with minimum checkpointing overhead. Stark is implemented based on Spark-1.3.1 by adding 2.9K lines of Scala code. Experiments on a 50-server cluster show that Stark reduces the delay by 4X and improves the throughput by 6X compared to Spark.

## 6.1 Design Overview

This section first provides a high level explanation of how Spark works in 6.1.1, and then discusses stark's architecture design in 6.1.2.

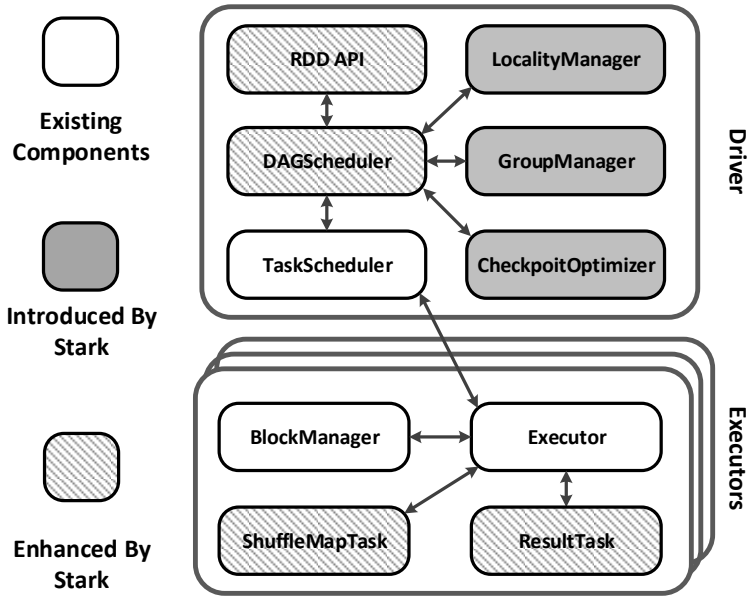



**Figure 6.1:** Stark Architecture

### 6.1.1 Spark Background

Spark [16–18] is a renowned open-source in-memory computing system that offers both streaming and batch processing capabilities. It has attracted immense attentions from both industry and academia. This section explains two of its major components, Spark Core [18] and Spark Streaming [16].

A Spark job involves a driver program, a cluster manager, and a set of worker nodes. The driver program takes care of the task scheduling of the job. Cluster manager monitors and manages worker nodes who host executors to run tasks and cache data partitions.

Spark Core is the cornerstone of the entire project. Its fundamental programming abstraction is Resilient Distributed Datasets (RDD)[1], an immutable logical collection of data partitioned across machines. RDDs can be created from importing external data or applying transformations on existing RDDs. Transformations indicate dependency relationships between RDDs, connecting RDDs into a Directed Acyclic Graph (DAG) which is called the lineage graph. The immutability allows RDDs to be recomputed when corrupted or lost

[1] *Dataset* and *RDD* are used interchangeably in this chapter.

using only the lineage information and data of any cut on the DAG. Spark Core materializes RDDs in a lazy fashion, such that an RDD will only be materialized if some action requires its descendants or itself. Spark supports two types of transformation dependencies, narrow and wide, judged by whether the transformation shuffles data to alter the partitioning strategy. A chain of narrowly dependent transformations is packed into a single stage. Barriers between the map and the reduce phases in wide transformations form stage boundaries. For the sake of accelerating failure recovery, shuffle maps always commit outputs into persistent storage (*e.g.*, HDFS). Spark Core achieves in-memory batch processing that may reduce the response time down to milliseconds.

Spark Streaming, stacked on top of Spark Core, works as a micro-batching stream processing framework. It batches incoming stream data of each timestep into in-memory data blocks, and creates an RDD per timestep. Such series of RDDs are named as a DStream. Further transformations and actions on those RDDs are handled by the Spark Core component.

### 6.1.2 Architecture

Although, by orchestrating various components of Spark, a large set of applications have significantly boosted their efficiency [122], applications that operate on dynamic dataset collections can still be improved in the following three aspects. First, in order to maintain the co-locality property, applications need to deliberately organize the in-memory data layout across the cluster. However, Spark randomly scatter partitions of independent RDDs into servers, exposing no control over partition placements. Second, as applications dynamically load and evict datasets, the data volume and computational demand received by the collection may vary over time. This requires those immutable RDDs in the collection to react to size and popularity dynamics, retaining load balancing. Third, dynamic collections of interdependent datasets may foster an ever-growing lineage graph. The system needs to minimize the overhead to achieve data persistency, and at the same time preserve failure recovery delay bounds. Therefore, we design Stark to handle these three problems accord-

ingly. Figure 6.1 illustrates the high level architecture of Stark. Besides enhancing multiple important building pieces of Spark, Stark introduces three novel components:

- **LocalityManager** enforces the same partitioner across multiple user-specified RDDs, and allocates corresponding partitions into same worker nodes in the best-effort manner. This helps to achieve data co-locality that could significantly benefit operations working on multiple RDDs (*e.g.*, cogroup and join).
- **GroupManager** introduces an extendable hashing policy to achieve partition elasticity, which allows immutable RDDs to shrink or expand partitions without repartitioning.
- **CheckpointOptimizer** employs a variant of network flow algorithm to optimally select the minimum amount of data to checkpoint, and at the same time fulfil a user defined failure recovery delay bound.

## 6.2 System Design

This section elaborates the design details of LocalityManager, GroupManager, and CheckpointOptimizer in 6.2.1, 6.2.2, and 6.2.3 respectively.

### 6.2.1 Locality Manager

Partition is the unit for computation and storage management in Spark. Co-partition and co-locality are two important features that closely relate to system performance. The former partitions multiple RDDs using the same partitioning strategy, which helps to avoid shuffling overhead. Let us use the notion *collection partition* to denote the corresponding partitions across co-partitioned RDDs. For example, collection partition 1 of two RDDs refers to the first partitions in both RDDs. The latter (co-locality) places an entire collection partition into the same executor, avoiding the cost of aggregating the data. Users may easily achieve co-partition by passing the same deterministic partitioner when constructing RDDs, while co-locality is more difficult to preserve. Spark randomly places partitions in the cluster, which means it is unlikely that data in the same collection partition would reside in the


**Figure 6.2:** Example of Violating Co-Locality

same server. Figure 6.2 demonstrates an example that violates data co-locality. The dataset collection contains two RDDs with each divided into three partitions. The number inside each partition represents its ID. In this example, a join or cogroup operation will create three tasks to process the three collection partitions respectively. Regardless of how Spark schedules those tasks, every task will encounter at least one remote RDD partition, as no collection partition locates in the same server.

To understand the importance of data co-locality, Section 6.2.1.1 explains default Spark behaviors and induced inefficiencies in greater details. Then, Section 6.2.1.2 presents the design of LocalityManager that allows users to preserve collection partition co-locality.

#### 6.2.1.1 Observations

Before investigating the benefits of data **co-**locality, let us first discuss how a Spark job could benefit from data locality. In Spark, the lineage graph of an RDD refers to all of its ancestors and corresponding transformations. Spark relies on this lineage graph to create stages when a user runs an action on the RDD. More specifically, a transformation with wide dependency generates a ShuffledRDD which contains a map phase and a reduce phase. Spark breaks the lineage graph at the barriers between the map and the reduce phases, leaving each connected component as a stage. Each stage contains a set of tasks with the same computational behavior. The task can be either a ShuffleMapTask or a ResultTask, depending on whether the stage ends at the map phase or the final RDD. The ShuffleMapTask commits map output data into persistent storage, from where the reducers retrieve data to continue the computation. Therefore, as reducers read map outputs from multiple servers through network, the reducing phases of ShuffledRDDs gain little performance improvements from enforcing data locality. However, data locality significantly reduces the execution time of transformations with narrow dependencies. For example, the code below generates two

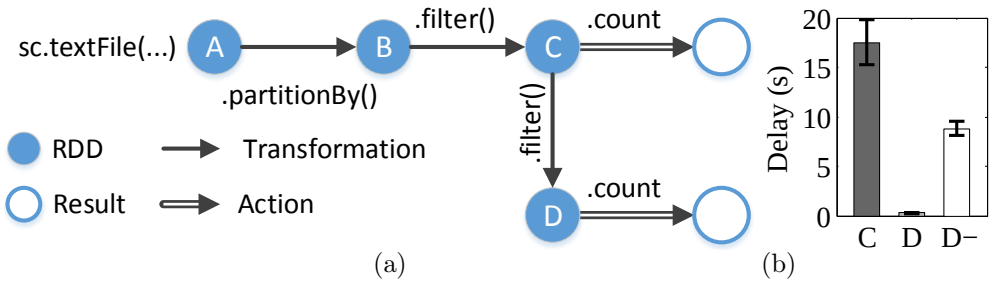



**Figure 6.3:** Data Locality Benefits

jobs, `C.count` and `D.count`. The lineage graph is shown in Figure 6.3 (a).

```
val A = sc.textFile(...).map(_ => (getTime(_), _))
val B = A.partitionBy(new HashPartitioner(2))
val C = B.filter(_ => _.startsWith("ERROR"))
val D = C.filter(_ => _.length > 30)
C.cache.count;    D.count
```

After Spark executes `C.count`, RDD `C` is cached in memory. RDD `D` relies on `C` as its parent. Figure 6.3 (b) shows how much execution time could be saved by preserving data locality when `sc.textFile` loads a $700MB$ text file. The letters `C` and `D` represent the execution time of `C.count` and `D.count` as the code shows. The job `C.count` creates two sequential stages. The first stage loads the text file, and commits all mapper outputs of RDD `B` into disk. The second stage starts from the reducers of RDD `B`, creates RDD `C`, and computes the count. When `C` is cached, `D.count` starts from the cached data, and the response time stays below 200ms. The `D-` represents the execution time of `D.count` after removing `.cache` from the last line of the code. The job `D-.count` creates a single stage that skips the `partitionBy` transformation, which helps to save 8s execution time compared to `C.count`. But, without data locality, `D-` has to start from the reducing phase of `B`. Consequently, the execution time increases from 0.2s to 9s.

Data locality reduces the job execution time by allowing stages to start from cached RDDs. When Spark fails to preserve data locality, instead of fetching from the executors where the RDD partitions are cached, it recomputes all transformations from the very beginning of the stage—reading data from the map outputs of ShuffledRDDs through network.

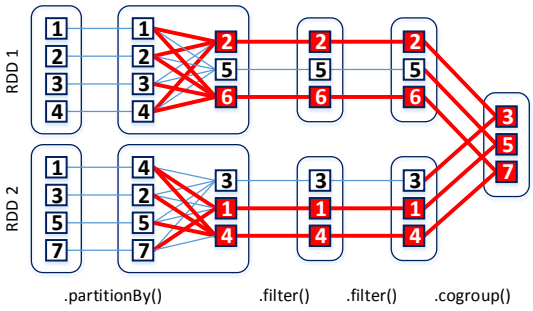



**Figure 6.4:** CoGroup Two RDDs in Spark

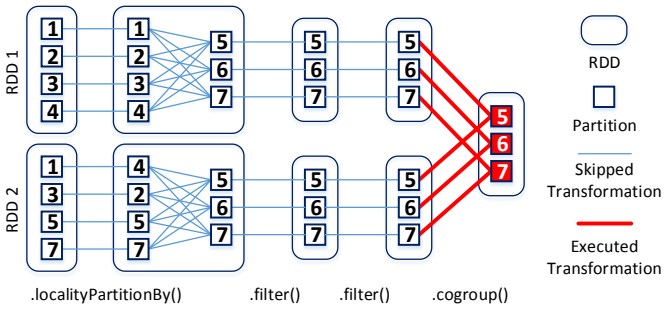



**Figure 6.5:** CoGroup Two RDDs in Stark

Although this design decision considerably reduces the complexity and overhead of keeping track of all cached and evicted data across the entire cluster, it on the other hand amplifies the penalty of scheduling a task onto a remote node. In the example shown in Figure 6.3 (a), forgoing data locality for action `D.count` forces Spark to start computing the stage from the reducing phase of RDD `B`, recompute RDD `C`, and finally create RDD `D`.

The example above demonstrates benefits of running Spark tasks with parent RDD partitions available in local cache, and penalties otherwise. Working on a collection of dataset exacerbates both the benefits and the penalties, as instead of a single RDD, each job deals with a collection that may contain a large number of RDDs. If a collection partition is scattered in different places, no single executor could preserve data locality for the task. Therefore, a job may trigger a huge amount of network and computation overhead. Figure 6.4 shows an example.

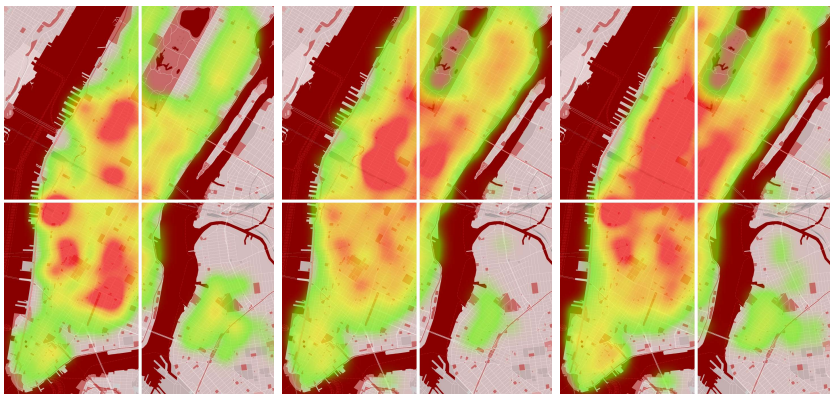
Compared to Figure 6.3, Figure 6.4 presents a more detailed view by emphasizing partitions of RDDs. Rectangles with round corners represent RDDs, and squares represent partitions in each RDD. The number in each partition shows the ID of the executor where the partition is cached. For simplicity, the figure only shows cogrouping two RDDs, whereas the number of involved RDDs in a real application might be much larger. On the right side lies the final cogrouped RDD. The red bold lines indicate which transformations on which partitions need to be re-computed. As can be seen in Figure 6.4, when co-locality is violated, a single job may trigger many partitions to be recomputed, even if those partitions have already been computed and cached somewhere else.

#### 6.2.1.2 Preserving Data Co-locality

LocalityManager helps to allocate RDD partitions in the same collection partition onto the same executor. Later in Section 6.2.2, we discuss techniques for solving potential side-effects of a collection partition growing too large to fit into a single executor. The LocalityManager internally remembers the mapping from collection partitions to executors. Users may create such mappings by calling the `localityPartitionBy` API on an RDD or a DStream. During task scheduling, the DAGScheduler first consults the LocalityManager to get the preferred executor ID, and then follows the default delay scheduling algorithm. A collection partition maps to a set of executors instead of a single one. Because whenever a task for a collection partition runs on a remote executor, the partition data is computed and cached in that executor, immediately making the partition data locally available for subsequent tasks on that same executor. This may happen when some collection partition takes too long to process or becomes too popular, overloading local executors. Therefore, the default delay scheduling policy tends to create more replications for those hotspot collection partitions.

Preserving co-locality significantly reduces job makespan. Based on the same example demonstrated in Figure 6.4, Figure 6.5 shows how partitions are allocated onto executors when LocalityManager is enabled. The three collection partitions consistently map to executors 5, 6, and 7 respectively, preventing jobs from having to read data from shuffled map outputs, and at the same time saving computational overhead of two transformations.

Enforcing collection partition co-locality also alters the behavior of memory allocation and task scheduling, both fundamentally affecting the system performance. Stark needs to make sure that all negative consequences are taken care of. As one obvious side-effect, a collection partition may grow too large or become too popular that overwhelms memory and/or computation resources of its corresponding executor set. Section 6.2.2 discusses this problem in detail and presents solutions. Besides that, another important core function is failure recovery. Spark recovers by recomputing unavailable RDD partitions from checkpoints and/or ShuffledRDDs. It speeds up this process by employing multiple executors to recover



(a) (b) (c)

**Figure 6.6:** Time-varying distribution of NYC taxi pick-up/drop-off events in 2013

RDDs in parallel. At the first glance, collection partition co-locality would slow down the recovery process, as a collection partition could be destroyed by even a single executor failure. Although this observation is true, we claim that the failure recovery with collection partition co-locality is at least as efficient as default Spark from the perspective of job makespan. Because the default Spark scheduler creates one task per result cogrouped RDD partition, which has to construct the entire collection partition in a single executor. Hence, even if a collection partition could be recovered using multiple executors, it has to aggregate into the same server before performing any further operations. Therefore, co-locality introduces no penalty for failure recovery.

### 6.2.2 Group Manager

In this section, we first consider trade-offs between different partitioning schemes in Section 6.2.2.1. Then, Section 6.2.2.2 presents the idea of extendable partition group that helps to achieve elasticity. Finally, Section 6.2.2.3 describes how Stark schedules partition groups when it fails to preserve data locality.

#### 6.2.2.1 Partitioning Trade-Offs

A dataset collection may subject to time-varying volume and distribution as datasets are dynamically inserted and evicted from the collection. For example, Figure 6.6 illustrates taxi

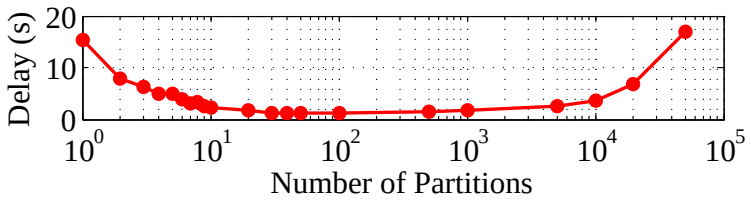



**Figure 6.7:** Partition Number Trade-Off

pick-up/drop-off events [75, 76] heatmap of three different 4-hour time slots in the Manhattan area. Suppose an application creates a dataset every ten minutes, and uses the dataset collection of the past hour to calculate taxi trajectories and optimize the advertisements displayed in each taxi's monitor. The three figures correspond to July $1^{st}$ morning, July $1^{st}$ evening, and July $4^{th}$ evening of 2013 respectively. The white grid helps to emphasize the locations of event hotpots. They clearly show that data distribution drastically changes over time, and there are much larger hotspot areas in (c) compared to (a) and (b). Therefore, no static partitioning algorithm could always preserve collection partition size under a reasonable threshold over time.

A straightforward solution to this problem would be to further divide data into finer granularity, which reduces the absolute size of partitions. However, this solution creates too many partitions that would overload the scheduler, and amplify system scheduling and monitoring overheads. Figure 6.7 depicts how the number of partitions affects the execution time of a Spark job. The experiment runs the same code as shown in Figure 6.3, and records the execution time of `C.count`. We manipulate the number of partitions by tuning the argument of `HashPartitioner`. The result shows that using more partitions initially does help reduce job execution time. However, as the number of partitions increases, the overheads gradually dwarf, and eventually completely overshadow the benefit of using higher parallelism.

To solve this dilemma, Stark handles the time-varying data and computation distributions by employing two mechanisms:

- **Extendable Partition Group** mitigates time-varying data distribution in a collection by splitting excessively large partitions to make use of memory and computation resources

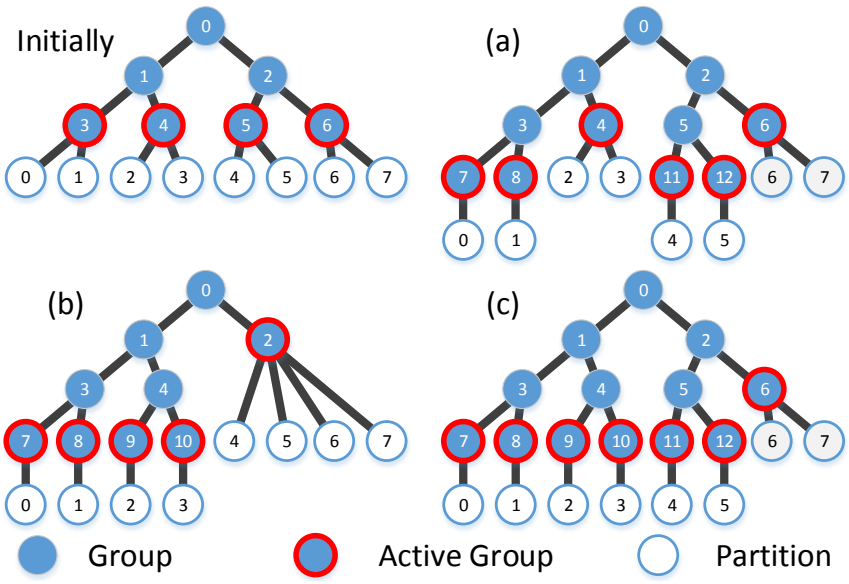



**Figure 6.8:** Extendable Partition Groups for NYC Taxi Data

on multiple machines, and merging tiny partitions to reduce the scheduling and control overhead.

- **Contention Aware Replication** scheme materializes multiple copies of collection partitions based on their demand, minimizing potential cache eviction penalties caused by excessive replications.

#### 6.2.2.2 Achieving Partition Elasticity

In Spark, partitioners share the same `getPartition` API to map a data key onto a partition ID. Achieving elasticity using the `getPartition` API would alter the key-to-partition mapping, resulting in exorbitant data shuffling cost when resizing partitions. To avoid this shuffling overhead, Stark attains elasticity from a higher level, which respects the mapping by keeping the `getPartition` API intact. Extendable partitioning enhances existing partitioners by introducing the concept of *partition group*. A partition group is a set of consecutive partitions. The extendable partitioner can be initially configured to contain $g$ groups. The partitioner will actually create $e \cdot g$ partitions, $e$ partitions per group. The $i^{th}$ group contains partitions $e \cdot i$ to $e \cdot (i+1) - 1$. To simplify the presentation, we require the configurable parameters $g$ and $e$ both to be powers of 2. This requirement can be easily relaxed by using

the smallest complete binary tree that contains exactly $g$ leaves.

Stark initially constructs a full binary tree where each leaf node in the tree corresponds to a partition group. We call this binary tree the *Group Tree*. Figure 6.8 elucidates example group trees constructed based on the data distribution as shown in Figure 6.6, assuming coordinates map to ordered-partitioned one-dimensional keys using Z encoding algorithm [123]. The four active groups in the initial status correspond to the four geographic regions as highlighted by the white grid. In Figure 6.6 (a), workloads overload the left two regions, which correspond to group 3 and 5 according to Z encoding algorithm. Hence, the Group Tree splits group 3 and 5 as shown in Figure 6.8 (a). Examples (b) and (c) work in the same fashion. A Group Tree supports two basic operations, split and merge. Split can be applied to any leaf node with more than one partition, dividing those partitions into two smaller groups. The merge operation can only be applied to two leaf node groups under the same parent node, concatenating partitions from the two groups into a larger group.

The split and merge operations are triggered by the total size of partitions in each group. Multiple RDDs may share the same Group Tree by using extendable partitioner under the same namespace, collectively affecting group sizes. The user may configure how many of the most recent RDDs are accounted when calculating the group size, as well as the upper and lower bounds of group sizes that trigger the split and merge operations.

A partition group is the minimum task scheduling unit in Stark. We introduce GroupResultTask and GroupShuffleMapTask as enhancements for ResultTask and ShuffleMapTask to allow multiple partitions in the same group to be packed into the same task. As we have discussed in Section 6.2.2.1, this feature helps to reduce scheduling and monitoring overhead by using a smaller number of tasks. Moreover, splitting (merging) a partition group also splits (merges) the corresponding local executors. This helps to minimize data movement during group dynamics by skipping cached partitions.

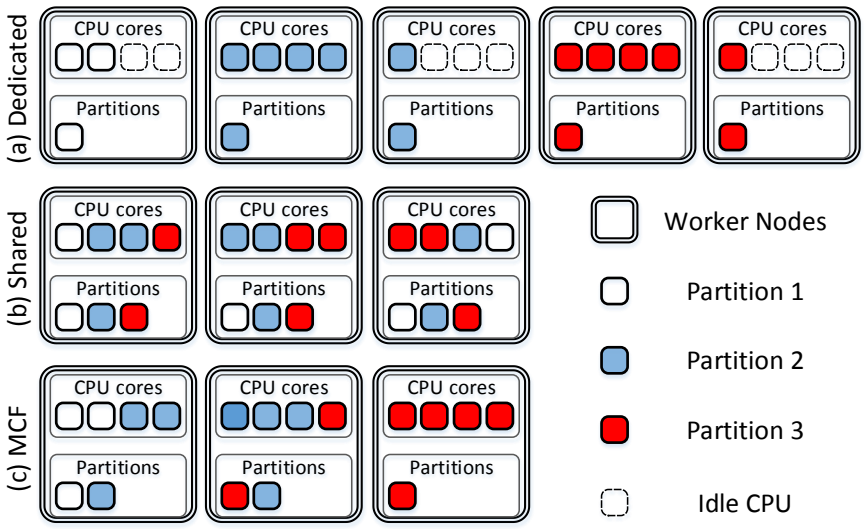



**Figure 6.9:** Task Scheduling Algorithms

#### 6.2.2.3 Contention Aware Replication

Extendable Partition Group solves the load balancing problem caused by data volume and distribution dynamics. As discussed in Section 6.2.2.1, there is another dimension of load balancing problem induced by time-varying and non-uniformly distributed computational demand on different partitions. To illustrate the problem, let us consider the same taxi advertising example mentioned in Section 6.2.2.1. Besides the spatial-temporal dynamics of taxi trajectories, the intensity of advertisement campaigns in certain areas may also change overtime. For example, theaters and shopping centers near the Time Square may want to deliver much more commercial messages to potential customers nearby in weekend evening compared to weekday morning. To address their demand, the system needs to first filter qualified trajectories using the location information, and then conduct subsequent optimization algorithms to match sponsor messages to taxi monitors. In this case, partitions that cover the Time Square receive higher computational workloads than others in weekend evening. Therefore, the amount of workload hitting the same partition changes over time, while such workload dynamics of different partitions may not follow the same pattern. To attack this load balancing problem, Stark delivers computational resource elasticity to the partition level by independently replicating each partition on demand while preserving data co-locality in

the best effort manner.

To achieve the computational resource elasticity, Stark needs to capture the right signal to trigger replicate and de-replicate operations. Failing to preserve data locality for a certain task on partition $\alpha$ conveys a relevant signal. This signal indicates either partition $\alpha$ has become a hotspot receiving higher computational demand, or the corresponding worker nodes where partition $\alpha$ resides has been assigned to too many different partitions. The latter situation forces every partition to replicate on an unnecessarily larger number of worker nodes, competing for limited computational resources. Although this could help to improve the CPU utilization, sharing worker nodes among many partitions catalyzes cache eviction, and makes data locality more difficult to achieve.

The delay scheduling policy improves data locality by allowing tasks to delay for a bounded amount of time to wait for local executors. When they fail to achieve data locality, all remote workers are treated equally. This design is reasonable and helpful when it was initially proposed for Hadoop MapReduce workloads. However, when imported into the realm of in-memory computation, scheduling a task on a remote node materializes all its narrowly-depended parents on that node, converting the node from `REMOTE` to `NODE_LOCAL` for subsequent tasks requiring the same input data. At the same time, this node's locality level may also revert back from `NODE_LOCAL` to `REMOTE` for some other partitions due to cache evictions. Therefore, schedulers for in-memory computation need to make more careful decisions when assigning tasks into remote worker nodes.

Figure 6.9 (a) and (b) illustrate two extremes: each worker node is dedicated to a single collection partition, or partitions can be assigned to any worker nodes in the cluster. In case (a), every partition exclusively consumes all RAM resources on its worker nodes, allowing it to fit more RDDs of the collection into the cache. Consequently, less online queries have to load data from disk. However, as we can clearly see in the example, many CPU cores stay idle under this scheme, as a price paid to guarantee the exclusiveness. In case (b), the scheduler blindly assigns tasks to remote resources, where any task may end up executing on any remote worker node. This scheduling policy optimally utilizes computational resources

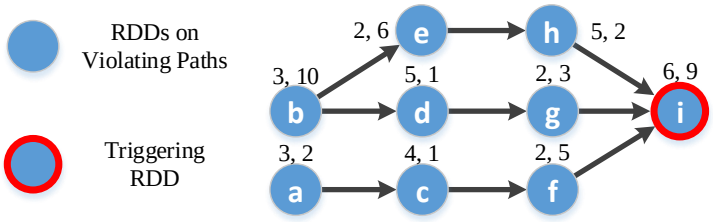



**Figure 6.10:** Checkpoint Example

in the cluster at the cost of restricting every collection to use a small share of RAM on worker nodes, which may even cause cascading cache eviction that results in higher cache miss rate.

**Algorithm 6.1** Minimum Contention First Scheduling

**Input:** task set of a stage **T**, offers **R**
**Output:** tasks to launch **L**

```
1: if max locality level is not REMOTE then
2:     use default delay scheduling
3: else
4:     L ← ∅
5:
6:     sort R according to the ascending order of unique collection partitions cached.
7:
8:     for i ← 1 ∼ |R| do
9:         t ← pick one task from T
10:
11:        T ← T \ {t}; L ← L ∩ (t, R[i])
12:
13:    end for
14:    return L
15: end if
```

We propose the *Minimum Contention First* (MCF) Delay Scheduling algorithm. MCF algorithm works exactly the same as the default delay scheduler before the locality level rises to `REMOTE`. When MCF algorithm is enabled, each executor keeps track of unique collection partitions cached in RAM. The scheduler gives higher priority to executors with a smaller number of unique collection partitions. Algorithm 6.1 shows the pseudo code.

### 6.2.3 Checkpoint Optimizer

Data co-locality and partition elasticity both help to avoid expensive data shuffling operations by preserving the same partitioning strategies across RDDs. Shuffling commits mapper outputs into persistent storage, which naturally breaks dependency chains to accelerate failure recovery. Hence, avoiding Shuffle transformations may lengthen failure recovery delay.

Therefore, the system needs to proactively and judiciously checkpoint RDDs to bound the failure recovery delay. In a simple case where RDDs are all independent from each other, the system has no choice but checkpointing all RDDs. However, many real-world applications form iterative structures, such as the *runningReduce* concept (*i.e.*, the `updateStateByKey` API) introduced in Spark Streaming [16]. In this case, the system has to select RDDs to checkpoint in the ever-growing application data. Otherwise, the failure recovery has to reconstruct RDDs from the very beginning. The existing solution [110] checkpoints the entire level of recently generated RDDs in the DAG, which is called the Edge algorithm. It guarantees bounded recovery delay, but may lead to excessive checkpointing overhead. This section optimizes the checkpointing algorithm to minimize the amount of data written into persistent storage, and at the same time bounds failure recovery delay.

#### 6.2.3.1 Measuring Parameters

From the perspective of failure recovery, each RDD associates with two important properties, the data size and the computation time. The system can estimate the data size by looking at the amount of RAM consumed by each cached RDD. Default Spark logs computation time at stage level, where a stage may contain multiple consecutive narrowly-dependent transformations, and thus span multiple RDDs. Hence, Stark needs to acquire the computation time from the transformation level. The delay of the same transformation can be different across tasks due to imperfectly balanced data distribution. Stark logs the delay of every transformation in every task, and takes the maximum across tasks as the estimation of the transformation delay. Data size represents the cost ($c$) of checkpointing an RDD, while computation time denotes the delay ($d$) of recovering the RDD. With these two notations, we design the optimal checkpointing algorithm for failure recovery.

#### 6.2.3.2 Optimized Checkpointing Algorithm

In the RDD DAG, each node associates with a cost $c$ and a delay $d$. Let an uncheckpointed path denote the path that contains no checkpointed RDD or ShuffledRDD. The path length

is the sum of all RDDs' delay $d$ along the path. Stark keeps track of all uncheckpointed RDDs, and triggers the checkpoint algorithm whenever the length of any path grows beyond the user defined failure recovery delay upper bound $r$. We call those paths violating paths.

The algorithm breaks all violating paths by checkpointing a set of RDDs with minimum total cost. Figure 6.10 illustrates an example with $r = 10$. The two numbers near each RDD represent the delay $d$ and the cost $c$ from left to right. Node $i$ triggers the checkpointing algorithm due to three violating paths of length 16, 16, and 15 respectively. As the goal is to find a minimum cut to isolate node $i$ from RDDs $a$ and $b$, we apply a variant of maximum network flow algorithm to solve this problem. The algorithm splits each RDD node into an *in* node and an *out* node. It then connects the out node of $i$ to a virtual sink node $t$, and connects a virtual source node $s$ to the in nodes of $a$ and $b$. The dependency relationships are preserved by linking predecessor's out node to successor's in node. The algorithm assigns the cost $c$ to edge between the *in* node and the *out* node of the same RDD as its edge capacity. The costs of all other edges are set to infinity. Finally, stark uses a standard maximum network flow algorithm to find the minimum cut, representing the set of RDDs with minimum total cost to checkpoint.

Stark determines the set of RDDs to checkpoint by tracing back from virtual sink $t$ to the first set of saturated cutting edges. However, searching for an exact cut may force the system to checkpoint RDDs that are too far away from latest RDDs, leaving a relatively long uncheckpointed path, which would inevitably trigger another checkpoint action soon. Hence, we relax this condition to allow Stark to stop at edges whose residual capacity is within $f$ times of the amount of network flow over it. This is the same as relaxing the checkpointing cost by $f$ times compared to the optimal solution.

## 6.3 Implementation

Stark is implemented based on Spark-1.3.1 by adding 2.9K lines of scala code. This section describes the components added or revised by Stark.

### 6.3.1 Locality Manager

We implement co-locality by introducing the LocalityManager, the `localityPartitionBy` transformation, and the `LocalityShuffledRDD`. They can be enabled by setting the `spark.scheduler.localityEnabled` property to true in the configuration file. Users may call the API on `PairedRDDFunctions` (*i.e.*, RDDs that store key-value pairs) to construct a `LocalityShuffledRDD` as below:

```
localityPartitionBy(p: Partitioner, ns: String)
```

LocalityManager creates a namespace if it has not seen `ns` before, or checks whether the partitioner `p` agrees with the existing partitioner registered with namespace `ns`. All RDDs under the same namespace must use the same partitioner. A namespace starts from a `LocalityShuffledRDD` and automatically carries on to all following narrowly-dependent RDDs. During scheduling, the DAGScheduler consults the LocalityManager for the preferred locations if there is a namespace associated with the RDD. In this way, the system preserves the co-locality in the best effort manner.

### 6.3.2 Group Manager

The GroupManager manages a bidirectional mapping between partitions and groups. Users may get access to the GroupManager from `SparkEnv`, and report an RDD by calling the `reportRDD(rdd:RDD)` API. This API will trigger GroupManager to calculate the collection partition size across all currently cached RDDs. Users can define lower and upper bounds on the collection partition size (*i.e.*, `spark.locality.max(min)GroupMemSize`). When a collection partition grows beyond those thresholds, the GroupManager splits or merges groups accordingly.

To reduce the scheduling overhead, we also introduce `GroupResultTask` and `GroupShuffleMapTask` that may operate on multiple partitions as a single task. Stark automatically creates group tasks if the target RDD associates with a namespace.

### 6.3.3 Checkpoint Optimizer

In Spark, users have to specify whether an RDD needs to be checkpointed before materializing it. However, this design prevents CheckpointOptimizer from applying the optimization algorithm. To break this constraint, we implement the `RDD.forceCheckpoint` API that creates an `RDDCheckpointData` object and calls the `doCheckpoint` method. This revision allows Stark to checkpoint an RDD after it has been materialized. The CheckpointOptimizer implements a variant of Edmonds-Karp [124] algorithm to find the optimal RDDs to checkpoint.

## 6.4 Evaluation

This section first describes datasets and clusters used in the experiments. Then, we evaluate improvements individually contributed by co-locality, extendability, and failure recovery in Sections 6.4.2, 6.4.3, and 6.4.4 respectively. Finally, Section 6.4.5 measures the overall system throughput and response time.

### 6.4.1 Experiments Setup

Our experiments run on a 50-server cluster [94] that consists of 40 Dell PowerEdge R620 servers and 10 Dell PowerEdge R610 servers. The Spark/Stark cluster runs on 40 R620 servers, each equipped with 16GB RAM. The remaining 10 R610 servers generate workloads and log delay. Evaluations employ a Wikipedia trace [125], an NYC taxi trace [75, 76], and a Twitter dataset crawled using Twitter REST APIs [126]. Due to the space limit, this chapter skips the statistics on these datasets. Dataset use-cases will be explained close to the corresponding experiments. Please refer to [127], [123], and [115] for detailed analyses on these three dataset respectively.

In the experiments below, we compare six different configurations in total. The prefix indicates whether the configuration uses Spark or Stark.

- *Spark-R*: creates a new RangePartitioner per RDD.

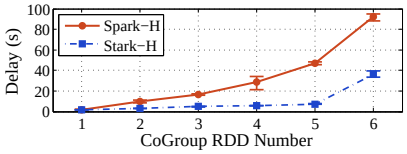



**Figure 6.11:** Co-locality Job Delay

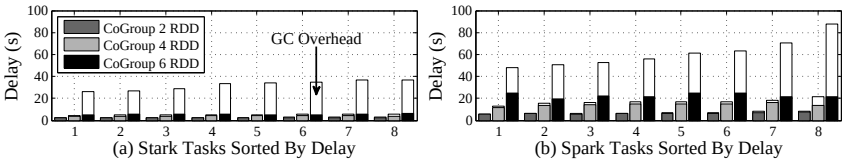



**Figure 6.12:** Co-locality Task Delay


**Figure 6.13:** Task Input Data Size

- *Spark-H*: all RDDs share the same HashPartitioner.
- *Stark-H*: applies the same HashPartitioner across RDDs with only co-locality enabled.
- *Stark-S*: applies the same StaticRangePartitioner across RDDs with only co-locality enabled.
- *Stark-E*: Spark-S plus Extendability enabled.

### 6.4.2 Data Co-Locality

In this section, we use Wikipedia trace data [125] and typical log mining jobs to measure the performance gain achieved by enabling data co-locality. We pick the log files that contain about 800MB data each, create an RDD for every file, and use the default hash partitioner with *eight* partitions (*i.e.*, this experiment only uses eight servers). Then, each job cogroups a range of trace RDDs, and counts the number of trace items that contain a randomly picked keyword. Figure 6.11 shows the results of the average delay of 10 queries. The $x$-axis is the number of RDDs that each job cogroups, and the $y$-axis the delay. The delay gap between *Spark-H* and *Stark-H* grows as the number of RDDs increases from 1 to 5, due to the fact that cogrouping more RDDs in *Spark-H* requires moving more data into the same executor through network. When cogrouping 5 RDDs, Stark reduces the delay from 46s to 9s by enforcing data co-locality.

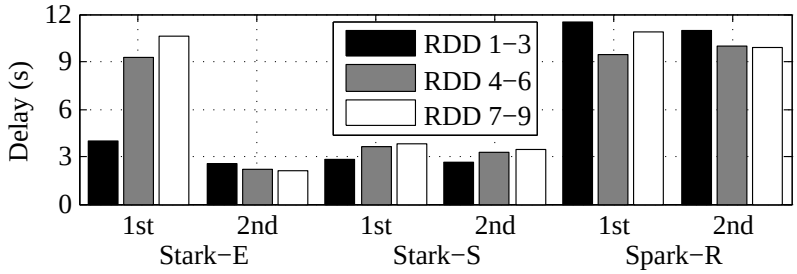



**Figure 6.14:** Job Delay under Skewed Distribution

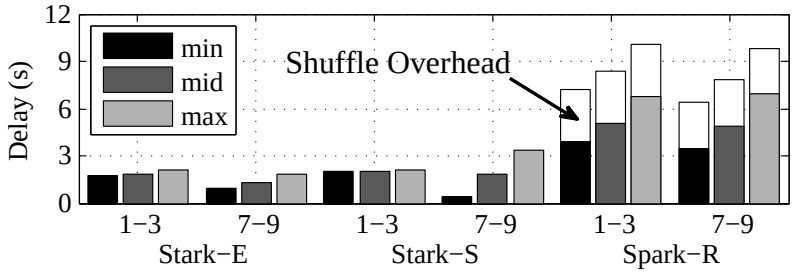



**Figure 6.15:** Task Delay under Skewed Distribution

The interesting behavior is that, when cogrouping 6 RDDs, the improvement ratio decreases to 3X. To understand this, let us take a closer look at the delay at the task level for specific jobs. Figure 6.12 (a) and (b) illustrate task delays when using *Stark-H* and *Spark-H* respectively. In each bar, the white portion denotes the garbage collection (GC) delay, and the remaining denotes the sum of all other delays. We can clearly see that performance gain drops due to the garbage collection overhead which is triggered as the memory consumption of cogrouping 6 RDDs together pushes towards executors' RAM limits.

### 6.4.3 Extendable Partitioning

In this section, we measure how Stark adapts to data distribution dynamics using consecutive Wikipedia hourly log files. As analyzed in [127], a peak hour log file may contain as much as twice the amount of data compared to that of nadir hours. In the experiments, we compare *Spark-R*, *Stark-S*, and *Stark-E*. Figure 6.13 shows how these three strategies partition data into groups or partitions, which immediately determines the input data size of different tasks. Each row represents a collection of three RDDs with RDD id marked on the $y$-axis. Each cell

corresponds to a collection partition (or groups). Darker colors refer to larger partition sizes. In the ideal case, all tasks should have an equal amount of input data, such that no overloaded task would delay the entire job. *Stark-S* clearly suffers from skewed data distribution as some collection partitions are much darker than others. *Spark-R* evenly distributes data across collection partitions, as different RDDs use different range partitioners specifically tailored for their own data distributions. In the *Stark-E* case, partition group boundaries (*i.e.*, sizes of cells in Figure 6.13) change across RDDs to cater skewed data distribution, resulting in a relatively balanced partition group size distribution.

Figure 6.14 shows job delays under three configurations. We distinguish the delay of the first job (1st) after group merges/splits from following jobs (2nd). Although *Spark-R* distributes data evenly, jobs take more than 10 seconds to process, regardless of if they are the first or subsequent jobs. This is because RangePartitioners use different boundaries to balance data across partitions, such that a cogroup operation inevitably triggers expensive data shuffling operations. *Stark-S* finishes jobs within 4s. As the partitioning strategy is static and data locality stays unchanged across RDDs, the performance of *Stark-S* ignores whether they are the first or subsequent jobs. However, it takes considerably longer delay to process data with skewed distributions (RDDs 4-6 and 7-9) compared to those with uniform data distributions (RDDs 1-3). *Spark-E* experiences large delay on the first job on RDD 4-6 and 7-9. As it alters the partition-to-executor mapping to regain load balancing, the first job has to reconstructs partition data in newly assigned executors, that contributes to the longer delay. These experiments convey that, if the data is only used once, then there is no need to enable extendable groups, as co-locality alone already achieves the shortest delay. However, many interactive and iterative applications [128] require to run a series of jobs on the same set of data, which would considerably benefit from extendable groups.

Figure 6.15 illustrates task level delay when cogrouping RDDs 4-6. In each bar, the white portion refers to the shuffling delay. It confirms that the shuffling overhead contributes a huge part to *Spark-R* job delay. We can also see that, the skewed data distribution causes imbalanced task completion times when using *Stark-S*, leading to a longer job delay.

### 6.4.4 Failure Recovery

Failure recovery is evaluated by using an example application as shown in Figure 6.16. The application tracks popular keys and corresponding contents in the similar way as Twitter trends. Blue nodes without border represent RDDs generated and consumed by the current step. Black RDDs are generated by last step and consumed by the current step, while RDDs with bold red rim are generated by the current step and consumed by the next step. In this way, steps are chained together into an ever-growing lineage graph. The application receives a *raw* RDD that contains key-value data in every step. The *raw* RDD partitions to the *kv* RDD by applying the `partitionBy` (pttBy) transformation. Then, the application aggregates the count (cnt) and content (ctt) by calling `reduceByKey` (rbk) API respectively. The *cnt* RDD cogroups (cogrp) with the decayed count RDD (dec) from the last step, and sums by key to create the *ccnt* RDD. Then, it only keeps the popular keys (acnt) by applying the filter API. The *ctt* RDD cogroups with the result RDD *res* from last step to generate the *cctt* RDD. Finally, *cctt* joins with popular keys (*acnt*) to create popular key-value pairs (*jall*) across steps, and cleans the data to arrive the final result (*res*) of the current step. Please notice that, we are not arguing this is the optimal way to implement the application logic, as how application is implemented falls out of the scope of this chapter.

We run the application for 10 steps with varying input data sizes. We feed the Wikipedia trace as the input data, and use a fixed-length prefix of the requested URL as the key. We first evaluate how accurate we could estimate checkpoint sizes using cached RDD sizes. As shown in Figure 6.17, where the white bars represent the cached RDD size and the gray bar the checkpoint size, there is a constant relationship between the cache and checkpoint sizes. Although this constant may change when we use different serialization algorithms, it does not affect the checkpoint algorithm, as the algorithm finds the relatively optimal set of RDDs to persist.

Then, we measure the amount of checkpointed data over steps. Figure 6.18 compares three schemes: 1) Stark enforces exact optimality (*Stark-1*); 2) Stark relaxes the optimality

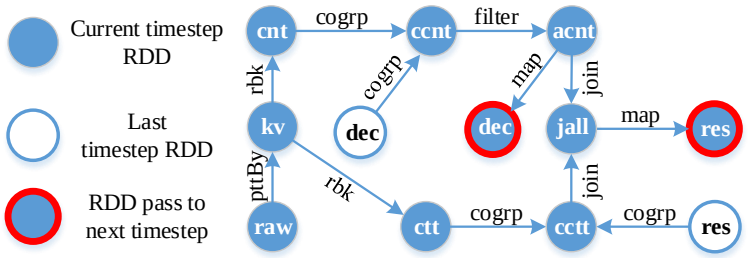



**Figure 6.16:** Application Lineage Graph

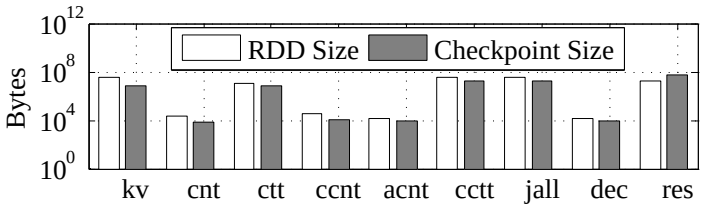



**Figure 6.17:** Estimate Checkpoint Size

by using $f = 3$ (*Stark-3*); 3) A variant of Tachyon [110] Edge checkpointing algorithm that checkpoints all leaf RDDs when invoked. Edge algorithm assumes there is a backend thread checkpointing data whenever the thread is idle, which differs from our assumptions that checkpointing algorithm is proactively triggered to enforce recovery delay bound. So, we revise the Edge algorithm to allow checkpoint operations to be conducted whenever the length of any uncheckpointed path violates the delay threshold. From Figure 6.18, we can see that Stark writes much less data compared to Tachyon, which immediately translates to savings on cluster resource consumption. Because checkpointing all leaves does not guarantee optimality. For example, after calculating *cctt*, the system realizes its recovery chain is too long due to the dependencies among *cctt*, *res*, and *jall*. Hence, Tachyon would checkpoint the current leaf RDDs *dec* and *cctt*. However, after generating *jall*, its recovery chain violates the recovery bound again due to the *ccnt*, *acnt*, and *dec* dependencies. Tachyon would then checkpoint the leaf *jall*. However, as shown in Figure 6.17, the size of *jall* is much larger than the size of *acnt*. Therefore, Stark would choose to checkpoint *acnt* instead to reduce the overhead.

*Stark-1* beats *Stark-3* in the first 4 steps as it enforces exact optimality. However, when the lineage grows larger, *Stark-3* outperforms *Stark-1*, as *Stark-1* tends to leave longer

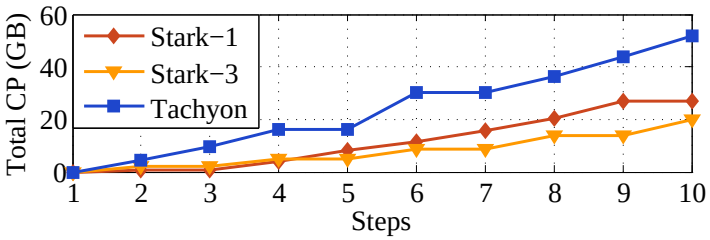



**Figure 6.18:** Total Checkpoint Size

uncheckpointed paths after each action, forcing checkpointing actions to be invoked more frequently.

### 6.4.5 Throughput and Delay

We measure the delay and the throughput using Twitter data. As only a small fraction of tweets are tagged with location information, we merge the NYC taxi trace and Twitter dataset together by appending a tweet after every taxi pick-up/drop-off event log, such that every tweet associates with a geographic coordinate and a new timestamp. We then replay the result dataset as a TCP stream source. Stark uses the streaming component to process incoming data, creating one RDD for every 5-minute data. The spatial coordinates are encoded into an one-dimensional space using the Z encoding algorithm [123]. Each job is based on the data within a random time range and a random geographic region, which triggers cogroup operations on multiple RDDs.

As the trace emits data at varying speed over time, the throughput may also change accordingly. In order to arrive at a steady number, we first measure the system throughput under a static speed by revising the trace timestamps and forcing the stream source to release an equal number of tweets per second. To calculate the throughput, we measure the number of jobs per second the system could handle when keeping the delay below 800ms. Figure 6.19 plots the result. The curve with purple triangles on the upper-left corner represents the *Spark-R* baseline. As it requires expensive data shuffling operation, jobs take 630ms to finish on average and handles only 9 queries per second. When using the same hash partitioner (*Spark-H*) across all RDDs, the response time drops to 405ms, and the throughput improves

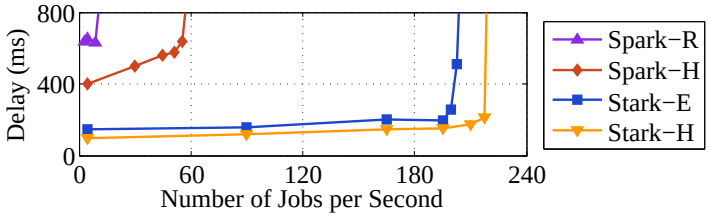



**Figure 6.19:** System Throughput

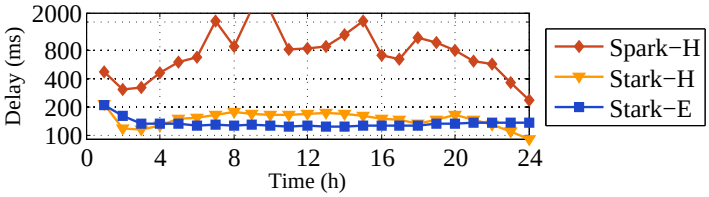



**Figure 6.20:** Job Delay Over Time

to 56 queries per second. After further enforcing data co-locality, response time immediately decreases to 109ms, and the system handles up to 220 queries per second. In this set of experiments, as both data generating speed and distribution stay unchanged, partition elasticity helps little. As shown by the *Spark-E* curve, the extra overhead introduced by grouping tasks slightly hurts the system response time and throughput.

In Figure 6.20, we measure the system response time by replaying the trace at the real speed according to the NYC taxi trace timestamp. The timestep size is set to 5 minutes. Queries analyze a random range of timesteps in the past three hours. Due to the unacceptably high response time and low throughput as shown in Figure 6.19, the experiment excludes the *Spark-R* baseline. To compare the response time, the amount of workload has to stay within the capability of all baselines. Hence, workload emulators generate 20 jobs per second. As shown by the red curve with diamonds, the response time of *Spark-H* surpasses 800ms when the amount of data generated per second increases. *Stark-H* keeps the response time below 200ms. *Stark-E* shows the benefits of extendable partitions. As the amount of data per timestep increases, each job spans to a larger set of executors with each executor processing a smaller amount of data. We can conclude from the results that, although *Stark-E* subjects to longer delay compared to *Spark-H* when the workload is static and light, it outperforms *Spark-H* by elastically scaling out when the amount of workload grows.

## 6.5 Related Work

In-memory computation [16, 18, 111, 113, 115, 129–135] has recently attracted immense attentions from both academia and industry. Multiple efforts have been expended on investigating various aspects of the problem and pushing forward this direction. For example, Spark [18], as a generalized in-memory computation system, has already greatly boosted data processing solutions for many application needs throughout the industry [122], gaining rapid growth in user population. Spark improves on Hadoop by storing the results of the intermediate stages in memory instead of disk, and thereby eliminating the bottlenecks caused by slow disk accesses. Its streaming components [16] sits on top of Spark core, and enables stream processing using micro-batching strategies. Naiad [113] unifies stream and batch processing systems, offer comparable functionalities as Spark. These systems offer limited optimizations for applications working on collections of datasets.

Spark has a rich ecosystem to handle specialized use cases. Spark SQL [111, 129] supports SQL-like queries on structured data in spark applications. It translates SQL queries into Spark jobs, and optimizes runtime code generation. MLlib [130], as another example, is a scalable machine learning library implemented on Spark. SocialTrove [115] applies Spark and in-memory caching to implement a content-agnostic summarization infrastructure for social data streams. GraphX [131] is a graph processing framework to provide vertex parallel abstractions using the concept of RDD. Several of the above applications also work on dataset collections. For example, the slice transformation in Spark Streaming allows jobs to operate on multiple timestep RDDs within a given time interval. Graphx relies on distributed joins to bring together edge and node data. Therefore, optimizations for dataset collection will benefit these applications as well.

# Chapter 7: Conclusion and Future Work

## 7.1 Summary

This thesis proposes a dynamic resource provisioning framework for data center workloads. The framework contains four components: system, sensor, policy and manager. The system is the target to enable dynamic resource provisioning, and the sensor collects system status. The policy makes provisioning decisions and the manager configures the system following the provisioning decision and takes care of potential transition penalties.

Due to the diversity in data center software systems, the policy and the manager components need to be customized for different types of workloads. This thesis focuses on workloads from storage and computing systems. For storage systems, such as the HBase/HDFS stack, dynamic resource provisioning may destroy the mapping from data items to physical servers, which in turn brings the penalty of losing data locality. I solve this problem by altering the data replica placement policy in the HDFS layer, allowing HBase daughter regions to land on a physical server with the region data already stored locally on the disk. For computing systems, such as Hadoop MapReduce and Spark, the thesis first proposes a theoretical solution to determine the minimum amount of resource to guarantee schedulability. Then, the design of namespace is introduced to enforce co-locality properties.

## 7.2 Future Research

Solutions discussed in this thesis are designed for data centers composed of conventional servers that individually possesses a huge amount of computation, storage, and network resources. In the future, we plan to explore a fundamentally different approach that hybrids conventional servers with ultra-low power embed devices.

To evaluate the feasibility, we built a cluster composed of 35 Intel Edison devices, individually running at low CPU frequency (dual-core 500MHz) while collectively yielding more energy-efficiency in various data center workloads. The Edison device is typically used for Internet-of-Things and sensor networks due to its ultra-low power consumption, but we also show that a larger cluster of such sensor-class micro servers still possesses significant computational power for data center workloads when managed properly, while achieving more work-done-per-joule. The intuitive reason that modern high-end processors are less efficient is that in order to reach higher speed, the energy consumption increases super-linearly, devoting much of the power to branch prediction, speculative execution and out-of-order execution. In contrast, small processors running at lower frequency are more energy efficient since their use much smaller die area for L2/3 cache compared to high-end processors, thus dedicating most energy to basic computations. In addition, a scale-out cluster of micro servers has the ability to shutdown nodes at a finer granularity, potentially increasing the energy proportionality and reducing the overhead of waking up servers. Finally, the low price and small physical size of micro servers lead to more cost-efficient and compact cluster deployment, while their ultra-low power consumption makes it easier to achieve higher server density without worrying about the cooling and power supply limits.

# Bibliography


[1] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *ACM SoCC*, 2013.

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

[3] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!" in *ACM SoCC*, 2014.

[4] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 77–88, 2013.

[5] G. Lee, B.-G. Chun, and R. H. Katz, "Heterogeneity-aware resource allocation and scheduling in the cloud," *Proceedings of HotCloud*, pp. 1–5, 2011.

[6] S. Li, S. Wang, F. Yang, S. Hu, F. Saremi, and T. Abdelzaher, "Proteus: Power proportional memory cache cluster in data centers," in *IEEE ICDCS*, 2013.

[7] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz, "Napsac: Design and implementation of a power-proportional web cluster," *ACM SIGCOMM computer communication review*, vol. 41, no. 1, pp. 102–108, 2011.

[8] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, "Robust and flexible power-proportional storage," in *ACM SoCC*, 2010.

[9] M. Lin, A. Wierman, L. L. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 5, pp. 1378–1391, 2013.

[10] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: a power-proportional, distributed storage system," *Microsoft Research, Cambridge, UK, Tech. Rep. MSR-TR-2009-153*, 2009.

[11] J. Heo, P. Jayachandran, I. Shin, D. Wang, T. Abdelzaher, and X. Liu, "Optituner: On performance composition and server farm energy minimization application," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 11, Nov. 2011.

[12] L. Wang and Y. Lu, "Efficient power management of heterogeneous soft real-time clusters," in *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS*, 2008, pp. 323–332.

[13] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam, "Managing server energy and operational costs in hosting centers," in *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2005.

[14] "Apache Hadoop," https://hadoop.apache.org/, September 2015.

[15] "Apache Hadoop V2 (YARN): Yet Another Resource Negotiator," http://hortonworks.com/hadoop/yarn/, October 2014.

[16] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM SOSP*, 2013.

[17] "Spark: Lightning-Fast Cluster Computing," http://spark.apache.org/, September 2015.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX NSDI*, 2012.

[19] "Apache Storm," https://storm.apache.org, September 2015.

[20] "Memcached," http://memcached.org/, September 2015.

[21] "Apache HBase," http://hbase.apache.org/, September 2015.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[23] S. Baruah, "federated scheduling of sporadic dag task systems," in *IEEE IPDPS*, 2015.

[24] J. C. Fonseca, V. Nélis, G. Raraviand, and L. M. Pinho, "A multi-dag model for real-time parallel applications with conditional execution," in *ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2015.

[25] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, "federated scheduling of sporadic dag task systems," in *ECRTS*, 2015.

[26] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *IEEE RTSS*, 2011.

[27] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *IEEE RTSS*, 2012.

[28] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global edf for parallel tasks," in *IEEE ECRTS*, 2013.

[29] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *ECRTS*, 2014.

[30] G. Liu, Y. Lu, S. Wang, and Z. Gu, "Partitioned multiprocessor scheduling of mixed-criticality parallel jobs," in *IEEE RTCSA*, 2014.

[31] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012.

[32] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *The Journal of Real-Time Systems*, vol. 1, pp. 27–60, 1989.

[33] B. Sprunt and L. Sha and J. Lehoczky, "Enhanced aperiodic responsiveness in hard real-time environment," in *IEEE RTSS*, 1987.

[34] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, 1995.

[35] S. Li, S. Hu, and T. Abdelzaher, "The packing server for real-time scheduling of mapreduce workflows," in *IEEE RTAS*, 2015.

[36] T. P. Baker, "A comparison of global and partitioned edf schedulability tests for multiprocessors," RTNS, Tech. Rep., 2005.

[37] J. M. López, M. García, J. L. Díaz, and D. F. García, "Worst-case utilization bound for edf scheduling on real-time multiprocessor systems," in *ECRTS*, 2000.

[38] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[39] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[40] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *IEEE RTSS*, 2005.

[41] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Trans. Comput.*, vol. 44, no. 12, pp. 1429–1442.

[42] S. K. Lee, "On-line multiprocessor scheduling algorithms for real-time tasks," in *IEEE Frontiers of Computer Technology*, 1994.

[43] G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[44] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm," in *Real-Time Systems Symposium, 1988., Proceedings.* IEEE, 1988, pp. 251–258.

[45] C. Maia, L. Nogueira, L. M. Pinho, and M. Bertogna., "Response-time analysis of fork/join tasks in multiprocessor systems," in *ECRTS*, 2013.

[46] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2014.

[47] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *IEEE RTSS*, 2010.

[48] J. Li, K. Agrawal, C. Gill, and C. Lu, "Federated scheduling for stochastic parallel real-time tasks," in *RTCSA*, 2014.

[49] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, "Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop cluster," in *IEEE ICDCS*, 2014.

[50] "Strategy: Break up the memcache dog pile," http://highscalability.com/strategy-break-memcache-dog-pile, August 2009.

[51] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: facebook's photo storage," ser. USENIX OSDI, 2010.

[52] L. Kai, A. Sridhar, S. Kannan, and S. Maguluri, "Indra: A data placement and replication system for online social networks," Technical Report, 2011.

[53] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[54] "Hadoop Distributed File System," http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2015.

[55] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[56] "And a fun weekend was had by all," http://blog.reddit.com/2010/03/and-fun-weekend-was-had-by-all.html, June 2012.

[57] C. Bash and G. Forman, "Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center," in *USENIX Annual Technical Conference*, 2007.

[58] J. Moore, J. Chase, P. Ranganathan, and R. Sharma, "Making scheduling 'cool': Temperature-aware workload placement in data centers," in *USENIX Annual Technical Conference*, 2005, pp. 61–74.

[59] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," ser. ACM SIGCOMM, 2001.

[60] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," ser. STOC, 1997.

[61] "Memcached internals," http://www.adayinthelifeof.nl/2011/02/06/memcache-internals/, November 2012.

[62] P. Saab, "Scaling memcached at facebook," http://www.facebook.com/note.php?note_id=39391378919, December 2008.

[63] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, pp. 281–293, Jun. 2000.

[64] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002, pp. 636–646.

[65] J. H. Lambert, *Observationes variae in mathesin puram.* Acta Helvetica, physico-mathematico-anatomico-botanico-medica, 1758.

[66] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, pp. 1830–1845, July 2009.

[67] B. Sanou, "The world in 2013: Ict facts and figures," in *International Communication Union, United Nations*, 2013.

[68] S. Steiniger and E. Bocher, "An overview on current free and open source desktop gis developments," *International Journal of Geographical Information Science*, vol. 23, no. 10, pp. 1345–1370.

[69] L. George, *HBase: The Definitive Guide.* O'Reilly, 2011.

[70] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40.

[71] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SOSP*, 2007.

[72] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *USENIX ATC*, 2013.

[73] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *IEEE International Conference on Mobile Data Management*, 2011.

[74] Y. Ma, Y. Zhang, and X. Meng, "St-hbase: A scalable data management system for massive geo-tagged objects," in *International Conference on Web-Age Information Management*, 2013.

[75] B. Donovan and D. B. Work, "Using coarse gps data to quantify city-scale transportation system resilience to extreme events," *Transportation Research Board 94th Annual Meeting*, 2014.

[76] New York City Taxi & Limousine Commission (NYCT&L), "Nyc taxi dataset 2010-2013," https://publish.illinois.edu/dbwork/open-data/, 2015.

[77] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *USENIX OSDI*, 2006.

[78] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: A facebook messages case study," in *USENIX FAST*, 2014.

[79] M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Springer Publishing Company, Incorporated, 2012.

[80] J. Lawder, "The application of space-flling curves to the storage and retrieval of multi-dimensional data," in *Ph.D. Thesis*, 2000.

[81] K.-L. Chung, Y.-L. Huang, and Y.-W. Liu, "Efficient algorithms for coding hilbert curve of arbitrary-sized image and application to window query," *Inf. Sci.*, vol. 177, no. 10, pp. 2130–2151.

[82] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Halevy, "Hyper-local, directions-based ranking of places," *Proc. VLDB Endow.*, vol. 4, no. 5, pp. 290–301.

[83] R. Dickau, "Hilbert and moore 3d fractal curves," http://demonstrations.wolfram.com/HilbertAndMoore3DFractalCurves/, 2015.

[84] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *USENIX ATC*, 2013.

[85] R. J. Chansler, "Data availability and durability with the hadoop distributed file system," in *The USENIX Magzine*, 2012.

[86] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *USENIX OSDI*, 2010.

[87] S. Li, S. Wang, T. Abdelzaher, M. Kihl, and A. Robertsson, "Temperature aware power allocation: An optimization framework and case studies," *Sustainable Computing: Informatics and Systems*, 2012.

[88] S. Li, T. F. Abdelzaher, and M. Yuan, "TAPA: temperature aware power allocation in data center with map-reduce," in *IEEE International Green Computing Conference and Workshops*, 2011.

[89] S. Li, H. Le, N. Pham, J. Heo, and T. Abdelzaher, "Joint optimization of computing and cooling energy: Analytic model and a machine room case study," in *IEEE ICDCS*, 2012.

[90] S. Li, S. Hu, S. Wang, L. Su, T. F. Abdelzaher, I. Gupta, and R. Pace, "WOHA: deadline-aware map-reduce workflow scheduling framework over hadoop clusters," in *IEEE ICDCS*, 2014.

[91] M. M. H. Khan, J. Heo, S. Li, and T. F. Abdelzaher, "Understanding vicious cycles in server clusters," in *IEEE ICDCS*, 2011.

[92] S. Li, S. Hu, S. Wang, S. Gu, C. Pan, and T. F. Abdelzaher, "Wattvalet: Heterogenous energy storage management in data centers for improved power capping," in *USENIX ICAC*, 2014.

[93] S. Li, L. Su, Y. Suleimenov, H. Liu, T. F. Abdelzaher, and G. Chen, "Centaur: Dynamic message dissemination over online social networks," in *IEEE ICCCN*, 2014.

[94] CyPhy Research Group, "UIUC Green Data Center," http://greendatacenters.web.engr.illinois.edu/index.html, 2015.

[95] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, 2010.

[96] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the hilbert space-filling curve," *SIGMOD Rec.*, vol. 30, no. 1, pp. 19–24.

[97] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *VLDB*, 2007.

[98] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *ACM SIGMOD*, 1984.

[99] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o(n log n)," in *IEEE Symposium on Interactive Ray Tracing*, 2006.

[100] R. Bayer, "The universal b-tree for multidimensional indexing: General concepts," in *Proceedings of the International Conference on Worldwide Computing and Its Applications*, 1997.

[101] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the ub-tree into a database system kernel," in *VLDB*, 2000.

[102] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *ACM SIGMOD*, 2010.

[103] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *ACM SIGCOMM*, 2001.

[104] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng, "An efficient multi-dimensional index for cloud data management," in *International Workshop on Cloud Data Management*, 2009.

[105] B. Cho and M. K. Aguilera, "Surviving congestion in geo-distributed storage systems," in *USENIX ATC*, 2012.

[106] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *USENIX NSDI*, 2014.

[107] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key-value store," in *USENIX OSDI*, 2010.

[108] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide Time to Relax*, 1st ed. O'Reilly Media, Inc., 2010.

[109] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *USENIX HotCloud*, 2010.

[110] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *ACM SoCC*, 2014.

[111] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *ACM SIGMOD*, 2013.

[112] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *ACM SIGMOD*, 2013.

[113] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *ACM SOSP*, 2013.

[114] "Apache Flink: an open source platform for distributed stream and batch data processing," https://flink.apache.org/, September 2015.

[115] M. T. Amin, S. Li, M. R. Rahman, P. Seetharamu, S. Wang, T. Abdelzaher, I. Gupta, M. Srivatsa, R. Ganti, R. Ahmed, and H. Le, "Socialtrove: A self-summarizing storage service for social sensing," in *USENIX ICAC*, 2015.

[116] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[117] M. Zaharia, "How spark usage is evolving in 2015," 2015.

[118] W. Chen, "Spark and Shark Bridges the Gap Between Business Intelligence and Machine Learning at Yahoo! Taiwan," in *Spark Summit*, 2014.

[119] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica, "Fast and interactive analytics over hadoop data with spark," *USENIX*, vol. 37, no. 4, pp. 45–51, 2012.

[120] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *USENIX ATC*, 2015.

[121] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010.

[122] "Databricks: Make Big Data Simple," https://databricks.com/customers, September 2015.

[123] S. Li, S. Hu, R. Ganti, M. Srivatsa, and T. Abdelzaher, "Pyro: A spatial-temporal big-data storage system," in *USENIX ATC*, 2015.

[124] N. Zadeh, "Theoretical efficiency of the edmonds-karp algorithm for computing maximal flows," *J. ACM*, vol. 19, no. 1, pp. 184–192, Jan. 1972.

[125] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[126] "Twitter Developers," https://dev.twitter.com/, September 2015.

[127] S. Li, S. Wang, F. Yang, S. Hu, F. Saremi, and T. F. Abdelzaher, "Proteus: Power proportional memory cache cluster in data centers," in *IEEE ICDCS*, 2013.

[128] "Changing the Way Businesses Comput and Compete: In-Memory Computing and Real-Time Business Intelligence."

[129] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *ACM SIGMOD*, 2015.

[130] "MLlib: Apache Spark's scalable machine learning library," http://spark.apache.org/mllib/, September 2015.

[131] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *ACM International Workshop on Graph Data Management Experiences and Systems*, 2013.

[132] A. Koliousis, M. Weidlich, R. C. Fernandez, P. Costa, A. L. Wolf, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *ACM SIGMOD*, 2016.

[133] J. S. Jeong, W.-Y. Lee, Y. Lee, Y. Yang, B. Cho, and B.-G. Chun, "Elastic memory: Bring elasticity back to in-memory big data analytics," in *USENIX HotOS*, 2015.

[134] L. Hu, K. Schwan, H. Amur, and X. Chen, "ELF: Efficient Lightweight Fast Stream Processing at Scale," in *USENIX ATC*, 2014.

[135] A. Bar, A. Finamore, P. Casas, L. Golab, and M. Mellia, "Large-scale network traffic monitoring with dbstream, a system for rolling big data analysis," in *IEEE BigData*, 2014.