Achieving Online Big Data Processing: From Offline to Online Data Analytics

by

Nan Zhu

School of Computer Science

McGill University

Montreal, Quebec, Canada

2016-05

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

Nowadays, a huge volume of data is increasingly produced by various sources, such as mobile sensing devices, system logs, and user activities over Internet. Accordingly, the information hidden in the massive size of data has a great potential to change and benefit our lives in many fields. To explore the values from data, researchers have made efforts to provide solutions to accommodate the big data applications. These applications not only require scalability on computation and storage but also bring a pressing challenge on data processing speed. If the data processing speed cannot catch up with the data generation speed, the computing results provided by the data analytic systems will be useless. Additionally, in many applications such as real-time recommendation system and web index maintenance, dataset involved in the computation needs to be frequently updated. To catch the change of individual data items and deliver the correct result reactively under the fast evolving dataset, we have to adopt online data analytic systems.

Unfortunately, there exists a gap between the state-of-the-art data analytic system and the requirement of the online data analytics. It prevents the current data analytic systems/algorithms from being adopted in the more challenging, yet more realistic, online scenarios. In this thesis, we identify three major requirements to move the conventional offline computing systems to online: (1) First, we need to build a fluent online processing pipeline which consumes input data reactively and with high throughput; (2) We desire a storage layer which serves online query and update to the computing state so that it can deliver the results adaptively against the fast evolving dataset; (3) Finally, while moving from offline to online, the data analytic systems should be backward compatible to its offline counterparts, hence, we need to keep the maximum compatibility with infrastructure, programming model, etc.

In this thesis, we study the solutions to fulfill these three requirements. (1) We design and implement the micro-batch-based online data processing pipeline with fine-grained resource allocation to avoid the per-record backup and handle the highly-skewed computing demands, e.g. videos with various length. (2) We build a parallel-friendly and high-performance computing state management system based on Locality Sensitive Hashing (LSH). Our system differentiates with the conventional LSH system in that it does not need to reconstruct itself to serve the online update requests and adopt multiple system-oriented optimization strategies to scale to large storage size and the multi-cores environment. (3) We achieve the scalable, efficient and reliable computing state management for online data analytics by introducing Resilient State Table (RST) as a component of Spark. So the system we have designed can seamlessly integrate with the programming model and scheduling policy of the original Spark framework.

We evaluate our design in various scenarios. Our design exceeds the performance of the state-of-the-art systems in many application domains, e.g. the content-based video indexing, high dimensional nearest neighbors search, real-time web index maintenance, and machine learning algorithms.

Abstract

Actuellement l'afflux de donnes produites par diverses sources, telles que les dispositifs de dtection mobile, les journaux d'vnements systmes, et les activits utilisateurs sur Internet, ne cesse de crotre. Par consquent, l'information enfouie dans ces donnes massives a un potentiel important de pouvoir amliorer nos connaissances et la faon d'aborder diffrents domaines. Les chercheurs ont donc oeuvr fournir des solutions innovantes pour explorer les informations contenues dans ces donnes massives. Ces applications requirent non seulement l'extension des capacits de calcul et de stockage, mais ajoutent un dfi important sur la vitesse de traitement des donnes. Si cette vitesse de traitement n'est pas en adquation avec la vitesse avec laquelle les nouvelles donnes sont gnres, les rsultats obtenus par les systmes d'analyse seront inutilisables. De plus, dans beaucoup de cas tels quels des systmes de recommandations en temps rel ou la maintenance de l'indexation de pages web, les ensembles de donnes impliqus dans les calculs doivent tre mis jour frquemment. Pour dtecter un changement dans un enregistrement individuel et dlivrer le bon rsultat de faon efficace dans un ensemble de donnes lui-mme sujet des modifications rgulires, nous devons passer la main des systmes d'analyse de donnes en ligne.

Malheureusement un foss demeure entre les systmes analytiques de pointe et les conditions pour leur utilisation sur des donnes en ligne. Ce dernier empche ces systmes/algorithmes d'tre utiliss dans les cas rels, mais plus complexes, des traitements de donnes en ligne. Dans cette thse, nous identifions trois prrequis majeurs pour basculer du modle conventionnel du traitement des donnes hors ligne au modle en ligne : (1) Premirement, nous devons construire un pipeline en ligne qui consomme des donnes d'entre rapidement, avec un dbit de sortie lev ; (2) Nous ncessitons un support de stockage qui rpond des requtes en ligne et qui est capable de mettre jour l'information qu'il contient malgr une frquence de modification importante ; (3) Finalement, tout en effectuant ces changements de hors-ligne a en ligne, le systme d'analyse doit conserver la rtro compatibilit avec ses quivalents hors-ligne, nous devons donc conserver une compatibilit maximale vis vis de l'infrastructure, des modles de programmation, etc.

Dans cette thse, nous tudions les solutions qui remplissent ces trois pr-requis. (1) Nous crons et implmentons un pipeline de streaming en micro-lots avec une allocation des ressources millimtre, vitant le cot de la sauvegarde par enregistrement et permettant de rpondre des demandes trs varies, ex : des vidos de taille variable. (2) Nous construisons un systme de gestion de la computation d'tats hautement paralllisable, bas sur le Locality Sensitive Hashing (LSH). Notre systme se diffrencie du LSH conventionnel car il ne ncessite pas de se reconstruire pour servir la requte en ligne et comprend des optimisations de stratgies systmes pour supporter la monte en charge du point de vue de l'espace de stockage et du nombre de coeurs de l'environnement. (3) Nous parvenons l'extensibilit et l'efficacit du systme de gestion de computation d'tats pour les analyses de donnes en ligne en y ajoutant les Resilient State Table (RST) en tant que composant de Spark. Le systme que nous avons bti peut donc s'intgrer aisment avec le modle de programmation et de planification du framework Spark original.

Nous valuons notre projet d'aprs diffrents scnarios. En effet, il surpasse les performances des technologies de pointe actuelles dans beaucoup de domaines, ex : recherche de contenu dans lors de l'indexation video, recherche multidimensionnelle des plus proches voisins, maintenance d'index en temps rel, et algorithmes d'apprentissage machine.

Acknowledgements

I am so fortunate to work with my advisors, Wenbo He and Xue Liu. They are both researchers with full of wisdom. They always give the brilliant input to push the research ideas to a higher level and bring the new insights to me when I am stuck in the difficult situations. I am thankful to them who guided me tirelessly throughout my PhD.

I am also indebted to my collaborators, Lei Rao, Yu Hua, Yixin Chen. The work presented in this dissertation is the result of the enjoyable collaboration with them. Chapter 2 is the joint work with Yixin Chen and Yu Hua. Chapter 3 also contains the valuable input from Yu. Chapter 4 is the joint work with Lei Rao. More broadly, I am fortunate to collaborate with Mingyuan Xia, Jie Liu, Yuxiong He, Sameh Elnikety during my PhD. Additionally, I enjoy these four years spending with my labmates, Wen Wang, Xi Chen, Landu Jiang, Fanxin Kong, etc.

Beyond direct collaborators on the projects here, I am also very fortunate to work with the open source big data and machine learning community. Tianqi Chen from the University of Washington got me to lead the development of XGBoost for JVM Platform, one of the hottest open source machine learning projects in the world. Cheng Lian, Matei Zaharia and Reynold Xin from Databricks guided me when I contributed to Apache Spark. I enjoy using what I learned to create high impact.

Last but not least, I want to thank my family for their unwavering support throughout my PhD. It is pretty hard to express my grateful heart to my parents who back me up for so many years and are always willing to do more. I also appreciate Haimei Li for her love and support. She is always patient when I was stressed, and she keeps me moving forward with full of motivations and makes my life beautiful.

Preface

This thesis describes the approach to move the offline data analytic applications to online, with the focus on the online data processing pipeline, high-dimensional computing state management and the integration with the current system. It is the first to address the specific topic on how to move from offline data analytic to online. Specifically, the online data ingestion system for video data, LSH-based computing state management for high-dimensional data and the stateful data analytics in Spark have not been discussed in the state-of-the-art research work.

The content in this thesis is also described in three published/to-be-published papers. They are listed in the Bibliography section as [121, 122, 123].

Paper [121] is a joint work with my advisor, Wenbo He and the collaborators, Yu Hua and Yixin Chen. In this project, I designed and implemented the major part of the system, and Yixin helped on the color distribution algorithm. Prof He and Prof Hua gave their input on the position of the idea and the writing of the paper.

Paper [122] is a joint work with my advisor, Wenbo He and Xue Liu and the collaborator, Yu Hua. In this project, I designed and implemented the system. Prof He, Prof Liu, and Prof Hua input their insights on the position of the idea and the writing of the paper.

Paper [122] is a joint work with my advisor, Wenbo He and Xue Liu and the collaborator, Lei Rao. In this project, I designed and implemented the system. Prof He, Prof Liu, and Lei Rao input their insights on the position of the idea and the writing of the paper.

Contents

1 Introduction **1**

1.1 Requirements of Online Data Analytics 3

1.2 Contributions 5

2 Background **8**

2.1 Design of Offline-oriented Frameworks 8

2.2 Design Patterns of Data Analytic Applications 12

3 Online Data Analytics in Practice **15**

3.1 Background 17

3.1.1 Online vs. Offline Content-based Video Indexing 18

3.2 Marlin Overview 20

3.3 Streaming Feature Extractor 22

3.3.1 Video Feature Selection 23

3.3.2 Distributed Streaming Feature Extraction 23

3.3.3 Resource-aware Data Stream Scheduling 25

3.4 Feature Indexing for Online 28

3.4.1 Distributed Inverted Index Structure 29

3.4.2 Content-based Search . . . 31
3.5 Implementation . . . 34
3.5.1 Feature Extraction with Resource-aware DStream . . . 34
3.5.2 Load Balancing in Distributed Feature Index . . . 34
3.5.3 Fault Tolerance . . . 35
3.6 Performance Evaluation . . . 36
3.6.1 Streaming Feature Extraction . . . 37
3.6.2 Distributed Feature Index . . . 39
3.6.3 Accuracy of Marlin . . . 40
3.7 Related Work . . . 41
3.8 Summary of this Chapter . . . 43

4 Computing State Management in Online Data Analytics 45
4.1 Background . . . 47
4.1.1 Locality Sensitive Hashing . . . 48
4.1.2 Online vs. Offline LSH . . . 50
4.1.3 LSH-Based State Management Layer . . . 52
4.2 Storage Design of PFO . . . 54
4.2.1 Hash Tables in PFO . . . 55
4.2.2 Hierarchical Memory System . . . 56
4.3 Online State Query/Update . . . 61
4.3.1 Adaptive Hash Tree . . . 61
4.4 Parallel-friendly State Management . . . 65
4.4.1 Parallel-friendly Indexing . . . 65
4.4.2 Concurrency Management . . . 67

4.5 Evaluation . . . 69
4.5.1 Evaluation Setup . . . 70
4.5.2 Test Hierarchical Memory System . . . 71
4.5.3 Test Parallel-friendly Design . . . 73
4.5.4 Test Accuracy . . . 75
4.5.5 Test Parameter Sensitivity . . . 76
4.6 Related Work . . . 79
4.7 Summary of this Chapter . . . 80

5 Online Data Analytic Framework 82
5.1 Background . . . 84
5.2 Resilient State Table . . . 87
5.2.1 Overview . . . 88
5.2.2 Programming with RST . . . 90
5.2.3 Resilient State Table within Spark . . . 93
5.2.4 Fault-tolerance . . . 96
5.3 Applications . . . 99
5.3.1 Refresh Index Against Fast-evolving Web Pages . . . 99
5.3.2 Asynchronous Machine Learning Algorithms . . . 101
5.4 Evaluation . . . 105
5.4.1 Compare with Spark . . . 106
5.4.2 Test Scalability of RST . . . 109
5.4.3 Test Integration of RST and Spark . . . 111
5.4.4 Fault Tolerance of RST . . . 112
5.5 Related Work . . . 114

5.6 Summary of this Chapter 116

6 Conclusion and Future Work **118**

6.1 Conclusion 118

6.2 Future Work 121

Publication List **123**

List of Tables

4.1 Analysis of Representative LSH systems and How PFO differentiates itself with them 51

4.2 Notations of PFO 53

5.1 Design space of Stateful Data Analytic Frameworks 86

List of Figures

1.1 Offline Web Index Update: Offline Web Index system has to update the index in batching, and for every running of the MapReduce, it has to scan through the entire database to reconstruct the index 3

1.2 Design Pattern of Online Data Analytics 5

2.1 Spark Architecture: the computing resources are managed by Master and Workers; Each Spark application consists of a Driver which serves as the schedules tasks and one or more Executors which gets tasks to run. 9

2.2 An Example of RDD: RDD is immutable. Users reflect the changes to the content in a RDD by calling the APIs like map(), reduceByKey() to create a new RDD. . 10

2.3 The updates on input (filled with grey in the figure) has to be reflected by redundant computation. 11

3.1 Architecture of Marlin: Marlin calculates the feature vectors of the keyframes and aggregates them to get the feature vector of the video. Marlin detects the similarity between the newly uploaded video and the existing videos to provide the optimization advice to the other systems. 20

3.2 Distribution of keyframe number in CC_WEB_VIDEO dataset. The number of the videos with the keyframe number ranging from 6000 - 10000 is too small to be visible but still larger than 0. 26

3.3 Mapping Video to Flow for DRR Fair Scheduling: Each video is mapped to flow, and the packet size is the keyframe number of the video. The key difference is that the "flow" here has only one packet 27

3.4 DRR Fair scheduling: $Video_2$ requires 150 keyframes which are larger than its current deficit value so that it is skipped for this scheduling round. However, it can be scheduled in next around as the quantum granted in this round will be accumulated to the next one. 28

3.5 Basic Inverted Index Structure: Each entry in the inverted index table maps to one dimension in the feature vector. The vectors which have non-zero value at the dimension resides in the list associates with that entry. V_0 = (1, 0, 0, 0, 1), V_1 = (1, 1, 0, 0, 0), V_2 = (0, 1, 1, 0, 0), V_3 = (0, 1, 0, 0, 0) 30

3.6 Sharded Inverted Index: We separate the entry lists in the inverted index into different shards according to the entry ID and assign each shard with a ShardLord, which is the only entity with the access permission to the shard. The communication between the ShardLords and the external system and the ShardLoad is via messages. We guarantee that the message queue of a ShardLord can only be accessed by a single thread at any moment 31

3.7 Scalabilty of Marlin: With 250 videos arriving with Poisson distribution, Marlin achieves 25X speedup with 16 cores and it takes around 52 seconds to extract features for all 250 videos. 38

3.8 Response Time with/without Resource-aware DStream Abstraction: The figure is generated by batches consisting of 50 arrival videos. We set quantum as 1 and DRR-based scheduler checks the scheduling candidates for every 1ms. 39

3.9 Scalability of Feature Index: with 16-cores setup, the speedup is 24x. The single-node setup (one core and four cores) suffers from the frequent happening of Garbage Collection 40

3.10 Processing Latency Under the Workload With Different Size 41

3.11 Accuracy of Marlin with different threshold: We can achieve 100% precision and meanwhile maintaining the recall as high as around 80%. We describe the vector feature with a 512-dimensional vector, which is concatenated with 256 bins for Hue, 128 bins for Saturation and 128 bins for Value. 42

4.1 Locality Sensitive Hashing. 49

4.2 Storage Memory Layer Design: PFO saves data in both RAM and Flash Memory (SSD-based). It consumes all I/O in RAM to improve the performance and saves snapshots (S_{ij}) in SSD for increasing the system capacity with the performance guarantee. Additionally, the off-heap space for each table is divided into multiple partitions (P_i) for exploit parallelism for processing. 54

4.3 Memory Layout in Off-heap space. In the figure, we only show the layout of MainTable which has vectors in Data segment for simplicity. For LSHTables, the content in Data segment is the IDs of the vectors. 59

4.4 Adaptive Hash Tree Structure: X(Y) in the Figure stands for that the original offset Y is replaced with X. In the hash tree, we progressively include more bits in the hash value of the vectors to locate them in the buckets. When there are more than t nodes under the same buckets we redistribute them to the next level of the hash tree. The leaf nodes containing only two fields means that it is the last node in its bucket. 62

4.5 Request Dispatching Module in PFO consists of a group of computing threads and multiple actors each of which maintains a hash tree as its state. 68

4.6 Read Latency of three types of single-layered design 72

4.7 Scalability of PFO in Multi-core platform (L = 1) 73

4.8 Scalability of PFO in Multi-core platform (L = 10) 74

4.9 Accuracy of PFO and LSB-Tree (k = 10) (the original paper of LSB-Tree did not report the performance of their system with more than 40 tables) 76

4.10 System Performance Against different C and m 78

4.11 System Performance Against different l and t 78

5.1 Architecture of Spark with RST 89

5.2 Programming Model of RST 90

5.3 Achieve State Locality and Reduce Remote State Access Under Spark Scheduling Policy 95

5.4 De-duplicate Updates from Restarted Jobs and Tasks. We keep track the lastest epoch and the last offset of the data entry producing the state updates for each RST partition. 98

5.5 Online Web Index Update Job Latency Comparison between Spark-Streaming-based and RST-based Stateful Implementation 107

5.6 Recommendation System Job Latency Comparison between Spark-based and RST-based Stateful Implementation 108

5.7 Converage Rate Comparison between Asynchronous Machine Learning implemented with RST + Spark and original Spark, RDD Partition Number = 40, RST Partition Number = 200, Step Size = 1, regParam = 0.01 109

5.8 Throughput and Latency of of RST in Distributed Servers 110

5.9 Throughput and Latency of RST against Different Workload Pattern 111

5.10 Throughput and Latency against Different Batch Size. 112

5.11 Checkpoint Impact to the Task Latency. We group the experiments into 5 groups, each whcih corresponds to a particular state size. Within each group, the left bar plot stands for the synchronous algorithm and the right one is the asynchronous algorithm adopted by RST. 113

5.12 Recovery Latency of Checkpoint and Recompute with 5GB State Against Different Dependency Chain Length. 114

Chapter 1

Introduction

The variety of the data sources and the size of generated data are dramatically increasing nowadays. Internet users are producing a vast amount of data across the web. For example, the users of Instagram share more than 80M pictures every day [54]. The Internet of Things (IoT) encompasses many aspects of human society, from connected cities, to connected cars and to sensors embedded in smart devices that track personal behavior [6]. The information hidden behind the massive size of data are changing our way to conduct operations in nearly all fields, e.g. scientific research [40], business [18, 83] and finance [111].

Not only the total volume of data is increasing, but also the velocity of the data generation is high and grows in a fast pace. The users of Facebook generates 50 million messages/sec or 5 trillion messages/day which are consumed by its infrastructure [95]. The Options Price Reporting Authority (OPRA), the organization taking in charge of aggregating all the quotes and trades from the options exchanges, stated that the peak message rates would be doubling every year since 2005 when the number was 122000/second [101].

Unfortunately, a lot of traditional data analytic applications, which are based on frameworks like Hadoop [47], Spark [100], cannot meet this requirement despite its prevalence. We refer the design pattern of these applications as *Offline Data Analytics*. We illustrate a web index system designed following the pattern of offline data analytics in Figure 1.1. The input of this application comes from the web crawlers which monitors the updates of the web pages and fetch them into a database/file system. A MapReduce job runs periodically to poll the database, parse all crawled pages, and calculate the in-links of each URL for further URL score calculation. The results of this MapReduce job is saved in another database. Other MapReduce jobs run in a similar way to conduct the further processing.

Design Pattern of Offline Data Analytic Applications In general, an offline data analytic application consists of a set of batching jobs which are chained through the database or file systems. Specifically, we summarize the design characteristics of offline data analytics as following:

— ***Input***: in the input part of each processing phase, offline data analytic applications accumulate data in a database/file system and then conduct further processing;

— ***Output***: in output part, offline data analytic applications generate batching output and assume that the results are write-once-read-many-times.

These two characteristics bring the low application effectiveness in several aspects. A too long interval between two polling on the database results in the stale results serving to the users. Shorten the interval length does not resolve the issue. When the newly generated data contains the dependency to the existing data, e.g. the newly crawled web page contains the link to the existing page [85], it requires batching jobs to traverse over the complete database which brings unexpected redundant computation.

In this thesis, our goal is to propose a new design pattern of data analytic applications, which resolves the challenge in not only handling data with the large volume but also with

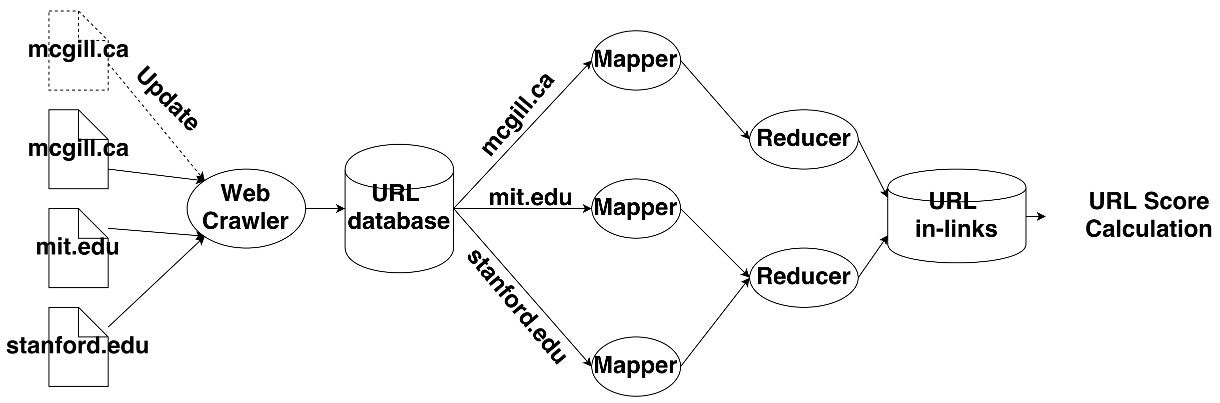


Figure 1.1 – Offline Web Index Update: Offline Web Index system has to update the index in batching, and for every running of the MapReduce, it has to scan through the entire database to reconstruct the index

the high velocity. We refer the new design pattern as "Online Data Analytics".

1.1 Requirements of Online Data Analytics

Streaming Data Feeding The first requirement to online data analytics is to replace the "accumulate-first" data processing workflow with the streaming data feeding. The offline data analytic applications take the input as bulks and read them at once at the beginning of processing. To consume data online, online applications treat the input as continuous data streams and apply the logic to the incoming data streams reactively. The state-of-the-art streaming processing engines [106, 23, 116] impose First-Come-First-Serve (FCFS) order of the elements in the data streams. However, we argue that the application-specific requirement has to be considered when building data streams. For example, when the computing resource demand is highly skewed among the elements in the data streams, the FCFS order may bring the compromised system performance. On the other side, the application-specific requirement is various, e.g. FCFS is mandatory in scenarios like online time-series analysis.

High-Performance Computing State Management Layer To avoid redundant computation which exists in offline data analytic applications, the online data analytic applications have to save the intermediate results in a scalable and flexible data structure for incremental computation. We refer the intermedia results as the computing state. The computing state management layer has to accept not only query but also update requests in an online manner and contains the parallel-friendly design to utilize the computational capacity fully in the modern CPUs. The most challenging part comes from the potential complicate structure of the computing state, and the tools to handle this complexity are designed with the offline computation assumption. Therefore, we have to refactor these tools to fit into the scenario when building computing state management layer.

Minimize the Cost For Migration A lot of existing applications are built with the frameworks which do not offer the facilities to achieve online data analytics. While we can invent new frameworks with the full-fledged or partial support to online data analytics [11, 68, 87, 15, 46, 49, 37, 77, 1], we have to re-write the existing applications with the new APIs of these new frameworks and introduce additional complexity to the infrastructure. Therefore, the third requirement for us to achieve online data analytics is on the cost side, i.e. how to minimize the changes applied to the existing infrastructure, programs and users' habit on the programming model.

In Figure 1.2, we illustrate the overview of online data analytics. Being different with the conventional offline design, the Extract-Load-Transform (ETL) and data analytic phases are integrated into a streaming pipeline. In the input phase, online data analytics enables the application-specific stream scheduling logic. The output of the online data analytic applications are generated by computing over both the input data and the computing state entries stored in the computing state management layer.

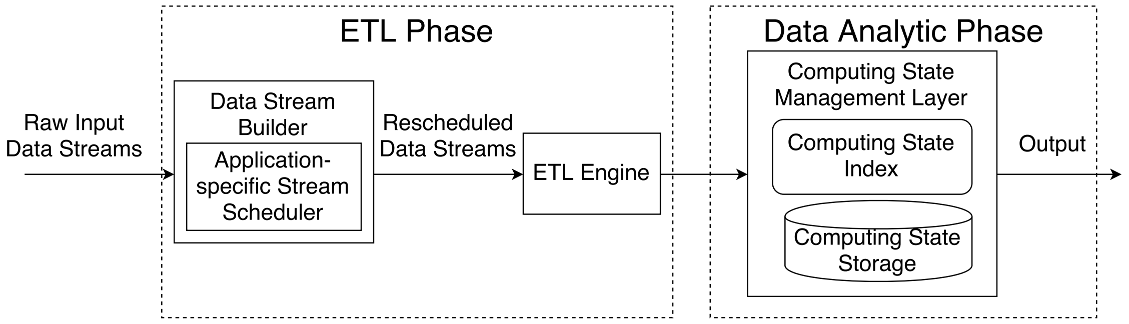


Figure 1.2 – Design Pattern of Online Data Analytics

1.2 Contributions

In this thesis, we make the following contributions to fulfill the above requirements:

— We propose the design pattern of online data analytic applications. We realize online data analytics in practice within the context of content-based video indexing. We refactor the conventionally offline content-based video indexing system and build a complete online processing pipeline from feature extraction to feature index for videos [121]. We introduce a streaming data feeding component which organizes the input video streams as micro batches and allocate the computing resources among videos with a credit-based algorithm to prevent compromised system performance due to the skewed resource demand. We use a distributed feature index which enables incremental index update.

— We design and implement the computing state management layer with the focus on enabling online update and parallel processing. To overcome the challenges due to the complicated structure of computing state entries, we study the solution by taking the high dimensionality as an example, i.e. the computing state entries are represented as high-dimensional vectors. We build a Locality Sensitive Hashing (LSH) based state management layer aiming to provide the fine-grained and low-latency online query/update to the high-dimensional unstructured state entries [122]. The LSH-based index is conventional to be built in an offline manner as it has to be reconstructed under online updates [104, 39, 67]. The proposed system contains a hierarchical memory system consisting of RAM and flash memory. The RAM space consumes all updates to avoid write persistent memory space in random and the hardware feature of flash memory fills to performance gap between the sequential and random read. Additionally, we use a Partitioned Hash Forest structure to improve the system performance in the multi-cores environment.

— To minimize the overhead to port the existing data analytic applications built on top of the frameworks without the full support of online data analytics, we introduce Resilient State Table (RST) [123] into one of the most popular data analytic framework, Spark. The existing efforts to achieve online data analytics either fall short in providing the low-latency and fine-grained state access [11, 49, 13] or imposes new programming model which is incompatible with the existing data processing frameworks [77, 37, 15]. RST not only offers the full-fledged online data analytic functionality but also integrates with the programming model and scheduling policy of the original Spark framework.

The rest of the thesis organizes as follows: We introduce the background knowledge in Chapter 2. In Chapter 3, we apply the online data analytics in the context of content-based video indexing. In Chapter 4, we present the computing state management layer for the online data analytics in the context of the high-dimensional unstructured data. We then describe how we achieve online data analytics with the backward compatibility to the existing data processing frameworks in Chapter 5. Finally, we summarize the contributions of this dissertation and discuss the future directions of this research topic in Chapter 6.

Chapter 2

Background

In this chapter, we introduce the background knowledge which facilitates the understanding of this dissertation. In Section 2.1, we describe the design of the state-of-the-art data processing frameworks and highlight why the existing applications based on these frameworks do not fit into the online scenario. In Section 2.2, we compare various design patterns of data analytic applications with the proposed design, "online data analytics".

2.1 Design of Offline-oriented Frameworks

In this section, we introduce the design of data analytic frameworks with the example of Apache Spark [100]. Apache Spark is one of the most prevalent data analytic frameworks, and it is representative in its design idea which targets to process large data volumes [38, 47]

Figure 2.1 shows the architecture of Spark. Spark follows the Master-Workers design style in which Master allocates computing resources in Workers to the applications. The applications utilize the resources of workers by starting Executors to execute computing

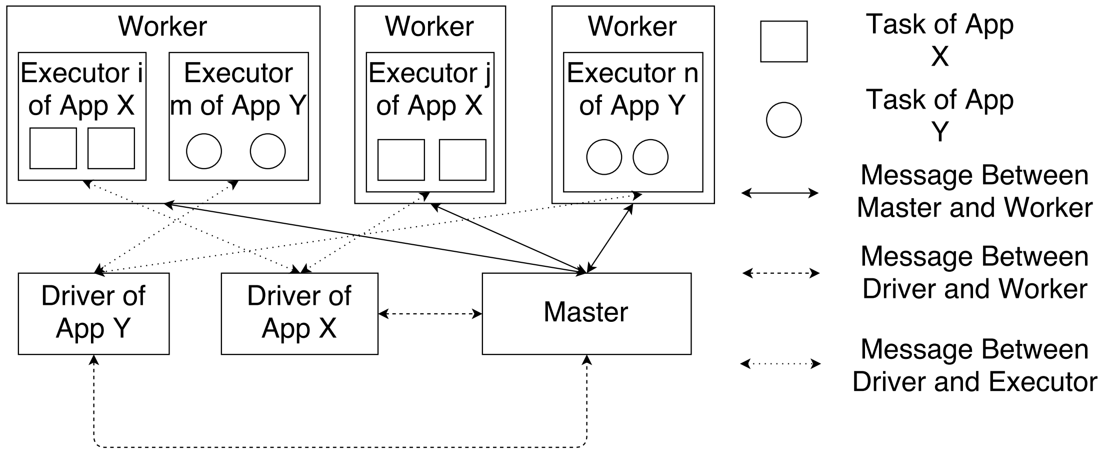


Figure 2.1 – Spark Architecture: the computing resources are managed by Master and Workers; Each Spark application consists of a Driver which serves as the schedules tasks and one or more Executors which gets tasks to run.

tasks. Driver, as the per-application scheduler, talks with Master to acquire resources and dispatches the tasks to the executors.

The core component in Apache Spark is an in-memory abstraction, called Resilient Distributed Dataset (RDD) [115]. RDD provides the fast and reliable in-memory storage for data, and the interfaces to the users to scale the computations over RDD to the commodity cluster. All data involved in Spark-based computation are wrapped by RDD. We give an example in Figure 2.2 to show the execution logic of the following Spark program:

```
1  val wordRawRDD = sc.textFile("inputFile").flatMap(_.split('␣'))
2  val wordAndFrequencyRDD = wordRawRDD.map(word => (word, 1))
3  val aggFrequencyForEachWord = wordAndFrequencyRDD.reduceByKey((
       freq1, freq2) => freq1 + freq2)
4  val result = aggFrequencyForEachWord.collect()
```

Line 1 reads a text file from the file system and splits each line at the space to form an

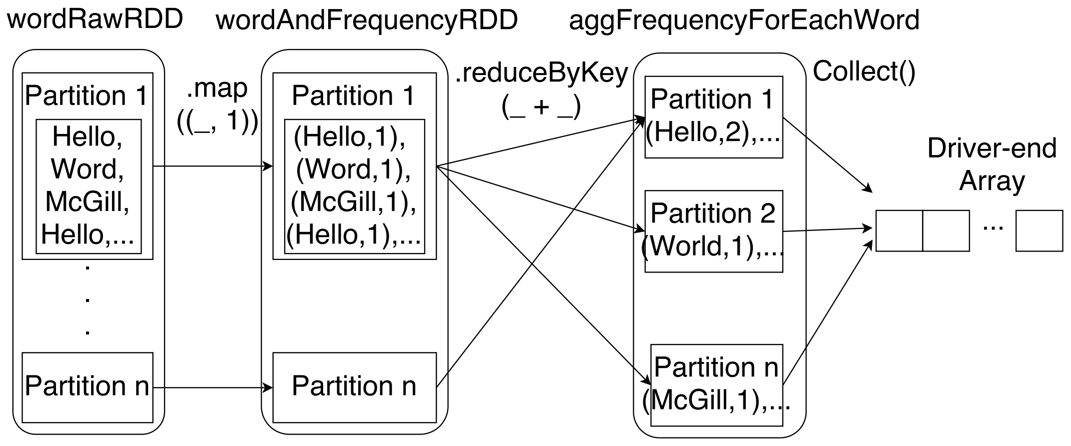


Figure 2.2 – An Example of RDD: RDD is immutable. Users reflect the changes to the content in a RDD by calling the APIs like map(), reduceByKey() to create a new RDD.

RDD wrapping a bunch of words. The following Line 2 maps each word into a (word, 1) pair. We then call reduceByKey() in Line 3 to aggregate the frequency of each word. The elements in RDD are assigned to a specified partition by a "Partitioner". The mapping relationship between two RDDs is in the unit of partition. The partition-to-partition dependency can be one-one (e.g. the dependency constructed by calling map() in Line 2) or one-to-many/many-to-one (e.g. the dependency created by calling reduceByKey()).

Programming with RDD is attractive for scaling the data processing to multi-cores and distributed servers. The processing of each partition in Figure 2.2 is indepedent so that they can be conveniently parallelized. RDD is immutable so that users do not need to worry about the concurrent mutation which brings most complexity in parallel programming [50]. In the above example program, when we call map() and reduceByKey() we do not change the content in wordRawRDD and wordAndFrequencyRDD respectively, but create the new RDDs, resulting that there is no concurrent threads modifying the same memory space.

Gap Between Offline Framework Design and Online Data Analytics However,

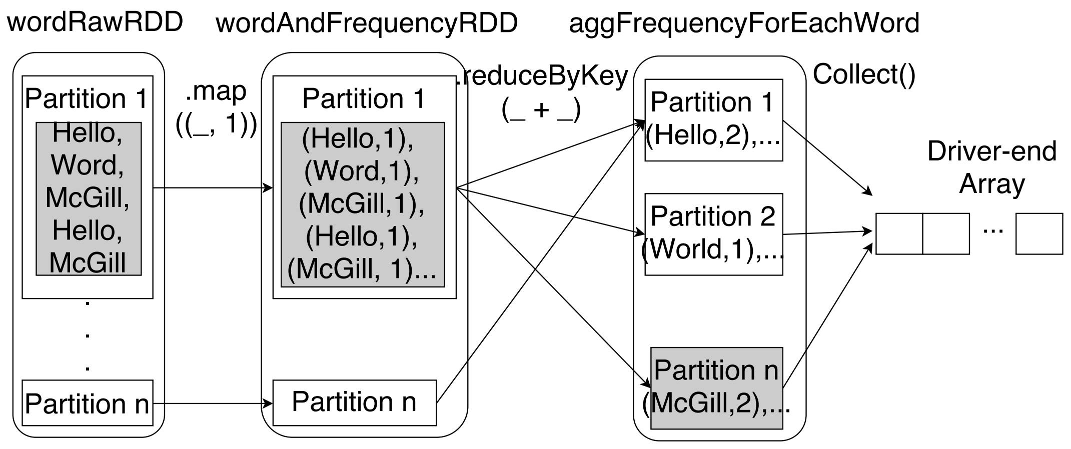

wordRawRDD
wordAndFrequencyRDD
aggFrequencyForEachWord
Partition 1
Hello, Word, McGill, Hello, McGill
.map ((_, 1))
Partition 1
(Hello,1), (Word,1), (McGill,1), (Hello,1), (McGill, 1)...
.reduceByKey (_ + _)
Partition 1 (Hello,2),...
Partition 2 (World,1),...
Partition n (McGill,2),...
Collect()
Driver-end Array
Partition n
Partition n

Figure 2.3 – The updates on input (filled with grey in the figure) has to be reflected by redundant computation.

the design of RDD makes Spark does not fit into the scenario of online data analytics. First, Online Data Analytics requires incremental computation against updated input which cannot be fulfilled with Spark. In Figure 2.3, we update the first input partition of the original text file in Figure 2.2 by adding a new word, "McGill". To update the frequency of each word in the text file, we have to re-apply the computation over the complete dataset, leading to the considerable amount of redundant computation. Second, to avoid redundant computation, we have to save the current frequency of each word, i.e. computing state in this case, and only mutate the involved computing state for each update on the input. This requirement does not match with the design of RDD. Due to the immutability of RDD, Spark reflects the changes applied to the elements in RDD by creating a new RDD with APIs like map(). If we save computing state in RDD, we have to create a new RDD for every fine-grained update on the computing state, causing significant overhead.

From the above analysis, we can see that frameworks like Spark are designed for coarse-grained computation. The similar design philosophy can be found in Hadoop [47], Flink [38], etc. Therefore, the data analytic applications based on these frameworks cannot fulfill the requirements of online data analytics.

2.2 Design Patterns of Data Analytic Applications

The researchers and the developers are making efforts to overcome the limitations in offline data analytics. In this section, we compare different design pattern of data analytic applications which are proposed for this purpose. We focus on three aspects of various design patterns:

— **Input** The input format of different design patterns decides how fast the applications can consume the newly arrived data.

— **Output** Output phase decides how efficiently the users can get data analytic results, e.g. whether introduce redundant computation, etc.

— **Compatibility With Offline Applications** We consider how many changes we have to introduce to the offline applications to move them from offline to the new design pattern.

Input To avoid the accumulate-first strategy in offline data analytics, researchers propose the *Streaming Processing* pattern which takes continuous streaming as input. The streaming processing frameworks like Storm [106], Samza [23], enables the applications following this pattern to operate in low latency. However, the state-of-the-art streaming processing pattern assumes that the processing overhead of all elements are similar so that the applications process the elements in First-Come-First-Serve (FCFS) order. This assumption does not hold for several scenarios. As we will introduce in Chapter 3, the video data contains highly skewed computing resource demand, and FCFS processing order brings sub-optimal resource allocation in this case. *Online Data Analytics* include an application-specific stream scheduler to reorganize the order of the elements in data streams and then feed into the system.

Output To eliminate redundant computation in offline data analytics, researchers propose several strategies to produce output incrementally. *Incremental Computation* eliminates the redundant computation by keeping the results from the preivous data analytic jobs and derive the contemporary results based on that. However, the applications in this category are still based on the batched processing. For example, Nectar [46] saves the results of previous batching jobs and associate the current job with the previous ones through hash tags to achieve incremental computation. The reason for this type of design is the lack of a computing state management layer which supports low-latency and fine-grained computing state access. *Stateful Computation* is a category of the applications which are based on the data analytic frameworks with the special design of computing state management layer. Depends on the frameworks design, the applications in this category have various compromising on the functionality, e.g. cannot scale to large state size, etc. Some streaming processing frameworks also contain computing state management component but usually suffer from compromised on scalability to large state size [23, 106]. Some of the frameworks [87, 37] provide the support of full-fledged stateful computation, including low-latency, scale to large state size, etc. We will have a comprehensive study of these frameworks in Section 5.1. Online data analytics requires the full-fledged computing state management.

Compatibility with Offline Data Analytic Applications Compatibility with the offline data analytic applications is critical to apply the design patterns in practice in terms of cost. Streaming processing is significantly different with the offline data analytics therefore does not support a smooth migration from offline data analytic to streaming processing applications. Applications in the category of incremental computation are usually based on the customized version of the original offline frameworks. Therefore, they provide the compatibility with the original offline applications while the processing efficiency of them suffer on their batching nature. The frameworks providing the full-fledged computing state management usually adopts an imperative programming model to simplify the operations to the state entries as put/get APIs. As we stated in Section 2.1, offline data analytic applications usually embrace the functional programming model to facilitate parallelization. Discarding the functional programming model means that we need to rewrite all applications with the new programming model. Additionally, imperative programming model would make the design of the critical processing stage like Extract-Transform-Load in applications over-complicated. Online data analytics masks the imperative operations with the high-level abstraction and is compatible with the functional programming model to reduce the overhead to migrate from offline to online data analytics.

Online Data Analytics is the design pattern of data analytic applications proposed in this thesis. Online data analytics takes continuous data streams as input and relax the constraint on the processing order of the elements in the stream. Therefore, the application-specific requirement can be embedded into the input phase. Online data analytics not only requires a full-fledged computing state management layer, but also pursue the backward compatibility to the offline data analytic applications.

Chapter 3

Online Data Analytics in Practice

In this chapter, we present the design pattern of online data analytic applications in the context of content-based video indexing. The state-of-the-art content-based video indexing system is usually designed to be offline. The challenge of porting the content-based video indexing system to online is that the video data applies much higher computational overhead than the other types of data due to its size and complexity. This higher computing pressure involves both the feature extraction and indexing part. Additionally, the video clips have various length and different number of keyframes that brings more challenges because the bigger/longer video may block the online pipeline and starve the others. We address these challenges by building a distributed streaming processing pipeline based on the novel resource-aware micro-batch model, and the feature index which consumes the output in feature extraction stage online without the necessary to reconstruct the index in an offline manner.

We describe the motivation and background knowledge for this chapter in Section 3.1 and then move forward to the overall architecture of Marlin in Section 3.2. We go into the details of feature extraction system and content-based index of Marlin in Section 3.3 and 3.4 respectively. We describe the implementation details in Section 3.5. Section 3.6 demonstrates the evaluation results, and Section 3.7 discusses the related work. We summarize the design of the system and discuss how to extend it to the other fields in Section 3.8.

3.1 Background

In the era of big data, various forms of data including trillions of text documents, billions of images, videos and audios and a huge amount of data on user activates (e.g., queries, page clicks, user relations) are transmitted, stored and generated in the cloud. Among of them, video data are dominant in volume. According to Cisco, the Internet video is expected to account for from 78% in 2014 to 84% in 2018, of all U.S. Internet traffic [22]. However, it is super challenging to provide useful insight in search, sharing, storage, transfer and piracy detection for video data.

A common but imperfect way to tackle the complexity of video data is to use the manual tagging with metadata, then apply Natural Language Processing (NLP), or the other text analytics methods for structured metadata to study the unstructured video data. There are two drawbacks to this approach: (1) The performance of online video systems depends on the quality and completeness of the metadata, however, having humans manually annotate videos in a large online video system is labour-intensive, time-consuming and error-prone; (2) the contemporary online video systems trust the users who upload the video clips to annotate the videos. A user may intentionally use deceptive descriptions to a video in order to gain popularity. It is very common that identical video clips have different sets of tags associated with them, and the video clips with the identical set of tags can be significantly different. As reported by Wu et al. in [112], by searching the 24 most popular keywords on YouTube, Yahoo! Video and Google video, around 27% online videos are duplicate or near-duplicate.

Hence, *Content-based* index and search are not only necessary but also fundamental in big video data research. It will benefit a variety of applications, including storage optimization [73], search result diversification [33, 93], polluted tag detection [61], and piracy detection, etc.

3.1.1 Online vs. Offline Content-based Video Indexing

Researchers have proposed various approaches to index videos based on their content instead of metadata. For example, authors of [118, 94] have managed to incorporate the temporal and spatial aspects of information into features, It has been reported that the computation of the sophisticated features is extremely time-consuming [98]. Therefore, the conventional video retrieval systems rely on the offline computation to extract features. For instance, MapReduce-based algorithms are widely adopted for multimedia feature extraction [117, 19, 66]. As we introduced in Chapter 1, while MapReduce-based approaches parallelize the data processing, the user has to endure the high latency for each job and the stale system status between two running instance of the offline jobs. Nowadays, the extreme volume and astonishing expanding rate of online video sharing and hosting systems (e.g., YouTube) require the online processing of the feature extraction. According to YouTube, over 300 hours videos are uploaded in a minute [114]. Hence, a fast video feature extraction system is in great demand to *make the feature extraction catch up with the video upload speed.*

In the setting of offline-based feature extraction, the content-based index design also assumes that the content of the index is updated in batching and offline manner. In this case, the entire index structure has to be reconstructed to include the newly uploaded videos to the index. In contrast, to achieve the online content-based video indexing, we need to eliminate the build-in-offline assumption and make the indexing structure updatable in an online fashion.

In this chapter, we describe an online content-based video indexing system for large-scale video sharing hosting platforms, called *Marlin* [121]. It spreads computing tasks, including feature extraction and indexing query/update, to distributed servers. The feature extraction and indexing operations are conducted as a pipeline. Once the feature of a video is extracted, the features are accessible for content-based search. Meanwhile, Marlin optimizes the computational resource allocation to speed up feature extraction. Specifically, we make the following contributions in this chapter:

— Streaming Feature Extraction: The feature extraction system in Marlin handles the streaming data in a micro-batching fashion [49, 68]. The videos are separated into small batches, and each batch contains videos arriving in a fixed time interval. Our system parallelizes the feature extraction of a certain video at keyframe level and further aggregates the keyframe features to get the feature representation of the videos.

— Fine-grained Resource Allocation: The existing streaming processing frameworks [106, 23, 116] apply the *First Come First Serve* scheduling policy. However, under such scheduling policy, a video with a large number of keyframes may take all resources, thus preventing the processing of the other videos in the same batch and even in the later batches. To increase the system throughput, we improve the current micro-batch mode with the fine-grained resource allocation mechanism based on the fair queuing algorithm. We model each video to a network flow, where the keyframe number of a video is considered as the packet size in a flow. We employ an $O(1)$ algorithm to prevent the unfair resource allocation and improve the overall system throughput (Section 3.3.3).

— Online Content-based Indexing: The feature index in Marlin offers two-folded benefits: 1) it provides the capability of distributed and parallel processing so as to accelerate the content-based search; 2) it incrementally updates the feature index with the newly arrived videos to avoid rebuilding the entire index, and pipelines with the feature

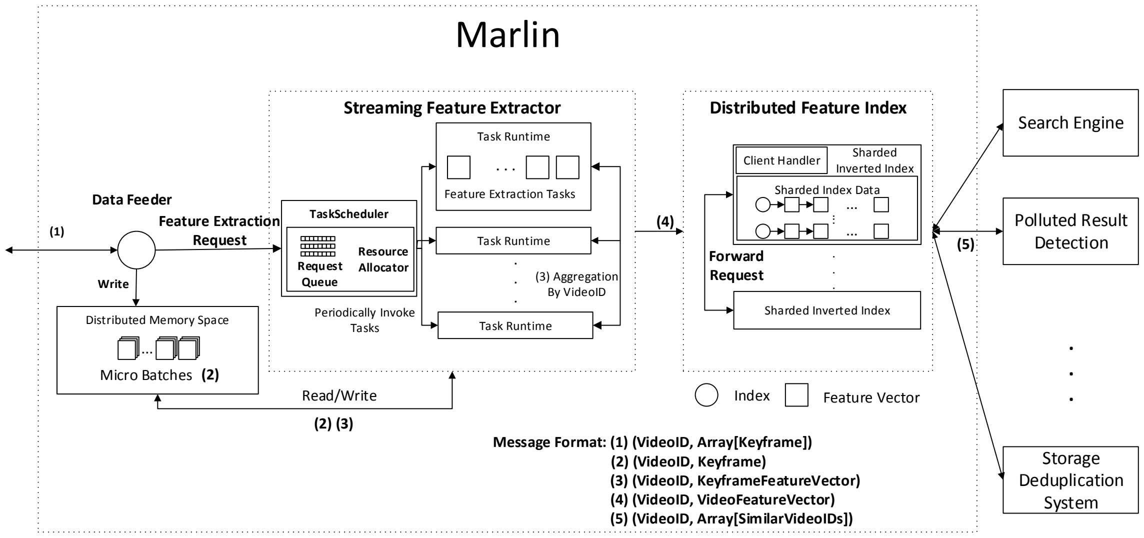


Figure 3.1 – Architecture of Marlin: Marlin calculates the feature vectors of the keyframes and aggregates them to get the feature vector of the video. Marlin detects the similarity between the newly uploaded video and the existing videos to provide the optimization advice to the other systems.

extraction without waiting for the completion of feature extraction for all the videos in a batch.

3.2 Marlin Overview

We illustrate the architecture of Marlin in Figure 3.1. Marlin consists of three components:

— **Data Feeder** Data Feeder takes video clips containing sequences of keyframes as input, and places them in the distributed memory space directly before running feature extractor algorithm. Rather than placing the keyframes to distributed file systems (e.g. HDFS [97]) then loading them into the memory space, it saves the I/O cost greatly. In the case of that the size of videos exceeds the available memory space, the

Least-Recently-Used (LRU) rule will be adopted to spill the least recently used video data to the disk.

— **Streaming Feature Extractor** Streaming Feature Extractor consists of a TaskScheduler and a group of TaskRuntime processes running on the distributed servers. The TaskScheduler distributes the two types of tasks, feature extraction and feature aggregation, to the TaskRuntime processes. The feature extraction task takes a keyframe and computes its feature vectors. The aggregation task yields video features by aggregating the vectors across the servers. We separate the continuous video streams into micro batches [49], i.e. we periodically deliver a set of videos for processing and the video set in each period is referred as a micro batch. Being different with the conventional micro batch model, the TaskScheduler assigns a video into a micro-batch according to not only its arrival time but also the number of keyframes, i.e. the demand on the computing resources. To prevent a video containing a large number of keyframes from overtaking resources and starving other videos, the TaskScheduler calculates the fair share of each video in every scheduling period and picks the videos whose computing demand are under their fair share to the next micro batch.

— **Distributed Feature Index** Distributed Feature Index is the index system storing the video feature vectors. It supports fast query as well as the incremental updates. Upon the feature vector of a video is calculated, it is sent to the distributed index system immediately, no waiting for the features of other videos in the same batch to be extracted. This design enables the feature vectors to be accessible to the content-based query once they are calculated. The immediately accessible feature differentiates Marlin with the batch processing approaches proposed in [108, 34], i.e. we virtually eliminate the explicit division of feature extraction and content-based index building phases.

The above three components map to the components in Figure 1.2. Data Feeder and Streaming Feature Extractor map to the streaming-based ETL engine and application-specific stream scheduler. Distributed Feature Index corresponds to the computing state management system where it indexes the video feature vectors according to their content to serve the query and update to the index.

In an architectural view, the major differences between Marlin and the other near-duplicate detection systems include 1) it processes everything in an online manner, eliminating the assumptions of pre-calculated features, static index, and explicit phase division; 2) it utilizes the in-memory and parallel computing to accelerate the data processing. In the following sections, we elaborate the design of Streaming Feature Extractor (Section 3.3) and the content-based Index (Section 3.4) respectively.

3.3 Streaming Feature Extractor

In this section, we present how we port the traditionally offline feature extraction to a streaming computing system. We discuss three topics involved in this process: a) select a simple yet efficient feature to represent the video to reduce the computational overhead for each video (Section 3.3.1); b) scale the streaming feature extraction to distributed servers (Section 3.3.2) and c) allocate the computing resources among the videos to prevent the big video clips occupying too many resources thus suffering the fluency of the online processing pipeline (Section 3.3.3).

3.3.1 Video Feature Selection

In pursuit of accuracy for content-based video indexing, video features are designed to be more and more complicated [112, 118, 94]. However, with the complex features, it is computationally expensive to extract the features online [98]. Also, the high complexity may make the parallelization pipeline paralyzed [17]. In this chapter, we adopt a simple feature, HSV color distribution, to represent a video. The HSV color distribution describes a video in three dimensions, Hue(H), Saturation(S), or Value(V), each of which is mapped to a histogram. The number of bins of these histograms is adjustable for accuracy tuning.

Simple features, like HSV color distribution, are efficient but less accurate in the content search. To compensate the accuracy loss, we scale the feature description to higher resolution, and then use distributed inverted index structure described in Section 3.4 to ensure the fast data access and efficient feature comparison.

3.3.2 Distributed Streaming Feature Extraction

Building a streaming processing system is challenging, especially that we need to guarantee the zero data loss with the high-speed data streams. Traditional stream processing frameworks either replicate data in multiple servers or backup the upstream data to prevent the data loss caused by the unexpected termination of the data processing jobs. However, in online video sharing and hosting systems, the multiple copies of the large volume of video data will incur large software/hardware overhead, and the upstream backup imposes the whole system to pause the processing and wait for a single point to replay all messages from upstream sequentially.

In this chapter, we adopt DStream abstraction [116] to implement our streaming video feature extractor (we also improve DStream on resource management which will be introduced in Section 3.3.3). DStream organizes the data streams as micro batches. Each micro batch is a set of videos and processed with parallel computing tasks. When processing each batch, DStream maintains data dependency as a directed acyclic graph (DAG). In the case of data loss, it recovers the missing data by traversing the DAG from the latest checkpoint to the missed data without the necessary to maintain multiple copies of each video.

The other important strategy to speed up the feature extraction is parallelization, in which the granularity of the task greatly affects the parallel performance. Too fine granularity will bring large communication overhead for aggregating the results across over the network to get the final video feature. According to [21], the overhead of network communication in parallel computing framework has been one of the dominating factors reducing the efficiency large-scale data processing. Too coarse granularity, e.g. organizing the feature extraction of a complete video as a single task, is not a wise choice either. Because a large amount of computation need to be completed before a task releases the resources, coarse granularity makes computing task management more challenging. For example, in speculative execution model [30], we choose to predictively start multiple copies of the slow tasks to mitigate the influence of abnormal server conditions. However, with multiple duplicates of computationally intensive tasks, a system will suffer from super high overhead. In this chapter, we choose keyframe-level parallelization, so that the individual keyframe feature vectors in a video is computed in parallel and then aggregated into video feature vectors according to the video ID (Message 3 and 4 in Figure 3.1) in parallel.

Being different with the conventional multimedia indexing system [94] which has explicitly separate feature extraction and indexing phases, Marlin does not wait for the completion of feature extraction of all videos in a batch before it performs the content-based search. Instead, with the support of the incrementally updatable index design which will be introduced in Section 3.4, a video feature vector is sent to the Online Feature Index as soon as it is computed.

3.3.3 Resource-aware Data Stream Scheduling

The conventional DStream does not consider the various computing resource demand among the arriving data objects, which may degrade the system throughput with the diverse resource demands of the data objects. In the original DStream abstraction, the system creates each micro batch by including the videos arriving at the system within a certain time interval and in the First Come First Serve (FCFS) order. However, with the keyframe-level parallelism, the demand for the computing resources of each video is proportional to its keyframe number. As shown in Figure 3.2, the distribution of the keyframe number in the standardized dataset CC_WEB_VIDEO [113] exhibits as a long-tail shape. If we simply assign videos to micro batches in FCFS order, a large video will easily overtake the computing resources. For instance, if a large video with thousands of keyframes arrives before short videos with less than ten keyframes, the large video will starve other short videos by taking all the computing resources and in turn to affect the system throughput.

In this chapter, we propose a *resource-aware DStream abstraction* to address fair resource allocation and improves the system throughput. We differentiate with the original DStream by introducing a resource allocator in the TaskScheduler of the streaming feature extractor. Instead of FCFS, we assign videos to the next micro batch according to the video size as well as arrival time. Our scheduling algorithm is an analog to the fair queuing problem in the network domain. Each video is mapped to a network flow with varying packet size (number of the keyframes), and the computing resources (CPUs and Memory) are mapped to the link bandwidth. Then our goal is to allocate bandwidth among the flows with the various packet

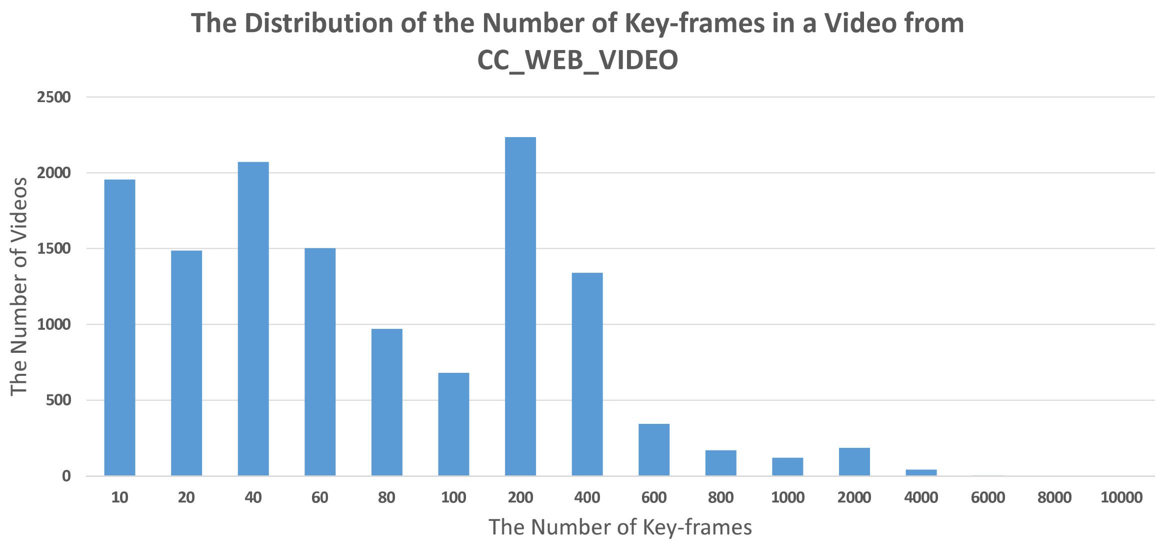


Figure 3.2 – Distribution of keyframe number in CC_WEB_VIDEO dataset. The number of the videos with the keyframe number ranging from 6000 - 10000 is too small to be visible but still larger than 0.

size and ensure that the large flows will not starve the small flows and still capture their chances to be processed.

Our algorithm is inspired by the Deficit Round Robin (DRR) algorithm in network field [96]. DRR algorithm assigns flow to queues and assigns a quantum of service to each queue in a round-robin fashion. Only the queues whose accumulated quantums are larger than the size of its packet will be served. The flows with the large packets are inclined to be back off. For each round, the size of the sent packet is subtracted from the quantum, and the remaining quantum value is accumulated to the next round. Thus, the system keeps tracking the deficits, and the queues that were short in the current round are compensated in the next round.

In the scenario of video feature extraction, videos are mapped as flows while the key difference is that the "flow" in our system has only one packet. We assign flows to a particular

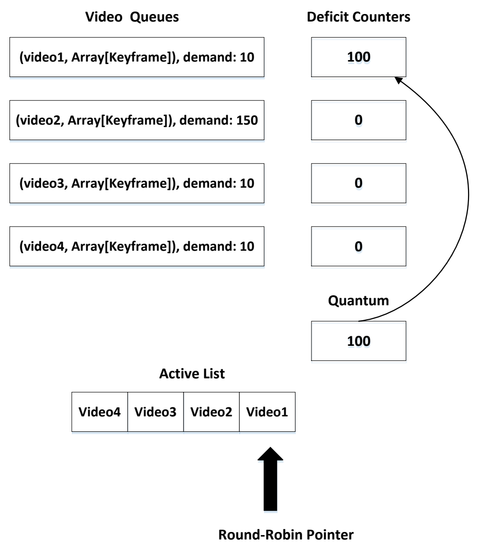

Video Queues
Deficit Counters
(video1, Array[Keyframe]), demand: 10
100
(video2, Array[Keyframe]), demand: 150
0
(video3, Array[Keyframe]), demand: 10
0
(video4, Array[Keyframe]), demand: 10
0
Quantum
100
Active List
Video4
Video3
Video2
Video1
Round-Robin Pointer

Figure 3.3 – Mapping Video to Flow for DRR Fair Scheduling: Each video is mapped to flow, and the packet size is the keyframe number of the video. The key difference is that the "flow" here has only one packet

logical queue which represents different levels of the computing resource demand. The unique ID of the queue is added to an active list when the video is received by Data Feeder (see Figure 3.1). When the queue ID is at the head of the list, we accumulate the current deficit of the queue $Deficit$ and the service quantum $Quantum$. If $Deficit + Quantum$ is larger than the size (the number of the key frames) of the video, we assign the video to the next micro batch to be processed. Otherwise, we skip the current queue and move to the next one. After being processed, the queue ID will be removed from the list.

Figure 3.3 shows a scenario of DRR scheduling where $Video_1$ with only ten keyframes can be scheduled in this scheduling round. Figure 3.4 shows another scenario where the required processing capacity is larger than the available resources. $Video_2$ requires 150 keyframes which are larger than its current deficit value so that it is skipped for this scheduling round. However, it can be scheduled in next around as the quantum granted in this round will be

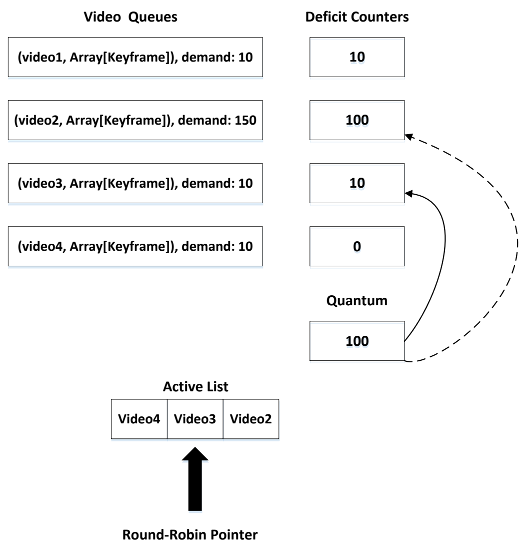

Video Queues
Deficit Counters
(video1, Array[Keyframe]), demand: 10
10
(video2, Array[Keyframe]), demand: 150
100
(video3, Array[Keyframe]), demand: 10
10
(video4, Array[Keyframe]), demand: 10
0
Quantum
100
Active List
Video4
Video3
Video2
Round-Robin Pointer

Figure 3.4 – DRR Fair scheduling: $Video_2$ requires 150 keyframes which are larger than its current deficit value so that it is skipped for this scheduling round. However, it can be scheduled in next around as the quantum granted in this round will be accumulated to the next one.

accumulated to the next one. In this way, we ensure that the scheduled according to their size, but will no be backed off infinitely just because it is large. We evaluate the resource-aware DStream abstraction in Section 3.6.1, where it shows the 5x overall throughput improvement brought by the adjustment of the processing order of the videos.

3.4 Feature Indexing for Online

After the videos are represented as feature vectors, we can search the videos based on the content by calculating the similarities between the query vector and the existing feature vectors, then respond with the videos whose vectors are similar to the query one. Two videos are considered to be similar if their similarity value is beyond the predefined threshold (Message (5) in Figure 3.1). However, without the support of efficient indexing structure, such comparisons would be painful and burdensome, e.g. the pair-wise comparison brings the $O(N^2)$ time complexity. In Marlin, we build a distributed index supporting: 1) **Incremental Update** The index structure is supposed to support incremental update: for each new feature vector of the video arriving at the system, we return the similar videos, update the index and immediately serve the updated index for the future content-based queries. This strategy is more efficient than re-indexing the entire collection; 2) **Parallel/Distributed Processing** To support the processing in streaming scenario, we distribute the workload of index request processing to multi-cores and further multi-machines.

3.4.1 Distributed Inverted Index Structure

Our approach is built on top of inverted index structure proposed in [8]. When there is a newly uploaded video whose feature vector has been computed by the method we introduced in Section 3.3, we update the index by adding this new vector and at the same time presents as the query to the existing index. Figure 3.5 shows the basic structure of the inverted index for the binary vectors. Each entry in the inverted index table maps to one of the dimensions of the vector. The vectors which have the non-zero value at the dimension resides in the list associating with that entry.

A major challenge in the inverted index system shown in Figure 3.5 is that it does not support the concurrent access to the index table. Any concurrent read/write or write/write has to be serialized by exclusive locks.

To support the concurrent read/write or write/write, we first propose a lock-free design of the inverted index utilizing multi-cores architecture. The proposed inverted index structure is shown in Figure 3.6. We separate the entry lists in the inverted index into different shards according to the entry ID and assign each shard to a *ShardLord*. ShardLord is the only entity

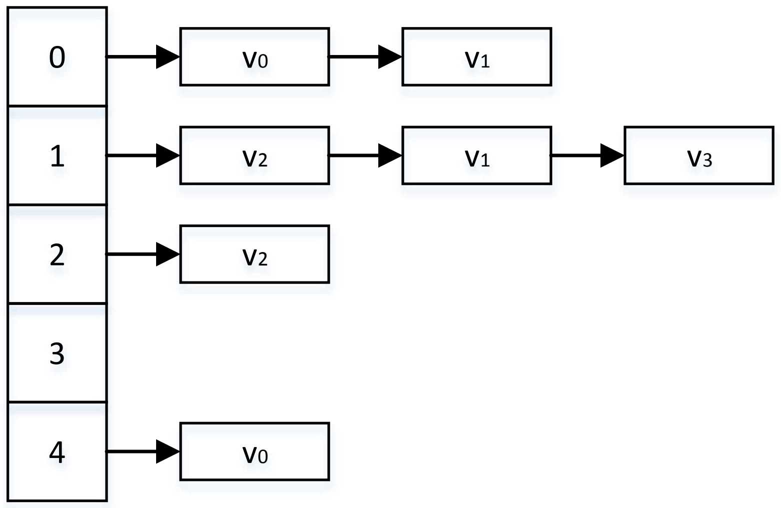


Figure 3.5 – Basic Inverted Index Structure: Each entry in the inverted index table maps to one dimension in the feature vector. The vectors which have non-zero value at the dimension resides in the list associates with that entry. $V_0 = (1, 0, 0, 0, 1)$, $V_1 = (1, 1, 0, 0, 0)$, $V_2 = (0, 1, 1, 0, 0)$, $V_3 = (0, 1, 0, 0, 0)$

with the access permission to that shard. The communication between the ShardLords and the other components in Marlin is via messages. Messages are queued in the message queue of the ShardLoad. To guarantee the thread safety, we guarantee that the message queue of a ShardLord can only be accessed by a single thread at any moment. This design follows the Actor model [2] if we abstract each ShardLoad as an actor. The advantage of the actor-based system is the easy support of high concurrency (abstract the system on high-level and minimize the context switch by sharing a thread among actors) and avoid the lock usage.

To further improve the capacity of the index system in a cluster of computers, we design the distributed scheme to spread the shards of the inverted index to multiple nodes in the cluster. We organize the nodes in the cluster in a peer-to-peer fashion and manage the membership of the nodes via Gossip protocol [90]. Each node in the cluster holds one or more ShardLords that maintain their shards. The access to the shards is the same as in

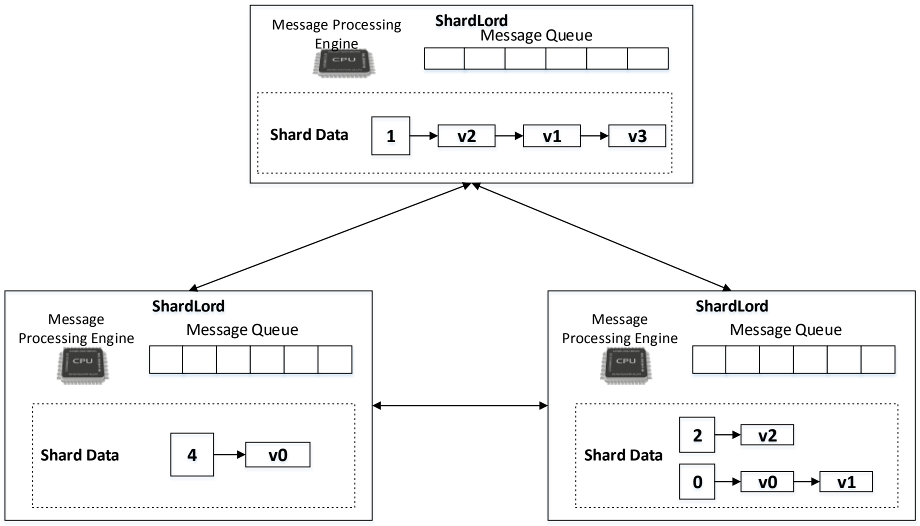


Figure 3.6 – Sharded Inverted Index: We separate the entry lists in the inverted index into different shards according to the entry ID and assign each shard with a ShardLord, which is the only entity with the access permission to the shard. The communication between the ShardLords and the external system and the ShardLoad is via messages. We guarantee that the message queue of a ShardLord can only be accessed by a single thread at any moment

multi-core scenario. Each node receives the newly generated video feature vectors from Streaming Feature Extractor and spreads it to the ShardLords across the cluster. The shard location across the cluster is maintained by an elected node, referred as coordinator. The other nodes get the shard location by querying the coordinator and save the location in the local cache to reduce the overhead of future queries.

3.4.2 Content-based Search

In this section, we discuss the content-based search process in our sharded inverted index system. We illustrate the details in Algorithm 1 and 2.

To assign the feature vectors to different shards, the Client Handler of each ShardLord (Figure 3.1) receives the feature vectors generated by Streaming Feature Extractor and performs Algorithm 1. We define n as the maximum number of shards in the system, d is the dimensionality of the feature vectors, and t is the similarity threshold selecting the similar videos. The basic idea of Algorithm 1 is that we traverse each non-zero dimension of the vector received from Streaming Feature Extractor and send the vector to one or more shards according to the dimensions (Line 7 - 9). In Line 5 of Algorithm 1, we introduce a v^{max}, which is a predefined vector and is consistent across all nodes in the cluster. Each dimension of v^{max} contains the largest possible value of that dimension across the cluster. For example, in the simplified scenario of Figure 3.5, v^{max} equals to (1, 1, 1, 1, 1), as all of the vectors in Figure 3.5 are binary vectors. $SCORE_v$ in Line 5 contains the accumulated similarity between the v and v^{max} one each dimension. The result of the algorithm is that, by indexing just enough number of feature dimensions, any vector v' whose similarity with v is potentially to be meet the similarity threshold can be identified as the response against the content-based search of v. We use this approach to mitigate the size of the inverted index as well as the cost to send vectors across the network. While a further optimization can be done by sorting the dimension according to the number of non-zero values as introduced in [8], it requires more data synchronization in distributed environment.

The node receives the V_j generated by Algorithm 1, and dispatches the V_j to the ShardLord that maintains Shard S_j. The ShardLord then executes Algorithm 2 which dynamically builds the inverted index and outputs the vectors which are similar to V_j. Algorithm 2 checks each dimension of v and find the vectors of the videos whose similarity is beyond the threshold.

Algorithm 1 Sharding Feature Vectors

```
1: procedure SHARDING($V$)    ▷ $V$ is the vector set received from Streaming Feature Executor
2:     $V_1, V_2, ..., V_n \leftarrow \emptyset$    ▷ the set of the normalized vectors sending to shard 1 to $n$
3:     for all $v \in V$ do
4:         for each non-zero dimension $d$ of $v$ do
5:             $SCORE_v \leftarrow SCORE_v + v_d * v_d^{max}$    ▷ $SCORE_v$ is the accumulated score of v, $v_d$ is the value at the dimension $d$ of $v$
6:             if $SCORE_v >= t$ then
7:                 $shardId \leftarrow mapToShard(d)$
8:                 $V_{shardId} \leftarrow V_{shardId} \cup (d, v)$
9:             end if
10:        end for
11:    end for
12:    for $j = 1$ to $n$ do
13:        Query location of Shard $S_j$ in shard location cache and if missed, query the Coordinator and saves the response in location cache
14:        Send $V_j$ to Shard $S_j$
15:    end for
16:    return $O$
17: end procedure
```

Algorithm 2 Find Matching Vectors in a Sharding to the input Vector

```
procedure FIND-MATCH($V, VS, t, I_p, I_{p+1}, ...I_q$)    ▷ the ShardLord maintains the inverted index entries with the IDs ranging from $I_p$ to $I_q$. $V$ is the vector set received from other nodes which contain the vectors to be indexed in this shard. $VS$ is the set containing all vectors in this shard
    $O \leftarrow \emptyset$    ▷ $O$ is the set of similar vectors' ID pairs
    for all $(d, v) \in V$ do
        $VS \leftarrow VS \cup v$
        for all $k \in I_d$ do    ▷ $k$ is the vector Id contained in $I_d$
            $v\prime \leftarrow VS_k$
            if $sim(v\prime, v) \geq t$ then
                $O \leftarrow O \cup \{(v, v\prime)\}$
            end if
        end for
        $I_d \leftarrow I_d \cup IDX_v$    ▷ $IDX_v$ is the index of vector v in $VS$
    end for
end procedure
```

3.5 Implementation

In this section, we discuss the implementation details of Marlin, with the specific focus on the resource-aware DStream (Section 3.5.1), Load Balancing in Online Content-based Feature Index (Section 3.5.2) and Fault-tolerance of the system (Section 3.5.3).

3.5.1 Feature Extraction with Resource-aware DStream

We implement the streaming feature extractor and resource-aware DStream abstraction by customizing Spark Streaming, the implementation of DStream abstraction [116]. Spark Streaming aggregates the data streams flowing into the system within the certain period as a Resilient Distributed Dataset [115]. We modify the logic to generate the RDD in Spark Streaming to implement the DRR-based resource allocation logic. RDD provides a partitioned distributed memory abstraction and for each partition, it starts a task for processing the data. The best part with RDD is that it records how every piece of data is generated, i.e. the "lineage" of the data, so that it handles data loss by re-run the computing tasks without the necessary to replicate data and to handle the accompanying data consistency issue.

3.5.2 Load Balancing in Distributed Feature Index

As stated, to be able to scale the system, we design the Feature Index in a distributed fashion. The unbalanced load makes some of the Feature Index servers having a long message queue where the messages are queued to be processed while others are idle. This situation lowers the overall system performance. We use Akka [107] to implement the sharded inverted index. We employ the facilities provided by Akka to implement the Load Balancing functionality.

Load Balancing in Feature Index We balance the load of the feature index processes by balance the number of the shards maintained by them. In our implementation, the cluster members elect the oldest member as the Coordinator, which maintains the shard location information across the cluster. Coordinator allocates the new shard to the node with the least number of the shards to achieve load balancing. To utilize the newly joined node in the cluster, Coordinator periodically checks the shard number on each node. When the gap between the maximum and minimum shard number across the cluster is larger than a user-defined threshold, Coordinator moves the shard from the overloaded nodes to the under-utilized nodes.

3.5.3 Fault Tolerance

We consider the three possible types of faults happening in Marlin:

— Data loss can happen for the failed processes, servers and the interaction with Data Feeder in Streaming Feature Extractor. For data loss in feature extraction job, we rely on the RDD lineage [115] to recompute data. To prevent the data loss due to the failure of Spark Streaming application, we enable the Write-Ahead-Log (WAL) in Spark Streaming to write all received data to a user-specified HDFS directory. The restarted Spark Streaming application can recover data from the directory before receiving new data.

— We also consider the message delivery reliability in the components implemented with Akka framework, i.e. ShardLords, Data Feeder. We use Akka's "At-least-once" mechanism to ensure that messages get delivered reliably.

— To avoid the data loss in Feature Index, the data maintained by the ShardLords are persisted to the shared storage system (e.g. HBase) and indexed by the ShardID. The restarted ShardLords recovers data, according to the ShardID from the shared storage system.

3.6 Performance Evaluation

This section describes the experimental evaluation of Marlin based on a real system implementation and the extensive simulation. Through the experiments, we are trying to answer the following questions:

— Whether the Streaming Feature Extractor scales with more computing resources, and whether the resource allocator in Streaming Feature Extractor improves the system performance (Section 3.6.1)?

— Whether the Feature Index responses the content-based search and index update request in real-time and scales with more computing resources (Section 3.6.2)?

— Whether Marlin can accurately capture the similarities among the videos to support content-based search (Section 3.6.3)?

We use the standardized dataset CC_WEB_VIDEO containing 24 categories of videos as workload. In each category, one of the videos is the seed video, and the others are with "Similar" or other marks based on the comparison with the seed. We feed the videos to the Data Feeder continuously with a test stub program to simulate the streaming video files.

All servers in the cluster used in the evaluation are with Intel(R) Core(TM) i3-3220 CPU @ 3.30GHz (4 cores with Hyper-Threading running at 1.6 GHz) and 16GB RAM. The servers are linked with 1Gbps Ethernet. Unless mentioned otherwise, the shard number is 300, the threshold filtering similar video pairs as 0.85 in the experiments (t in Algorithm 1), and the batching interval of the Spark Streaming as 1s. We use the standalone deployment model of Spark 1.2.0 and allocate 4GB memory to each executor. We will specify other parameter values in the following subsections.

3.6.1 Streaming Feature Extraction

We test the scalability of Streaming Feature Extractor in this section. We compose the workload with 250 videos which are randomly chosen from CC_WEB_VIDEO dataset and feed them to the system in the Poisson distribution (with the mean value of 30). We use the Apache Commons Mathematics Library [4] to generate the Poisson-distributed workload and all videos are sent within 50 seconds.

Figure 3.7 shows the throughput speedup of the system with different core number. Because we have to allocate one of the cores to receive data from Data Feeder, we take the core number of 2 as the baseline (1 out of 2 cores is used for processing the feature extraction task) and set the corresponding speedup as 1. We observe that Marlin achieves 25X speedup with 16 cores comparing against the sequential algorithm. We measure that 16 cores setup of Marlin takes 52 seconds to complete the feature extractions for all videos. The performance improvement comes from two factors: 1) The more processing cores bring more computational power; 2) the distributed architecture amortizes the memory overhead with more machines so that it reduces the system performance degradation (e.g. avoid Garbage Collection).

We also test the effectiveness of DRR-based resource-aware DStream abstraction in Marlin. Due to the limited size of the physical cluster in hand, even the videos with the least numbers of keyframes occupy the considerable portion of the cluster computing resources, so we decide to use the simulation to evaluate resource-aware DStream. We simulate a 100-

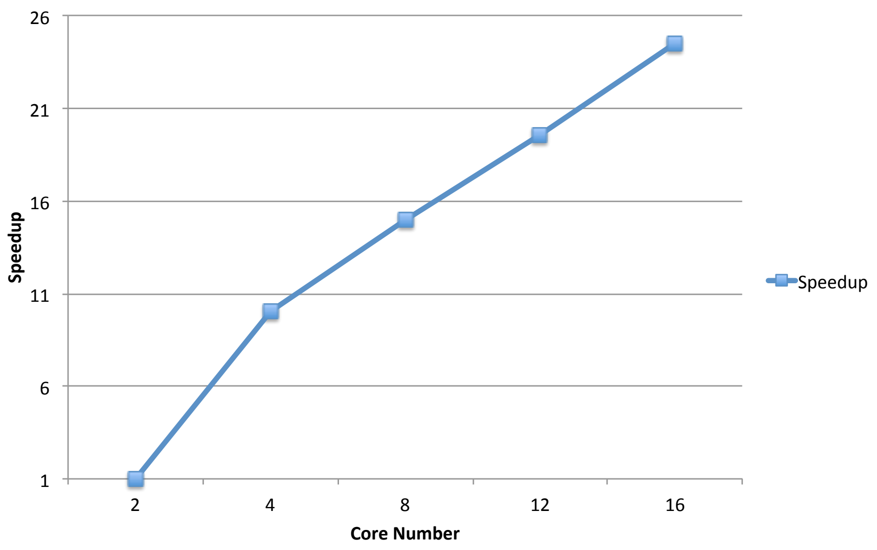


Figure 3.7 – Scalabilty of Marlin: With 250 videos arriving with Poisson distribution, Marlin achieves 25X speedup with 16 cores and it takes around 52 seconds to extract features for all 250 videos.

cores cluster and serve it with batches containing 50 videos. The size of the videos is sampled based on the size distribution of CC_WEB_VIDEO dataset (see Figure 3.2). We set the time cost to generate the feature vector for each keyframe as around 50ms. Regarding the DRR scheduler configuration, we make quantum as 1 and DRR scheduler checks the scheduling candidates every 1 ms. We demonstrate the experimental results in Figure 3.8. We observe that with the DRR-based resource allocator, the response time of the videos is proportional to their computing resources demand, i.e. key frame number. Additionally, since we give the higher priority of processing to the small videos, the response time of the large videos also improves for nearly 5X due to the disappearing of the computing resource contention with the small videos.

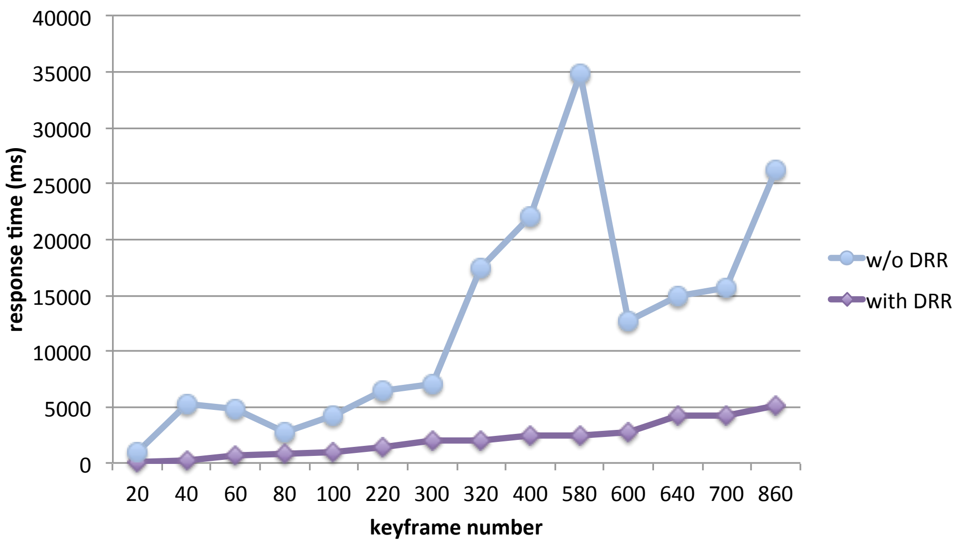


Figure 3.8 – Response Time with/without Resource-aware DStream Abstraction: The figure is generated by batches consisting of 50 arrival videos. We set quantum as 1 and DRR-based scheduler checks the scheduling candidates for every 1ms.

3.6.2 Distributed Feature Index

To test the performance of Distributed Feature Index, we build a customized benchmark by replicating the videos in the 9th category of CC_WEB_VIDEO. In the experiment, we feed the system with 500 requests per second. Figure 3.9 shows the scalability of the feature index when we increase the size of the cluster from 1 to 16 cores. When we increase core number from 1 to 4 cores, we found that the speedup increases only for 2x. The reason is that these four cores are on the same machine, so both one and four cores setup suffer from the fact that a large number of in-memory objects occupy the heap space of the process, and Garbage Collection (GC) happens more frequently than multi-nodes setup. When we scale the system to multiple machines, the speedup grows quickly. With 16 cores (4 cores per machine), we can increase the speedup to as much as 24x.

The second experiment we conduct to test the feature index performance is to measure

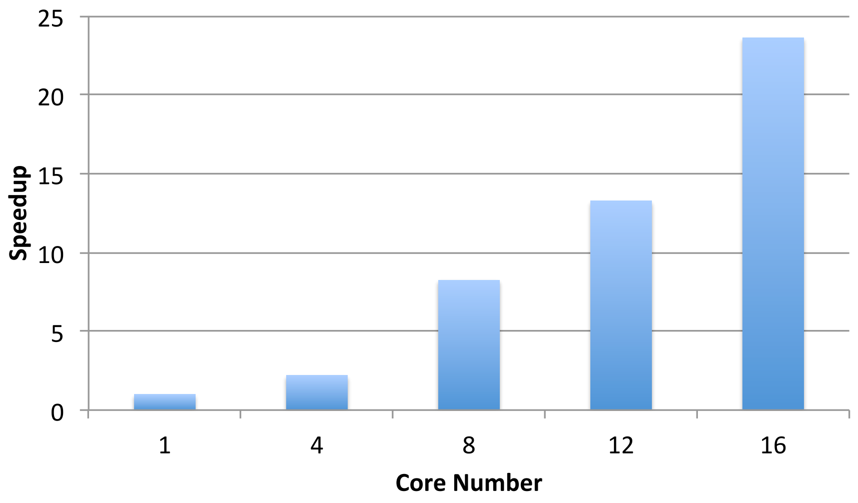


Figure 3.9 – Scalability of Feature Index: with 16-cores setup, the speedup is 24x. The single-node setup (one core and four cores) suffers from the frequent happening of Garbage Collection

the processing latency under the highly-concurrent requests. We assign the feature index with 16 cores (4 cores per server) and feed the system with the various number of requests as a single batch and measure the average processing latency. We demonstrate results in Figure 3.10. We observe that with the increasing number of requests in the batch, the average latency increases accordingly since the servers have the limited computing resources and the processing of the requests competes for the resources with each other. However, the feature index is still able to consume the update requests in real-time. With 500 requests in the batch, it takes 34 ms on average to update the index with the latest video.

3.6.3 Accuracy of Marlin

Figure 3.11 shows Marlin's accuracy with different threshold deciding whether two videos are similar. We use 9th category in CC_WEB_VIDEO in this experiment. We can achieve 100% precision and meanwhile maintaining the recall as high as around 80%. This result

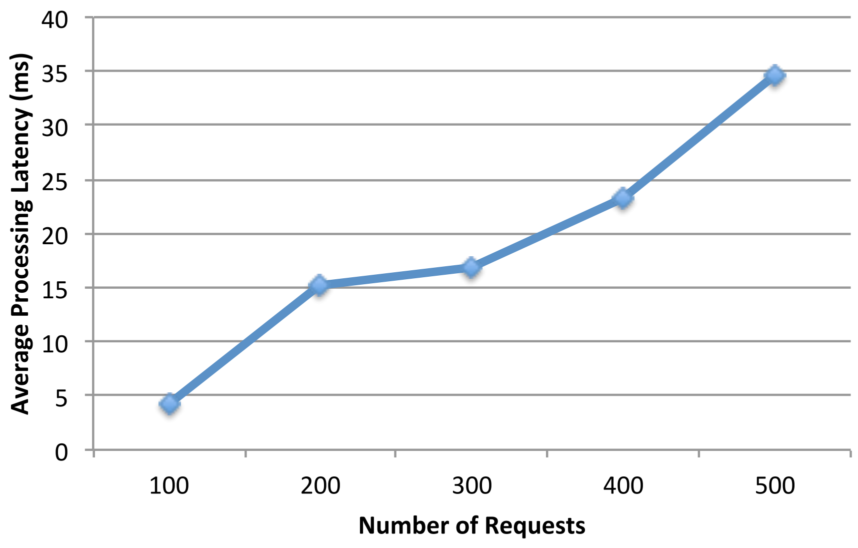


Figure 3.10 – Processing Latency Under the Workload With Different Size

is better than the reported performance of color distribution algorithm in [94], in which the recall drops to 0 when the precision is 100%. The reason for the much better accuracy of Marlin is that we describe the color distribution of the videos with higher dimensional vectors. Though the higher dimensionality of the vectors brings more computation overhead, the distributed and parallel architecture of Marlin mitigates that and keeps the satisfying computing performance. Therefore, we have much larger tunable space to achieve better accuracy. In the experiment, we describe the vector feature with a 512-dimensional vector, which is concatenated with 256 bins for Hue, 128 bins for Saturation and 128 bins for Value.

3.7 Related Work

Content-based Video Search The proposed approaches in [118, 94, 98] are various ways to detect the similarity of the videos. The authors of [94] leverages relative gray-level intensity distribution within a frame and temporal structure of videos along the time line

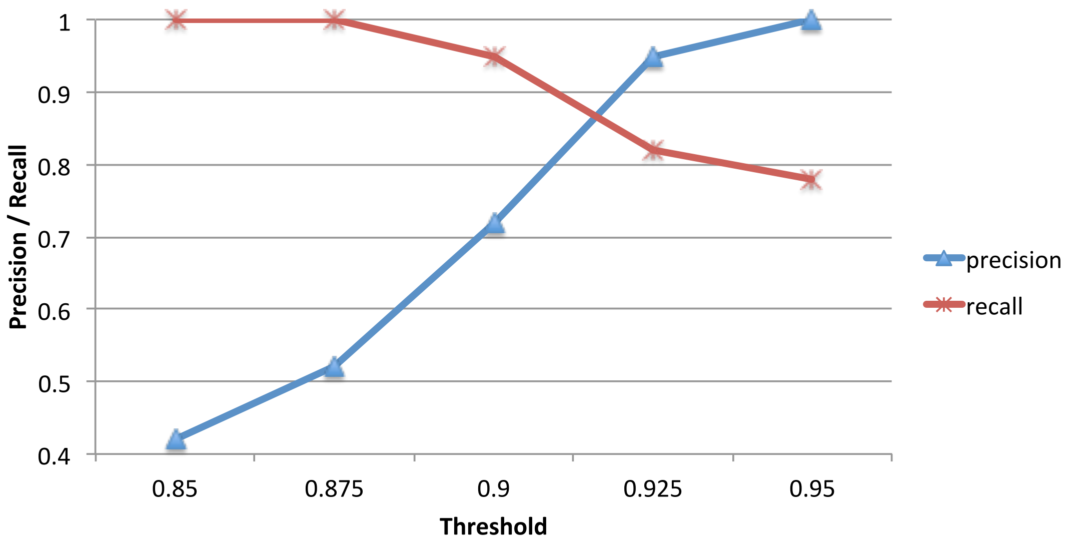


Figure 3.11 – Accuracy of Marlin with different threshold: We can achieve 100% precision and meanwhile maintaining the recall as high as around 80%. We describe the vector feature with a 512-dimensional vector, which is concatenated with 256 bins for Hue, 128 bins for Saturation and 128 bins for Value.

to represent the feature of the video. This approach requires that the frames of the same video are processed in the same thread to extract the temporal information, i.e. it is hard to parallelize efficiently with the state-of-the-art parallel computing framework. The proposed system in [98] employs multiple features of the video and learn multiple hash functions to map the video frames into the Hamming space. The high complexity of this system prevents it from being used in the online scenario, like Marlin. Besides that, both of these two approaches requires the pre-calculated feature of the videos and assumes the static feature index.

Similarity Indexing During the past years, many methods were proposed to index data in the feature space. Spatial indexing approaches like *kd-tree* [9] were proposed to handle low-dimensional data. But this category of approaches suffers from the exploded search space when data is with high dimensions. Though we can use the index like *LSH* [32, 103] to reduce the search space. Additionally, conventional LSH scheme does not scale to the modern computer architecture [103] and it does not support online update efficiently (We will have a detailed discussion in the next chapter). We implement the distributed inverted index which dynamically update the index structure and answer the similarity search queries. PLSH [103] also proposed to use distributed design to accelerate the similarity search process, it has to broadcast the request to all servers due to the lack of indexing structure. Marlin forwards the similarity search query to the exact server serving the similar candidate by indexing the vectors with their significant dimensions and provides data persistence functionality for the fault-tolerance purpose.

3.8 Summary of this Chapter

In this chapter, we have developed Marlin, an online data analytic application which indexes videos based on the visual features of the videos. Our key idea is to integrate both the feature extraction and content-based index as an online processing pipeline. Specifically, we develope a micro-batch based streaming feature extractor and a distributed feature index supporting incremental update. Additionally, we develope the fair queuing theory based resource allocator to achieve load balancing and ensure the fluency of the online data processing pipeline. The extensive experimental and the simulation-based evaluations demonstrate the superiority of our system over the existing work.

The research work presented in this chapter is among the first to combine the feature extraction and the content-based video index as an online data analytic pipeline. The evaluation results show that our design is promising in developing the future online video sharing and hosting services. Marlin serves as an successful example to realise online data analytics in practice. On a higher level, the design philosophy of Marlin brings great potential to benefit the existing online data analytic system design in other fields, include the measurements from sensor applications, log records, blog or twitter posts and web page clicks, etc. Ingesting these types of data in online and pipelines with the data analytic phase is computational-intensive, contains various application-specific requirements on the resource allocation, and demand the efficient computing state management strategy to perform incremental computation. Marlin sufficiently provides a reference to the design of these applications and prove the feasibility of the design pattern of online data analytics.

Chapter 4

Computing State Management in Online Data Analytics

In the previous chapter, we focus on how to make the input data be consumed in online with high throughput, and we use an inverted-index-based data structure to serve the computing state query/update. In this chapter, we have a further discussion about the design of an efficient computing state management layer, with the focus on the case that the computing state entries have complex structure.

We use the very high-dimensional computing state as an example. We propose a system based on Locality Sensitive Hashing (LSH), which is one of the most promising tools to handle high-dimensional data but does not fit in the online data analytic applications due to its conventional design.

In this chapter, we achieve online data analytics by answering the question, "how to adapt LSH to handle the high-dimensional and unstructured computing state for online data analytics?". Specifically, we put our effort on building an LSH-based state management layer with several system-oriented improvements, including the hierarchical memory system, reconstruction-free data structure, and parallel-friendly data partitioning. With the improvements, we can scale to large state size, afford the online update and be friendly to the parallel processing.

This chapter is organized as follows: Section 4.1 covers the motivation and background knowledge. Section 4.2 goes into the details of the hierarchical memory design. Section 4.3 describes how we use the adaptive hash tree to handle online update requests. In Section 4.4, we discuss the parallel-friendly indexing structure and the concurrency management module in our system. Section 4.5 contains the extensive experimental evaluation. Section 4.6 discusses the related work. Section 4.7 summarizes the chapter and discuss the difference between PFO and the general online database system.

4.1 Background

The recent exponential growth of social media, multimedia, and artificial intelligence applications have produced many instances where high-dimensional feature vectors are used to represent data. Accordingly, the online data analytic applications/algorithms working in these scenarios usually have to deal with the computing state with the high dimensionality.

One of the most prevalent data analytic algorithms handling high-dimensional data is Nearest Neighbors (NN) search. Given a dataset $D \subset \mathbb{R}^d$ and a query data point q, Nearest Neighbor (NN) search problem aims to find a data point o in D, and requires any other data points $p \in D$, $\|\mathbf{p}, \mathbf{q}\| \geq \|\mathbf{o}, \mathbf{q}\|$, where $\|\|$ represents the distance between two points. To improve the efficiency of NN search in a high-dimensional space, researchers developed approximate version of NN search algorithms, i.e. Approximate Nearest Neighbor (ANN), to locate $o^* \in D$, where $\|\mathbf{o}^*, \mathbf{p}\| \leq c\|\mathbf{o}, \mathbf{p}\|$, c is the approximate ratio and usually larger than 1. NN search is the fundamental technique supporting various applications, e.g. recommendation system [91], machine learning [109], etc. However, the computational overhead and the storage pressure increases with the growth of dimensionality dramatically, a.k.a. "Curse of Dimensionality" [10, 58, 110], which makes it super-challenging to achieve online NN search.

The state-of-the-art online NN search systems sacrifice the system effectiveness for gaining the responsiveness. We use the online recommendation system [91] as an example to illustrate the scenario. Online recommendation system aims to capture the real-time user behavior and produce the recommending suggestions interactively. Therefore, it has to keep the users' behavior as computing state and adjust the recommendations against the frequent updates to it. In a large-scale system with the tremendous number of users/items, the user behavior has to be represented as the high-dimensional vector, each dimension of which stands for whether this user clicks the corresponding item. The high-dimensional vector not only brings the storage pressure but also makes the computation over them challenging. For example, the pair-wise comparison between the vectors becomes prohibitively expensive to calculate the similarities for the recommendation. The existing systems usually remove all computing state which locates outside of a time window [52, 16]. The consequence is the missing of the opportunity to make more comprehensive recommendations.

4.1.1 Locality Sensitive Hashing

Instead of removing all state entries out of the time window in the online NN search systems, we propose to adopt the approximation technique to handle the high dimensional state entries in the computing state management layer. The key idea is to use Locality Sensitive Hashing (LSH) [27, 53] to index the state entries in the system. LSH is proposed as one of the most promising tools to handle high dimensional data. It has been widely adopted in the production environment, e.g. Google News [25] (in the offline scenario). The basic idea of LSH is to reduce the dimensionality of the data points by mapping them into a lower-dimensional space with the specially designed distance-preserving hash functions. Consequently, the hash values of the data points are the same or similar with high probability if they are close to each other in the high-dimensional space.

Locality Sensitive Hashing (LSH) is formally defined as following:

Definition 1 (Locality Sensitive Hashing) Given a distance R, an approximate ratio c as well as two probabilities p_1 and p_2, a hash function $h : R^d \to \mathbb{Z}$ is called (R, c, p_1, p_2)-sensitive, if

— If $\|\mathbf{p}, \mathbf{q}\| \leq R$, then $Pr[h(p) = h(q)] \geq p_1$;

— If $\|\mathbf{p}, \mathbf{q}\| \geq cR$, then $Pr[h(p) = h(q)] < p_2$;

We guarantee that $c > 1$ and $p_1 \geq p_2$. In practice, we use a *Compound Hash Key*

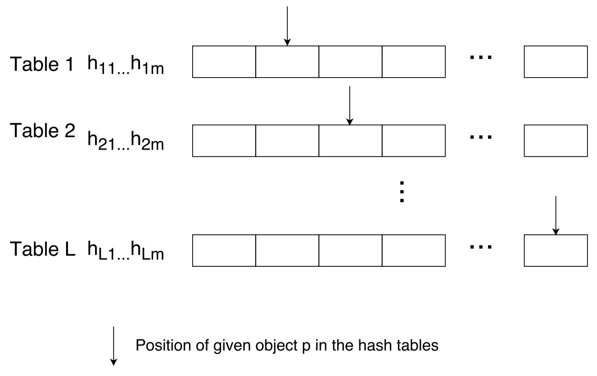


Figure 4.1 – Locality Sensitive Hashing.

to map the object into a bucket in an LSH hash table. Given a compound hash function $G = (h_1, ..., h_m)$, the bucket ID consists of $(h_1(p), ...h_m(p))$. In this chapter, we adopt the angular distance of two vectors. We use the hash function which is parameterized by an unit vector a, and the hash value of the query object q is $h_a(q) = sign(a.q)$.

Within a single hash table in LSH, the conventional approaches only take the data points with the same hash value, i.e. in the same hash bucket, as the candidates of the nearest neighbor. This rule is too strict as it filters out too many data points. To mitigate this, we usually use L hash tables in an LSH-based system. We illustrate the structure of LSH in Figure 4.1. Multiple hash tables in LSH cause significant memory overhead to the LSH-based system. To reduce the number of the required hash tables in LSH, one approach is to take the data points with the "similar" hash values as the candidates of the nearest neighbor [70, 67, 104]. To measure the distance between two compound hash values, we adopt the similar definition as in [104]:

Definition 2. (Distance of Compound Hash Values). Given two compound hash keys K_1 and K_2, the distance between them is denoted as $dist(K_1, K_2)$ and is defined as: $dist(K_1, K_2) = \frac{1}{LLCP(K_1,K_2)}$, where LLCP is the longest length of the common prefix of two compound hash values.

4.1.2 Online vs. Offline LSH

The most challenge to embrace LSH in the online data analytic applications comes from the gap between the existing LSH design and the requirements of the state management layer in online data analytic applications. Comparing to the requirements stated in Section 1.1, we identified several facts that hinder the application of LSH in online data analytics:

— **Storage** A challenge we face is the dilemma that we should whether to pursue data access efficiency or storage scalability? For the seek of the efficiency, we prefer to use RAM to implement LSH; but for scalability, usually people adopt disk storage. Achieving both efficiency and scalability is desired for big data applications with online performance requirements, but the state-of-the-art LSH design [103, 70, 67, 39] cannot achieve both goals.

— **Parallelization** The parallel processing has been adopted to improve the data analytic applications' performance in most of the scenarios [100, 55, 47, 78]. Therefore, the query/update requests to the computing state arrive at the state storage layer concurrently. Most of the current LSH designs aim to optimize the efficiency of a single query request [104, 39, 67] but ignore the support to parallel processing. The others propose the parallel/distributed LSH system. A few of them execute the queries in batch [7, 48], hence only work in the offline scenario. The others require broadcasting requests to all servers due to the lack of cluster-wise indexing. We are in need of indexing the high-dimensional state entries with LSH but with better scalability against the concurrent query/update requests.

— **Online Update** The online data analytic application requires the state management

	Storage Medium	**Parallel Data Access**	**Online Update**
Multi-Probe LSH [70]	RAM	Single Thread	No
LSB-Tree [104]	Disk	Single Thread	No
C2LSH [39]	Disk	Single Thread	No
SK-LSH [67]	Disk	Single Thread	No
PLSH [103]	RAM	Distributed	Pause the processing to consume it
Distributed LSH [7]	Disk	MapReduce	No
Hypercurves [105]	RAM	CPU&GPU for Construction	No
GPU-LSH [84]	RAM	GPU for Construction	No
PFO	Hierarchical Memory System	Multi-Threaded	Parallel-friendly Index and Smart Write Task Dispatching

Table 4.1 – Analysis of Representative LSH systems and How PFO differentiates itself with them

layer to offer the capability to query/update the state entries in fine-granularity. The contemporary LSH systems [84, 105] are designed with the assumption of no or infrequent update requests. Therefore, they have to be reconstructed to consume the online updates or pay additional space and/or accuracy cost to maintain the updated version of the state entries [67, 103]. It is not practical to use LSH in contemporary big data systems if LSH fails to support the online update efficiently.

We summarize the representative LSH-relevant research work in Table 4.1.

4.1.3 LSH-Based State Management Layer

In this chapter we design ***PFO***, A **P**arallel-**F**riendly State Management Layer for **O**nline data analytic applications. We utilize multiple system-oriented optimization strategies to serve massive state size, concurrent and online requests. Specifically, we made the following contributions:

— **Hierarchical Memory** To resolve the dilemma between the query efficiency and system scalability, we adopt a hierarchical memory system in PFO. All query and update requests to PFO are firstly consumed in RAM memory space to improve the processing speed. In RAM space, we employ a data placement strategy to reduce the overhead brought by Garbage Collection. To scale the system capacity, we write data to flash memory in the case of overloaded RAM and organize data in a read-friendly format to improve the query efficiency.

— **Reconstruction-Free Hash Tree** We build a hash tree structure which accepts the online update requests without the necessary to reconstruct itself. With the hash tree, PFO accommodates the online update requests more efficiently than the other data structures which require reshaping the index, e.g. B-Tree [104].

— **Parallel-Friendly Design** We propose an indexing structure called Partitioned Hashing Forest (PHF) to facilitate the parallel data access. By applying the LSH functions to the data points and then to their hash values, we divide the memory space into two levels, called *HashTree* and *Partition* level. With PHF, the state entries in PFO is partitioned with the respect to their locations in the high-dimensional space. Each partition can be accessed in parallel without the crossing dependence. Additionally, we have a concurrency management strategy to avoid the synchronization across threads. By eliminating the cross-threads synchronization, we significantly improve the performance when the query and update requests arrive at the system concurrently.

We implement PFO in a prototype and evaluate it on 10-cores server with the standard benchmark datasets. We compared the performance of PFO with its variations to show the effectiveness of its components. Our results show that PFO: 1) scales with the growing size of the state entries in the hierarchical memory system; 2) scales processing throughput close to linearly with the increasing number of cores; 3) achieve the higher accuracy with less resource consumption. We list the notations used in the rest of the chapter in Table 4.2.

Notations	**Explanation**
L	the number of hash tables used in LSH schema
C	the number of LSH hashing functions which are used to locate the data point in partition level
S_{ij}	the snapshot of partition i of a hash table locating in flash memory which is saved at moment j
m	the number of bits in a data point's hashing value which is used to locate the hashing tree in H layer
t	the allowed number of data points residing in the same bucket (except the last level of the hash tree)
l	the number of slots in the directory node
s	the size of the data point in terms of bytes
r	the record ID of the data point
h	hash value of the data point
$A(q)$	nearest neighbors candidates of the query data point, the data points in this set whose distance to the query data point is no larger than the user-defined threshold, R, are selected as nearest neighbors
o	offset of the leaf in off-heap space
M	maximum length of hash key in PFO in terms of bits
h'	variation of h by discarding h's' last $i * log_2(l)$ bits, where i is an integer ranging from 0 to the height of the hash tree
h'_{max}	the maximum length of the existing h' in hash table in terms of bits
KL	the longest length common prefix of a data point's hash key and the other one

Table 4.2 – Notations of PFO

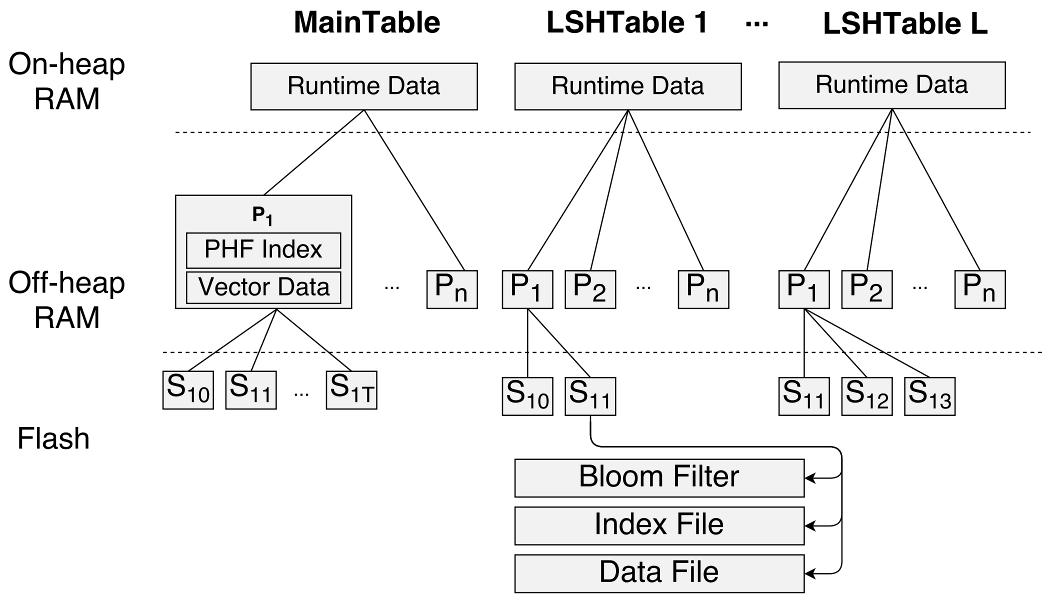


Figure 4.2 – Storage Memory Layer Design: PFO saves data in both RAM and Flash Memory (SSD-based). It consumes all I/O in RAM to improve the performance and saves snapshots (S_{ij}) in SSD for increasing the system capacity with the performance guarantee. Additionally, the off-heap space for each table is divided into multiple partitions (P_i) for exploit parallelism for processing.

4.2 Storage Design of PFO

In this section, we present PFO's storage design which consists of multiple hash tables locating across hierarchical memory space. This design resolves the dilemma between the efficiency and capacity of LSH systems. Figure 4.2 illustrates our design. In the following subsections, we step through the components in the figure. We begin the description with how we organize state entries in multiple data tables in Section 4.2.1. After that, we move forward to the design of each layer in the hierarchical memory system of PFO in Section 4.2.2.

4.2.1 Hash Tables in PFO

LSH systems usually employ multiple LSH tables to achieve better accuracy for nearest neighbors search. However, the state-of-the-art systems have focused on the optimization of individual LSH table, but ignore the overall performance and overhead of the systems. For example, in LSH systems described in [104, 67], the vectors have multiple copies, each of which is saved in an LSH table. This storage approach incurs extra storage overhead. Even worse, in LSH systems like SKLSH [67], the order of the vectors in an LSH table affects the system accuracy. In online scenarios where the vectors are subject to the frequent updates, the order of the vectors may need to be changed accordingly, and additional efforts are required to maintain the index of the vectors in multiple LSH tables to optimize the query response of the LSH systems.

To overcome the issues in the existing LSH design, we adopt a "MainTable + LSHTables" structure in PFO. As shown in the top line of Figure 4.2, PFO consists of one MainTable and L LSHTables. The MainTable stores the state entries (represented as vectors), which are indexed by the unique IDs. The LSHTables only store the IDs of the vectors. Therefore, PFO saves the storage space by keeping a single copy of each vector in MainTable and achieves better accuracy with multiple LSHTables which only store the vector IDs.

PFO handles query and update requests as follows. When fetching nearest neighbors for the given state entry q, we calculate q's hash values with the hash functions associated with L LSHTables and fetch the IDs of the nearest neighbors candidates, referred as $A(q)$, from all these tables. We then remove the duplicate IDs and fetch the corresponding vectors from MainTable with the remaining ones. When adding a new vector to PFO, we first save it in MainTable and then update L LSHTables with its ID according to its hash values in these tables. MainTable is built with the hash function minimizing the hash conflict

(MurmurHash3 [45]) providing an O(1) complexity when reading/inserting/deleting. Since MainTable does not impose any constraint on how the vectors shall be placed in the memory space, we do not have to rebuild the whole hash table for vector updates.

4.2.2 Hierarchical Memory System

As we discussed in Section 4.1, the online environment brings the challenges on both system capacity and responsiveness of PFO. Therefore, neither of pure-RAM approaches like E2LSH [27] and Multi-probe LSH [70], or pure-disk approaches like SK-LSH [67] are ideal in the case. In this section, we introduce how we overcome the challenges with Hierarchical Memory System in PFO. The left part of Figure 4.2 suggests that each table in PFO locates across three types of memory: *on-heap*, *off-heap* and *flash* memory. Among the three components, on-heap and off-heap are in RAM and flash memory is built with Solid State Disk (SSD).

RAM

To respond quickly to the query/update requests and offer scalable storage capacity as well, we employ both RAM and flash memory in PFO. RAM serves as the buffer for query and update operations. When handling a query request, we first search the $A(q)$ in RAM before we go to the disk space. When we update a data point, it is added in the RAM. If the RAM is overloaded, the data points will be dumped to flash memory.

We divide the RAM space in PFO into on-heap and off-heap to reduce the overhead applied by the program runtime. Recently, more and more database systems serving a large volume of online user requests are developed in programming languages that rely on automatic Garbage Collection (GC) [102] to reclaim memory space. The representative examples include HBase [5] and Cassandra [3] serving Facebook and Apple respectively. However, when the program is under significant memory pressure and getting free memory space is tough, GC will bring considerable overhead to the user applications [72, 41]. In the worst case, GC threads pause the application threads to get free memory space with the best efforts, a.k.a. Stop-The-World GC. To minimize the negative effect brought by GC, we save different types of data in on-heap and off-heap spaces.

In **on-heap** part, we only keep the runtime data, which includes system runtime parameters and the handler of the off-heap data. The runtime data is small and brings ignorable overhead to the program memory. We save the PFO tables in **off-heap** space. Each table is divided into independent partitions (referred as P_i in Figure 4.2) to facilitate the parallel access to the whole table. Every partition contains the content incurring most of memory cost, ***Data*** and ***Index***. For MainTable, Data is the vectors representing the data points; while for LSHTables, Data are the IDs of the vectors. To locate the IDs (in LSHTable) and vectors (in MainTable) quickly, we employ a Partitioned Hash Tree (PHF) Index to serve the purpose. As the name suggests, PHF is a set of hash trees, the non-leaf nodes of which are kept in the off-heap memory as Index. (We will leave the detailed introductions to the partitioning algorithm and the PHF index to Section 4.4.1). By saving MainTable and LSHTable in off-heap memory that is not subject to GC, we minimize the chances to trigger GC, therefore, achieve the high system performance of PFO.

We describe the memory layout in a single partition of the PFO tables in Figure 4.3. By saving the roots of the hash trees in the fixed offset, we start searching $A(q)$, the nearest neighbors of vector q, in the hash tree by fetching the root node, step down the levels and eventually locate the leaf node, i.e. the vectors or vector IDs. As a result, we may perform read operations for multiple times for ***Index*** segment and for each time, we get the offset indicating the next memory block to read. To further reduce the memory cost in MainTable, we save the vector in a compressed format in ***Data*** segment. We design the format based on the fact that the vectors are mostly sparse in many scenarios, e.g. recommendation system [52], machine learning [88]. A vector consists of 3 fields, size, non-zero indices and non-zero values. "size" is the dimensionality of the vector, "non-zero indices" records the dimensions where the vector has a non-zero value and "non-zero values" are the non-zero values in the dimensions referred by non-zero indices field. To represent the vectors with the same LSH values, we have the offset field in each instance of the vector indicating the offset of the next data point with the same LSH value.

A major challenge for off-heap memory space design comes from the frequently updated vector. As a result, we need to have a fast approach to invalidate the memory space used by the old vectors, and reclaim it for future use. Therefore, we employ ***Header*** segment containing the metadata describing the whole partition. We allocate memory in times of 16 bytes and address the memory space with the eight-bytes-length address. When the vector is updated, we save a new version of the vector in Data segment of the memory space and update the corresponding bytes in Index. After that, we reclaim the space that is used to save the old version. We reclaim the memory space through a set of LinkedLists in the Header segment of the off-heap memory. The offset of the first LinkedList is $RECLAIMED\_LIST$. Given a vector with the size of s bytes, when its space is reclaimed, we add the offset address of the memory block to the LinkedList with the offset $RECLAIMED\_LIST + (s - 16)/2$. When we allocate the memory space for a new vector with the size of s bytes, we first check if there are reusable space in the LinkedList offsetting at $RECLAIMED\_LIST + (s - 16)/2$.

In practice, we have a limited number of LinkedLists, thus, have the maximum size of each allocated memory block. We chain the memory blocks in order to support the vector

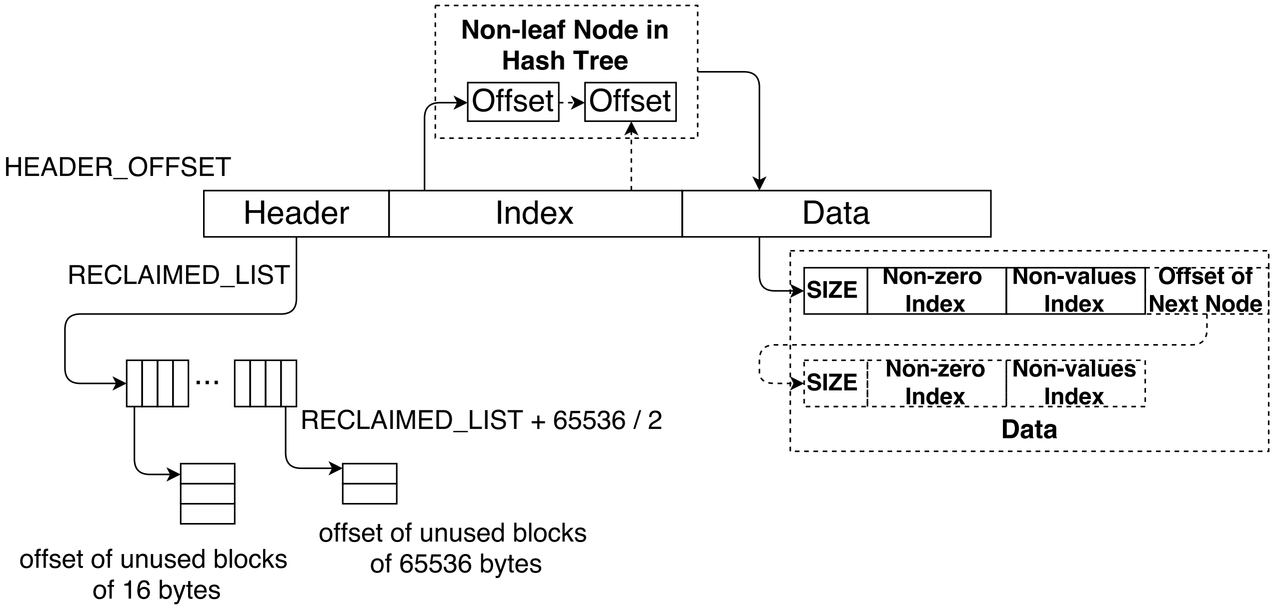


Figure 4.3 – Memory Layout in Off-heap space. In the figure, we only show the layout of MainTable which has vectors in Data segment for simplicity. For LSHTables, the content in Data segment is the IDs of the vectors.

whose size is larger than the maximum memory block size.

Flash Memory

Given the limited space in off-heap memory, we employ flash memory to accommodate a high volume of data in the online environments. PFO allows the user to set a threshold of the data size in off-heap space. When the data size is beyond the threshold, we make a read-only snapshot of the data in off-heap space, and save snapshot in the flash memory (referred as S_{ij} in Figure 4.2, where i is the table's partition i in off-heap space, j stands for the timestamp when creating this snapshot). Figure 4.2 shows that the individual snapshot in flash memory contains *Index* and *Data* files, corresponding to Indexing and Data segments in Figure 4.3.

In LSH system, different LSHTables calculates the LSH values of the state entries with the different set of hash functions. They may give different answers to whether two vectors are in $A(q)$ of each other. As a result, it is impossible for us to arrange two vectors in MainTable in the sequential disk space for more efficient disk operations if their hash values are equal. In another word, the access pattern in MainTable has to be random. We utilize flash memory to serve **query request** with the salient performance. According to the previous studies, there is no evident gap between the random and sequential reading performance in SSD-based flash memory [31]. By utilizing the flash memory, we do not need to pay extra overhead even vectors in MainTable are put in random. Regarding the **update request**, flash memory does not provide the random write performance that is as good as its read counterpart so that we need to avoid the random write. The snapshot we keep in flash memory layer is read-only, i.e. when off-heap partition reaches its threshold, we create the snapshot, and the future updates on the vector data will be consumed by the RAM-based buffer first and reflected in the future snapshots. Since we write only once for every snapshot, it is easy to perform the sequential write.

With the time passing, each partition may have multiple snapshots in flash memory. As a result, searching a particular data point requires traversing multiple snapshot files, which brings large query overhead. To deal with multiple snapshots, we generate a summary for each snapshot with Bloom Filter [12] (shown in Figure 4.2). The Bloom filter is a compact signature built on the hash keys in each snapshot. The key in the Bloom Filter of the MainTable is the vector ID. In the LSHTables, we use the indices of all non-empty buckets as the keys of Bloom Filters. To search $A(q)$ in flash memory, we go through the snapshot files in the reversed time order. For each snapshot, we test the Bloom filters instead of searching snapshot file directly. If any Bloom filter matches, the corresponding vectors are retrieved from flash. Bloom Filter based lookups may result in false positive; thus, a match could lead to an unnecessary flash I/O. According to studies in [35], we are able to control the false positive rate in low level with a very small disk space cost, e.g. with 13 hash functions and 2.29MB disk cost, we can build a Bloom Filter for 1M items with the false positive rate which is as low as 0.0001.

4.3 Online State Query/Update

To achieve online data analytics, we need to provide the index structure which not only maps the hash keys of the state entries point to their locations in the storage system but also apply the minimum change to the indexing structure when handling update requests. Operations altering the shape of the indexing structure consumes many CPU cycles and blocks the access to the affected region of the index until the reshaping is finished. For example, PLSH [103] requires to maintain the updated vectors in another hash table and stop the processing to merge hash tables periodically. In this section, we propose to use a hash tree structure which supports the in-place update when new data points are added and keep the query effectiveness by adjusting the resolution to identify the nearest neighbors.

4.3.1 Adaptive Hash Tree

Figure 4.4 illustrates the hash tree structure. The hash tree consists of two types of nodes, leaf node and non-leaf node. The non-leaf node is an integer array with the length l. Each slot in the array maps to a bucket in the hash table. The value in the slot is the offset of the first leaf node in the slot or the offset of the non-leaf node in the next level of the tree. A leaf node consists of three fields: KEY field saves the vector ID, $VALUE$ field

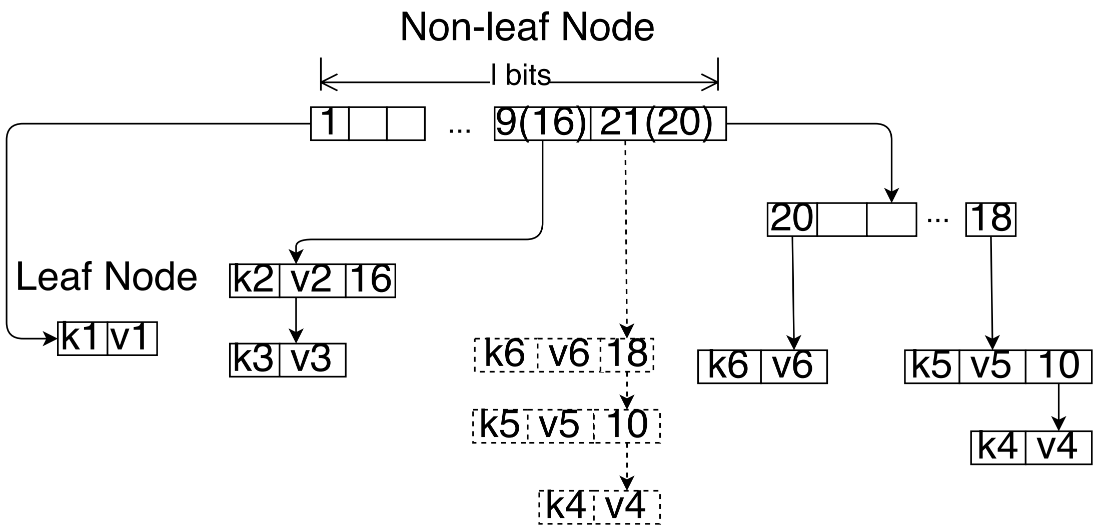


Figure 4.4 – Adaptive Hash Tree Structure: X(Y) in the Figure stands for that the original offset Y is replaced with X. In the hash tree, we progressively include more bits in the hash value of the vectors to locate them in the buckets. When there are more than t nodes under the same buckets we redistribute them to the next level of the hash tree. The leaf nodes containing only two fields means that it is the last node in its bucket.

is the vector data (for MainTable) or not exists (for LSHTable), $NEXT$ indicates the offset of the next vector within the same slot of the non-leaf node.

In the following paragraphs, we describe the update process to showcase how we traverse the hash tree.

— **Step 1**: We encapsulate the vector's ID as a leaf node, and save the node in the storage space with the offset o. Given that the LSH value of the vector h and the directory node length l in bits, we use the first $log_2(l)$ bits of h as the vector's slot position in the root node;

— **Step 2**: If the slot has not been occupied, we update the value in the corresponding slot of the root node (or the intermediate non-leaf node) with o and finish the update process. For example, we update the first slot of the root node in Figure 4.4 with

K1's offset 1. Otherwise, we move to Step 3;

— **Step 3**: If the slot has been occupied, and the corresponding memory block is a non-leaf node, we use the next $log_2(l)$ bits of h as the object's slot position in this non-leaf node. We recursively repeat this step until we find an empty slot or we used all bits in the h. If we find an empty slot, we perform the operations in Step 2. Else if the slot has been occupied, and the corresponding memory block is a leaf node offsetting at o', we chain the newly added node to the leaf node by updating $NEXT$ of the new node with o' until the chain size larger than t. When the length of the chain reaches at t, we expand the hash tree by adding a new non-leaf node and redistribute leaf nodes in the new non-leaf node.

The middle part of Figure 4.4 illustrates an update process by chaining the leaf nodes without reaching the chain size threshold t. K3 was originally in the root level of the hash tree, with the offset as 16. After we insert K2 which offsets at 9, we replace 16 in the slot of the root node with 9 and update the NEXT field of K2 as 16. The right most of Figure 4.4 describes the expanding of the hash tree when the chain size reaches t. Suppose we set t as 2. When we are inserting K6 whose offset is 20 to the last slot of the root node, we find that there will be more than t nodes in the same bucket. To spread these nodes to the next level, we add a new non-leaf node with the offset of 21 and replace the 20 in the slot of the root node with 21. We then extract the next $log_2(l)$ bits in the hash values of K4, K5, and K6 and use them as the slot index in the new non-leaf node. For example, the next $log_2(l)$ bits of K4 and K5 refer to the last slot of the new non-leaf node and the next $log_2(l)$ bits of K6 point to the first slot.

The query process of the hash tree is a simplified version of the update, it only locates the vectors with every $log_2(l)$ bits in the hash values.

The fundamental idea of the above algorithm is to adaptively control the resolution to identify nearest neighbors and minimize the involved area of indexing structure being changed with update requests. (1) We progressively include more bits in the hash value of the vectors to form the ID of the bucket where it locates. By setting an explicit threshold t, we can guarantee the effectiveness of the responses to nearest neighbors queries. Given a query data point q, when there are few vectors (less than t) have the common prefix with q, we want to include these data points as the candidates of the nearest neighbors to avoid return empty to the query. When there are more vectors (more than t) in the same bucket, we have to upgrade the resolution of identifying the nearest neighbors. We raise the bar on the minimum length of the common prefix by moving the data points whose hash values differ in the following $log_2(l)$ bits to different slots (Step 4). Without this step, we may have too many candidates to check whether they are within the distance of R from q in an efficient manner. (2) The modification to the indexing structure is performed by changing the value in the slot of non-leaf nodes. Comparing to the state-of-the-art work which sorts the vectors according to LSH values and change the indexing structure by reconstruction [67], we can perform update operations online.

Parameters l and t define the shape of the hash tree by regulating the number of data points in each bucket, and in turn influences the efficiency and effectiveness of nearest neighbors search. Larger value of t increases the chances to find nearest neighbors by enlarging the search range, yet introduces larger overhead for including many data points in similarity calculation to find the ones within the user-defined distance, R. Too small t excludes too many data points which possibly also exclude the ones within R from the query range. The change of the l value exhibits the reversed impact against t on the efficiency and accuracy. With the standard benchmark datasets MNIST [62] and COLOR [56], we prove that PFO retains the salient efficiency and much higher accuracy than the competitors with proper parameter configuration (Section 4.5.5 and 4.5.4).

4.4 Parallel-friendly State Management

Compared to the state-of-the-art LSH design, PFO is more feasible to handle a large number of concurrent query and update requests and maximize system throughput for the admitted requests. PFO benefits from two design innovations: (1) an indexing structure facilitating the parallel data access with LSH-aware partitioning (Section 4.4.1), and (2) a concurrency management module mitigating the cross-threads synchronization (Section 4.4.2). In this section, we will demonstrate the design details of these two innovations.

4.4.1 Parallel-friendly Indexing

To enable the parallelization, we partition the vectors into several groups and make the access to different vectors affect each other as little as possible. PFO adopts Partitioned Hash Forest (PHF) to achieve this object. PHF separate the memory space of a hash table on two levels, Partition and HashTree level. Our goal is to place the vectors which are potential to be the nearest neighbors in the same hash tree. Therefore, the data residing in different partitions or hash trees are not possible to be involved in the same query or update request, enhance requests targeting at various hash trees can be handled in parallel.

The partitioning algorithms in MainTable and LSHTables are different. In MainTable, given a vector v, we apply a hash function designed to minimize the hash conflict, e.g. MurmurHash3 [45], to its vector ID. To locate the hash tree within a partition for v, we extract the first m bits of the hash value as the tree ID. The memory space partitioning in

LSHTable has more challenges than that in MainTable. We have the additional requirement to preserve the distance between the vectors when partitioning the memory space, i.e. we need to ensure that only the vectors in the same hash tree are possible to be the nearest neighbors. Upon achieving this, we retain the desired benefits of PHF: the irrelevant vectors are indexed in different trees so that the trees can be updated in parallel and independently.

The idea of the partitioning algorithm for LSHTables in PFO is to apply the LSH functions for two times. This algorithm is inspired by the Layered LSH [7] which is used to determine the location of a server saving the vector based on its content. To determine the hash tree for a given vector v for an LSHTable, we first apply the LSH functions associated with the LSHTable and get its LSH value h. We take the first m bits of h as the hashing tree ID in *HashTree* layer. However, only partitioning in HashTree level is not precise enough. For example, hash values 01111 and 00000 are significantly different with each other, but they will be indexed in the same hash tree if we set m as 1. We introduce C locality sensitive hashing functions to decide the location of v in *Partition* level. By applying these functions on h, only the similar hashing values are in the same region in Partition level. By introducing C hashing functions in Partition level and extracting m bits from vector's hashing value, we separate the memory space into 2^{C+m} regions each of which corresponds to a hash tree and only the similar vectors are in the same tree.

The first glance of PHF may give a ***wrong*** impression that it is just the conventional *lock-stripping* strategy [50] which manages the data structure with independent locks to mitigate the lock contention. In fact, PHF structure significantly boosts the performance of LSH system that cannot be achieved by the lock-stripping approach and hash table structure with the fixed length of bucket index.

The LSH-based query is essentially the query request target to a particular key range of the hash table, i.e. multiple hash buckets in the conventional hash table. Given the definition of the distance between the hash keys in Section 4.1.1, we need to query across all hash buckets whose index range between $h - 2^{M-KL}$ and h to get all near neighbors candidates. The conventional lock-stripping strategy manages the buckets with the same B_I mod L_{max} value with the same lock, where B_I is the bucket index and L_{max} is the maximum number of locks. For fetching the near neighbors of the query vector, we must obtain up to L_{max} locks. The range-based lock allocation that manages the nearby buckets with the same lock does not work either for the similar reason. The demand for multiple locks for a single read/write request evidently increases the chances of lock contention, thus decreasing the system performance in the concurrent environments.

The parameters C and m influence the throughput of PFO by allowing different level of parallelism. On the other hand, the nearest neighbors might be filtered out because they are assigned to different partitions mistakenly due to the approximate nature of LSH. In this case, the accuracy of PFO suffers from data partitioning. In Section 4.5.5, we evaluate the impact on PFO's throughput and accuracy brought by these two parameters. We prove that PFO exhibits salient throughput and accuracy despite the potential problem that some nearest neighboring vectors are dumped to different partitions/trees.

4.4.2 Concurrency Management

With only the parallel-friendly data structure, it is not enough to maximize the system performance. The reason is that different threads targeting to the same hash tree have to be synchronized. To minimize cross-threads synchronization, we introduce the *Actor*-based concurrency management strategy of PFO [2].

Actor is a memory-efficient entity that maintains its state. The only way to query/update

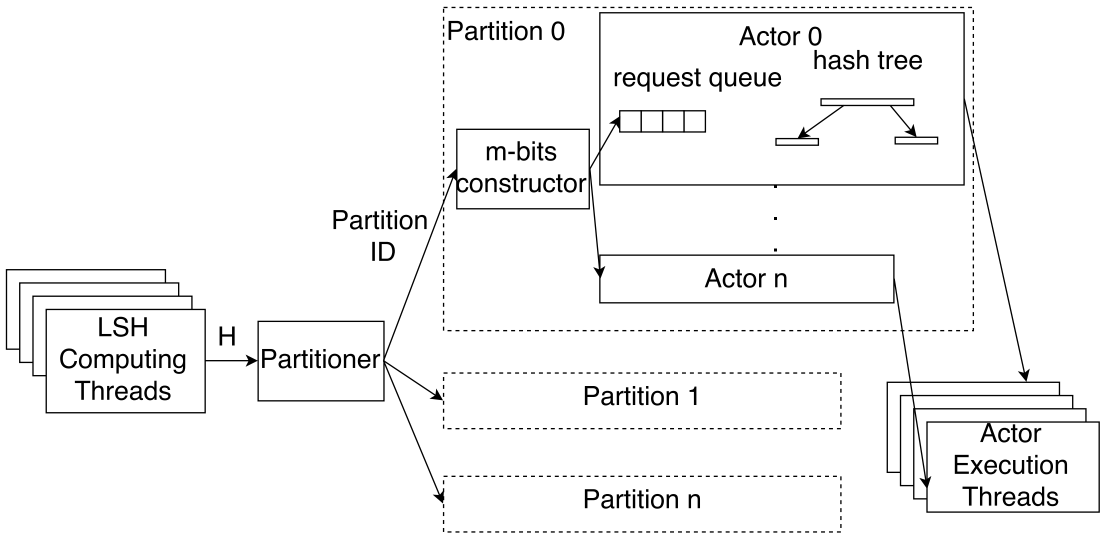


Figure 4.5 – Request Dispatching Module in PFO consists of a group of computing threads and multiple actors each of which maintains a hash tree as its state.

the state of the actors is to send the message to them. At any moment, there is at most one thread having the access to the message queue of the actor. To achieve concurrent processing, Actor model encourages the fine-grained partitioning of state, and computation. The actor is enqueued to the task queue of a thread for processing the message.

As depicted in Figure 4.5, the Concurrency Management module in PFO consists of a group of computing threads and multiple actors each of which maintains a hash tree as its state. The computing threads handle the computation of the hash values of the query vectors. These threads firstly process the requests so that the hash value calculation can be parallelized at the maximum level since there no thread synchronization in this step. After getting the hash value h of the vector v, we locate the hash tree for v with the method we introduced Section 4.4.1. The query/update request is then sent to the actor that maintains the corresponding hash tree as the state. After receiving the request, the actor is attached to a thread for processing the request. Actor-based concurrency module in PFO minimizes the synchronization necessary across the threads. Each thread processes the requests in its full speed instead of pausing a while just because it attempts to access some region of the hash table that is currently updated by others.

A potential problem in this design is the skewed data distribution across hash trees which causes load imbalance among actors. The same issue appears in the general key-value store design [65, 57, 75]. We use a similar partitioning approach with the previous studies [65], i.e. using the first $C + m$ bits to locate the hash tree. With the standard key-value store benchmark, YCSB [24] which exhibits a Zipf-distributed population of size 192M keys with the skewness 0.99, partitioning the memory space with the first 4 bits in hash value of the key makes the most popular partition only 53% more frequently accessed than the average. Additionally, multiple LSHTables in PFO will mitigate the influence of the data skewness with more hash trees. Our experimental results in Section 4.5 proves that PFO keeps high throughput under this potential issue.

4.5 Evaluation

To evaluate PFO, we implement a prototype based on MapDB [59] database engine. MapDB is a pure-Java database, and we choose MapDB because of its clear interfaces and implementation so that we can easily customize MapDB to achieve our goal. We implement the hierarchical memory system by customizing MapDB's storage module and build hash tree based on MapDB's hash tree implementation. We implement Actor-based concurrency management module with Akka [107], and replaced MapDB's multi-threading module with it.

4.5.1 Evaluation Setup

Our testbed equips with Intel(R) Xeon(R) CPU E5-2687W v3 (25M Cache, 3.10 GHz) and 32GB RAM memory. The testbed server uses SK hynix SH920 2.5 7MM 512GB SSD as the external memory. Unless mentioned otherwise, we use 1 LSHTable ($L = 1$) in PFO, and the table is partitioned into 4 partitions ($C = 2$). In each partition, we use 16 hash trees ($m = 4$). Within each tree, we allow at most 500 nodes in the same bucket (except the bottom level) ($t = 500$), the length of the non-leaf node is 128 ($l = 128$).

We use three datasets to evaluate PFO. The first one is Enron Email Dataset [28] which contains around 650,000 text files. We preprocessed the dataset with TF-IDF weighting scheme by only selecting the top weighted 0.5% words, which limits the vector dimensionality as 9,331. We use this dataset to test the scalability and efficiency of PFO because the total number and the dimensionality of the vectors are large. We use MNIST [62] and COLOR [56] datasets to evaluate the accuracy of PFO. The MNIST dataset contains 60,000 points. Each point is a 784-dimensional vector representing the pixel value of a 28 * 28 image. The dataset also contains a test set of 10,000 points. The COLOR dataset contains 68,040 instances, each of which describes the color histogram of an image.

In the following sections, we aim to answer the following questions through the extensive evaluation:

— Does the hierarchical memory design in PFO improve the system efficiency and scalability, comparing with a single-layer design? (Subsection 4.5.2)

— Does the concurrency management in PFO improve the system capability to handle online requests? How does it compare with the conventional multi-thread model and the other indexing structure? (Subsection 4.5.3)

— Does the indexing structure in PFO precisely find the nearest neighbors of the query data point? (Subsection 4.5.4)

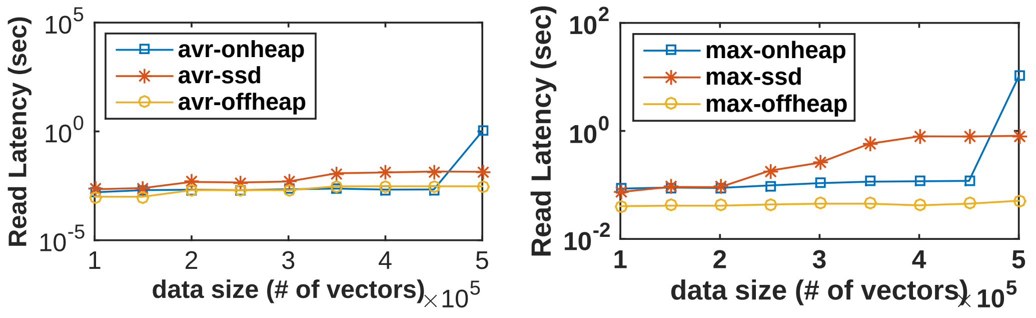
— How is PFO sensitive to the system parameters, i.e. how a user tunes the performance of PFO, regarding efficiency and accuracy? (Subsection 4.5.5)

4.5.2 Test Hierarchical Memory System

In this subsection, we evaluate the hierarchical memory design in PFO by measuring the query latency on each layer with various amount of data. Based on the observation of the latency change on each layer, we can get the maximum data capacity of each layer so that we know whether the hierarchical memory in PFO is superior to a single layer design.

To understand the query latency in the different layers of the memory system of PFO, we run three experiments with all vector data in on-heap, off-heap and flash memory respectively. We query data in the system with ten threads, each of which generated 10,000 read requests. In these experiments, we used an open source Open Addressing Hash Table [99] to index data in the on-heap case; we use PHF to index vectors in off-heap and flash memory (SSD) case. We measured the average and maximum query latency and show the results in Figure 4.6.

Figure 4.6(a) and 4.6(b) demonstrate the average and maximum read latency with three types of storage medium. On average, on-heap memory space offers the shortest read latency with no more than 450,000 vectors. The latency of off-heap and on-heap RAM are nearly the same within this data scale. Fetching the nearest neighbors for a particular query data point from SSD is around 2X slower than RAM (but still delivers sub-second latency). When we scale to 500,000 vectors, we observe that 1) the average latency of on-heap memory dramatically increases with 500,000 vectors; 2) the maximum latency of on-heap memory is always larger than off-heap memory. Both of the phenomena are due to the garbage collection



(a) Average Read Latency of Three Types of Single-Layered Design

(b) Max Read Latency of Three Types of Single-Layered Design

Figure 4.6 – Read Latency of three types of single-layered design

occurring in JVM. When GC happens (even they are just minor GC), GC threads compete with the application threads for CPU resources; the latency of the "unlucky" requests that are processed at the same moment with GC prolong with different levels. The worst case is that when GC module stops the application threads to struggle for more memory space, the maximum latency increased by orders of magnitude, from milliseconds to 10 seconds. In our case, it happens when we have more than 500,000 vectors. Because all the requests are blocked for waiting for GC to complete, the average latency increases accordingly, from milliseconds to second.

Through the extensive evaluation of the read latency in three layers, we have the following two observations:

— **Single-Layered design is not efficient**. With a single flash memory layer, the query latency is around 2x slower than RAM; with the single layer with RAM, the capacity of the system is limited by the total size of physical RAM in the host. On-heap RAM is not a feasible solution for storing a large volume of objects as the GC would degrade performance dramatically.

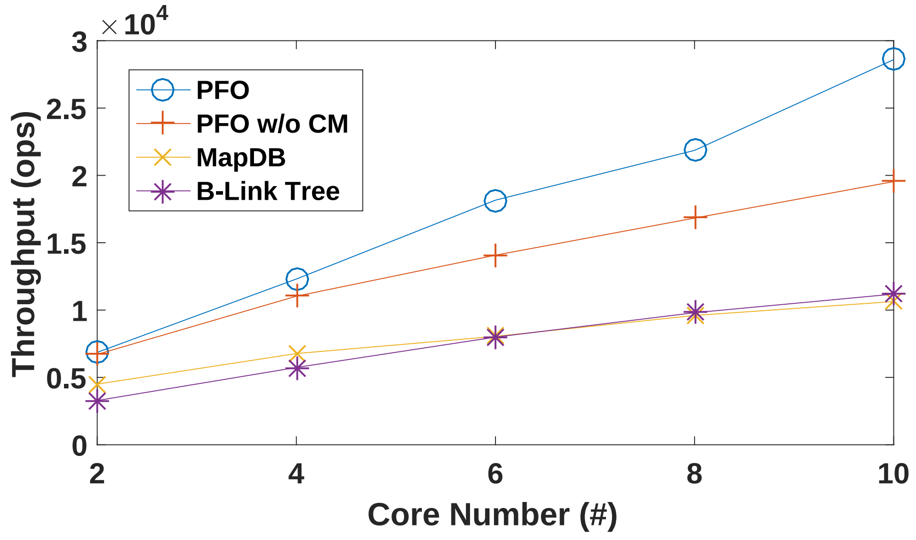


Figure 4.7 – Scalability of PFO in Multi-core platform (L = 1)

— **PFO overcomes the drawbacks by introducing hierarchical memory storage**. By consuming the I/O in off-heap RAM, we can process the request in memory. We scale the system capacity with SSD-based flash memory. When querying data in SSD-based flash memory, we still achieve the sub-second query latency.

4.5.3 Test Parallel-friendly Design

In this subsection, we focus on the testing of the parallel-friendly design of PFO. We test the write throughput of PFO with various numbers of cores and compare it with various indexing structures. We measure the write throughput of the systems when writing 500,000 objects to the memory and show the results in Figure 4.7 and 4.8.

In these two figures, we replace PFO's data partitioning and concurrency management modules with various designs and compare the performance. First, we build PFO on top of the original MapDB (referred as "MapDB"). From Figure 4.7, we observe that we improve

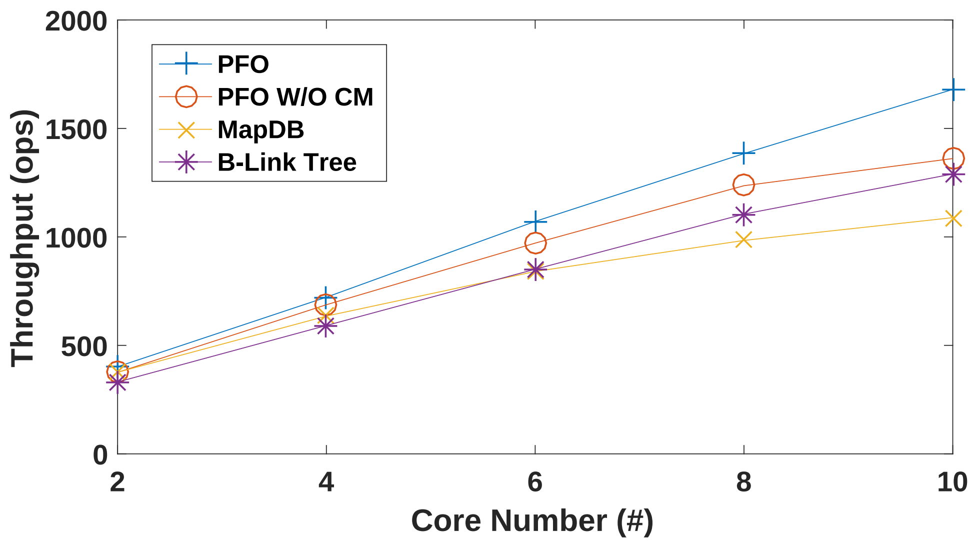


Figure 4.8 – Scalability of PFO in Multi-core platform (L = 10)

MapDB's scalability in PFO and gain a 2.5X speedup on throughput. Second, as B/B+ Tree is one of the most widely used data structure in the previous LSH-based indexing [104, 67], we replace PFO's hash forest with a B-Link Tree [63]. B-Link Tree is a parallel-friendly version of B+ Tree, which adopts the fine-grained locking strategy. We observe that PFO still gains 2.5X speedup over B-Link Tree. Finally, we remove the actor-based concurrency management in PFO ("PFO w/o CM" in Figure 4.7) and compare the performance with PFO. PFO is around 50% faster than PFO w/o CM in this case. Based on the comparison, we prove that the data partitioning and the concurrency management strategy in PFO significantly improve the performance.

In Figure 4.8, we use 10 tables in PFO and evaluate in the same approach as in Figure 4.7. The speedup of throughput ranges between 20% to 70%. We also observe that (1) the overall throughput drops down dramatically; (2) the performance improvement is mitigated comparing to 1-table case. The reason is that the total lock number increases with more tables so that more operations are running concurrently. From this figure, we can see that more hash tables in LSH-based systems not only impose more resource consumption but also makes the system optimization less effective. In the next section, we will show that PFO does not need to introduce many tables to achieve the high accuracy comparing to the state-of-the-art work.

4.5.4 Test Accuracy

To evaluate the accuracy of PFO, we compare the accuracy of PFO and LSB-Tree [104] with the metric in Equation 1. We use two standard benchmarks here, MNIST [62] and COLOR [56]. We use the accuracy metric, error ratio, to measure the accuracy of PFO. Error ratio is widely used in the other LSH-relevant work [104, 67], and it is defined as:

$$r = \frac{1}{k} \sum_{i=1}^{k} \frac{\|\mathbf{o_i}, \mathbf{q}\|}{\|\mathbf{o_i^*}, \mathbf{q}\|} \tag{4.1}$$

where o_i $(i = 1...k)$ are the k objects found in the same bucket with q, o_i^* $(i = 1...k)$ are the ground truth of q's k nearest neighbors. Both o_i and o_i^* are ranked by the increasing order of their distance to q. By finding k nearest neighbors for each query, we are setting the threshold R as the distance from the kth neighbor to the query object. The ideal value of r is 1. We set k as 10.

In Figure 4.9, we use the results from the original paper describing LSB-Tree directly, since we use the same experimental setup. We found that the accuracy of PFO improves significantly with a small number of tables. We attribute the improvement of accuracy to that we precisely use the hash code to index the elements while LSB-Tree converts the hash key to z-order value to be indexed by a B-Tree with the cost of accuracy loss. Additionally,

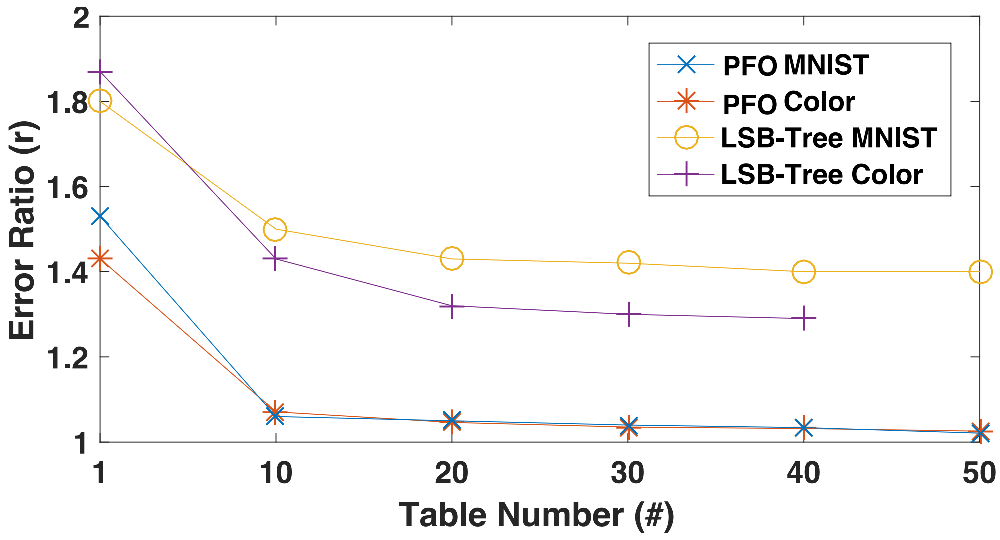


Figure 4.9 – Accuracy of PFO and LSB-Tree (k = 10) (the original paper of LSB-Tree did not report the performance of their system with more than 40 tables)

a small number of tables sufficiently compensate to loss of the data points caused by the data partitioning algorithm of PFO.

4.5.5 Test Parameter Sensitivity

In this subsection, we discuss the sensitivity of PFO against different parameters. The first type of parameters is the parameters regulating the number of data partitions, i.e. C and m. The second type is the parameters deciding the shape of each hash tree, i.e. t and l which regulate how many elements are allowed to be in the same bucket (above the last level of the tree) and the size of the directory node in the hash tree. In this section, we focus on the performance of a single LSHTable.

Data Partitioning Parameters

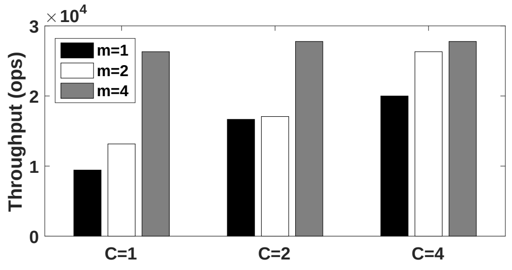
Recall C represents the number of hash functions to determine data point's partition, and m is the number of bits in the data point's LSH value to decide which hash tree it belongs to. To understand how parameters C and m bring impacts to PFO's performance, we show the measurement of PFO's throughput and accuracy against different combinations in Figure 4.10. Figure 4.10(a) shows the system throughput under a synthetic workload. We compose the workload by using ten threads loading 500,000 vectors to PFO. The general trend is that with more data partitions, the system throughput increases accordingly. Because we have only 10 cores in our testbed, the system has the maximum throughput at around 28,000 operations per second. The interesting phenomenon is that the throughput does not increase in proportion to the number of hash trees. The reason is that the elements do not distribute evenly among the hash trees. The skewed distribution can be mitigated by introducing more LSHTables to PFO or more intelligent mapping from hash tree to actors, which would be addressed in our future work.

Figure 4.10(b) shows the accuracy of PFO brought by partitioning the hash table into multiple partitions/hash trees. We also use the error ratio metric defined in Equation 4.1. We use the standard COLOR dataset in the experiment. From Figure 4.10(b), we observe that introducing more partitions/hash trees does degrade the accuracy. In practice, we need to make the tradeoff between the accuracy and system throughput when using PFO.

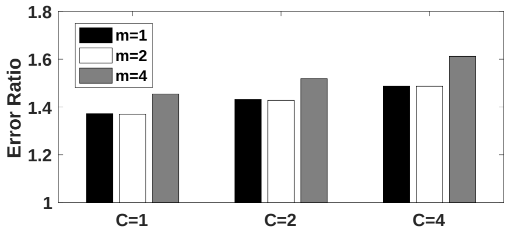
Hash Tree Shaping Parameters

We also test that how the shape of a hash tree influences the accuracy and efficiency of PFO. We investigate by changing the value of l and t, and fix m and C as 2 and 1 respectively.

l and t influence both the efficiency and accuracy of PFO. We used the accuracy metric

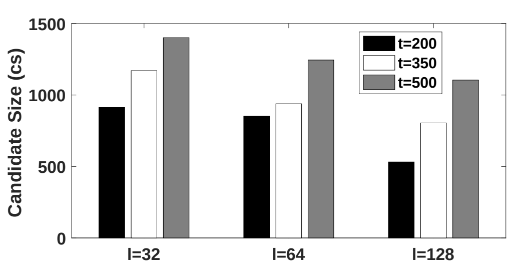


(a) Throughput with different C and m

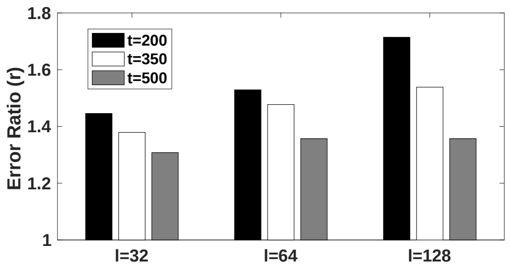


(b) Accuracy with different C and m

Figure 4.10 – System Performance Against different C and m



(a) e with different l and t, the lower the better



(b) Accuracy with different l and t

Figure 4.11 – System Performance Against different l and t

defined in Equation 4.1. To define the efficiency, we measured how many data points are included in the same bucket but have to be excluded since they are far more away with the query data point than the kth nearest neighbors. We call this metric as candidate set size, which is defined as following:

$$cs = \frac{|A(q)|}{k} \quad if|A(q)| > k \tag{4.2}$$

where $|A(q)|$ is the number of the objects which are in the same bucket with object q. The ideal case is that we achieve the salient error ratio with the small cs.

We use COLOR dataset and send 50 requests in each experiment. We report the sum of e for these 50 requests in Figure 4.11(a). Given the fixed l, increasing t to allow more data points in the directory node introduces more overheads when searching the nearest neighbors. However, it increases the chances to find kNN by enlarging the search range as shown in Figure 4.11(b). With the increasing l, the objects are spread in a wider range. Thus we have less read overhead. Consequently, there is a higher probability to put the content-similar objects in different buckets. We observe the higher error ratio with a larger l.

4.6 Related Work

Approximate Nearest Neighbor: There have been the existing efforts to improving the performance of LSH-based NN search system. We summarize the existing work on the new variations of LSH in Table 4.1. Lv et al. [70] proposed to reduce the requested number of hash tables of LSH by probing more data within a table to answer a query. Their ideas of taking the data points with the "similar", instead of strictly identical, hash values as NN candidates were widely adopted in the following research work. LSB-Tree [104], SK-LSH [67] and C2LSH [39] followed this idea with various approaches to improving the efficiency and accuracy of LSH-based systems. All of these approaches prioritize the efficiency of a single query while did not consider how to optimize the LSH indexing structure to facilitate the query/update in a concurrent environment.

Parallel AND Distributed LSH: PLSH [103] and Distributed LSH system proposed in [7] scale the LSH-based NN search system in a distributed fashion. Though the distributed computing schema scales the overall capacity of the system, it is only an efficient solution to improve the system scalability when we have exploited the full potentials of a single host.

"Fully utilize a single before you go to multiple ones" is also a new common agreement in the community [20, 86, 60]. However, the proposed distributed LSH systems did not explore the full potentials of the computing power of a single host. For instance, PLSH only considers using RAM to store LSH tables, and it must broadcast a single request to all machines to fetch the NN results due to the lack of an indexing structure. Hypercurves [105] and GPU-LSH [84] utilized GPU to parallelize the index construction but it does not support to update the indexing structure in online.

Near Neighbours Search via LSH: LSH is not necessary to be used to find the closest point to the query object in the feature space, thus being helpful to reduce search range for applications involving similarity search. For example, Google utilizes LSH [25] to cluster the users before calculating the similarities among the users for news recommendation service. LSH in Google News are calculated with MapReduce [30], i.e. only being calculated in batching but does not fit in the scenario that read/write requests arrive continuously. PLSH proposed in [103] is to streaming the similarity search among the Tweets and LSH is also as a pre-stage minimizing the search scope before calculating the similarities. As stated above, it does not fully utilize the computing power of a single host, e.g. the capacity of the system is limited by the physical RAM size of a host, and it does not maximize the individual host throughput with the concurrent query and update requests.

4.7 Summary of this Chapter

To achieve the online data analytics with the high-dimensional computing state, we need to overcome the challenges on both the computation and storage caused by the increased dimensionality. The state-of-the-art approach is to prune the database to only keep the computing state generated within a window, but at the cost of the compromised system effectiveness.

An approximation-based design like Locality Sensitive Hashing (LSH) accelerates the data analytic applications handling high-dimensional data, e.g. Nearest Neighbor Search. However, the state-of-the-art design of LSH-based indexing structure does not the online state update/query. In this chapter, we have discussed the design of a new computing state management layer based on LSH, called PFO, that supports online update and scales up to 2.5X throughput in multi-core platform.

The idea in this chapter can be extended to a broader range. (1) The approximation-based strategy is feasible to facilitate the incremental computation with the computing state which is complex in structure. (2) Utilizing the modern hardware, like flash memory, is effective to refactor the originally offline-oriented data structure to manage computing state entries online. (3) When developing parallel solutions for computing state management, we need to control the granularity of parallelization in the processing of a request.

Chapter 5

Online Data Analytic Framework

By building an online data processing pipeline and the fine-grained state management layer, we are able to achieve online data analytics in various scenarios, e.g. video content-based indexing, high-dimensional nearest neighbor search. However, we cannot ignore the fact that a lot of existing applications are built with the framework which hasn't provide the facilities to support online data analytics, e.g. Spark [100]. Users have deployed these frameworks in a broad range and build various applications with the programming model of them. Moving to online data analytics with new frameworks/systems and new programming interfaces involves an enormous amount of cost for rewriting programs and adding new components to the users' infrastructure.

In this chapter, we design and implement a data analytic framework supporting online data analytics with the focus on the problem about how to minimize the cost to moving from offline to online data analytics. The goal of our design should include: (1) impose minimum overhead to the existing infrastructure and changes to the programming model; (2) provide the full-fledged state management serving the online data analytic applications.

We introduce Resilient State Table (RST) to Spark, one of the most popular data processing frameworks. RST provides the efficient, scalable and reliable computing state management and integrates with Spark's in-memory abstraction and scheduling policy seamlessly. With RST + Spark, users can build both offline and online data analytics with the same set of APIs and run them within the same framework.

In this chapter, we present the motivation and the design space of RST in Section 5.1. After that, we move forward to discuss the design of RST in Section 5.2, with the focus on the integration with Spark's programming model, scheduling and fault-tolerance mechanism. We describe two applications built with RST + Spark in Section 5.3 and evaluate the performance of RST in Section 5.4. We discuss the difference between RST and the other related work in Section 5.5. We summarize the chapter and discuss the impact of the design principle presented in this chapter in Section 5.6.

5.1 Background

Apache Spark is one of the most popular data analytic frameworks [115, 100]. According to the survey in 2015, more than 1000 companies deploy Apache Spark in their data infrastructure [26]. For example, Janelia Farm [14] consumes 1TB data for every hour with Spark. Other companies like Tencent, the largest social network service in China, maintains the Spark cluster consisting of as many as 8000+ servers.

As we analyzed in Section 2.1, Apache Spark adopts a side-effect-free (functional) programming model facilitating the parallelization, but stands in opposition with online data analytics. The key reason for Spark not be able to fit in online data analytics is the lack of the computing state management layer. The lack of support for accessing online computing state brings the compromised performance and capability to Spark. For example, to run the machine learning algorithms updating the state (i.e. model parameters) iteratively, Spark has to collect all of its states to the single-pointed server and broadcast again for every iteration. It increases the network overhead and compromises the convergence rate of the model training [124].

While researchers propose the frameworks [11, 68, 87, 15, 46, 49] to provide the full-fledged or partial state management capability, they introduce the additional complexity for the user. To move the data analytic applications from offline to online, they have to either rewrite their offline applications with the programming models of these new frameworks, or introduce the additional complexity to their infrastructure, i.e. make Spark co-exist with these frameworks but take care of the complexity of the coordination between the applications written within different frameworks.

We list the design space of state-of-the-art data processing frameworks in Table 5.1. As show in the first partition of the table, despite their prevalence, the mainstream data processing frameworks are designed for stateless parallel data processing, like Spark [115], MapReduce/Hadoop [30, 47] and Dryad [55]. They do not provide the fine-grained-accessible data layer so that they cannot achieve stateful data analytics. Some researchers proposed customized version of the frameworks like HaLoop [13], Incoop [11] and the easy-to-integrate operator [68, 49]. While these proposals build the bridge connecting the existing data-parallel processing frameworks and the stateful data analytics, they fall short in either fine-grained or low-latency access. The other systems propose new frameworks. A few of them still cannot scale the computing state access to the cluster. While the others provide the required state management functionality [37, 87], they impose the overhead on the complexity of the infrastructure and programming model.

In this chapter, we present the design of *Resilient State Table* (RST), which brings the full-fledged state management functionality to Spark with seamless integration so that the users can build online data analytic applications within Spark. Specifically, we make the following contributions:

— **Efficient, Scalable, and Reliable Stateful Data Analytics** RST behaves like hash tables and allows the user to query/update a particular state entry by providing the hash key. Comparing to the frameworks with the local state storage [77, 15], RST scales the state storage to the cluster so that it tackles the scenario with a large volume of computing state. RST tracks how each state entry is generated so as to recover it through recomputing in the case of data loss. Also, it allows creating checkpoint with the minimum interruption to the undergoing processing.

— **Seamless Integration with Spark** RST integrates with both Spark's programming model and scheduling policy. RST provides the Spark-styled programming model to introduce the minimum changes to the existing Spark programs for moving from

Category	System	State Management				Programming Model
		Fine-grained	Low-latency	Iterative	Large State	
Stateless Framework	Spark [115]	Stateless Data-Parallel Processing				Functional
	MapReduce [30]					Functional
	Hadoop [47]					Functional
	Dryad [55]					Graph-oriented
Independent Stateful Frameworks	SEEP [15]	yes	yes	yes	no	Operator-oriented
	SDG [37]	yes	yes	yes	yes	Imperative
	Naiad [77]	yes	yes	yes	no	Graph-oriented
	Flink [38]	partially support	yes	no	yes	Functional
	Samza [23]	partially support	yes	no	no	Operator-oriented
	Storm [106]	partially support	yes	no	no	Operator-oriented
	Piccolo [87]	yes	yes	yes	yes	Imperative
Stateful Component in Stateless Framework	DStream [116]	partially support	yes	yes	yes	Functional
	Comet [49]	no	yes	no	no	Functional
	Incoop [11]	no	no	no	yes	Functional
	HaLoop [13]	no	no	yes	yes	Functional
	CBP [68]	yes	no	no	yes	Operator
	RST	yes	yes	yes	yes	Functional

Table 5.1 – Design space of Stateful Data Analytic Frameworks

offline to online data analytics. It also reduces the overhead on network transfer by co-partitioning with Spark's RDD or batching remote requests.

— **Stateful Data Analytic Pipelines built with Spark + RST** We build the stateful data analytic applications with Spark-like APIs and run them in the Spark cluster. RST not only with the original Spark model but also the more advanced computing model built on top of Spark, e.g. Spark Streaming [116]. Without the necessary to add the additional components to the infrastructure, we are able to perform stateful data analytics with the better performance comparing to the original Spark.

We evaluate RST with the extensive experiments. The results show that the performance of RST scales linearly with the state size and available computing resources. We also compare the performance of data analytic applications built with Spark + RST and the ones built with the original Spark framework. We can achieve significant performance improvements in various scenarios, e.g. real-time web index updates, recommendation system, and machine learning algorithms.

5.2 Resilient State Table

To fill the gap between Spark and the requirements of state management for online data analytics, we introduce Resilient State Table (RST) into Spark. RST is a distributed in-memory hash table providing the fine-grained and low-latency access to the computing state. RST integrates with Spark seamlessly by offering the APIs to the users so that they can query/update the state entries in each Spark computing task. In Section 5.2.1, we have a high-level introduction of RST's components and working mechanism. We then describe the programming model of RST (Section 5.2.2). We discuss how RST interacts with Spark's scheduling strategy in Section 5.2.3, and introduce the fault-tolerance mechanism of RST in Section 5.2.4.

5.2.1 Overview

In Figure 5.1, we illustrate the architecture of Spark enhanced with RST. RST behaves as a hash table which maps the hash key (the input of the query/update requests) to the corresponding state entry. The Spark application has one or more RSTs, each of which corresponds to an individual type of state involved in data analytic process. Each RST has multiple partitions which are stored across the cluster. To manage RSTs, we add a StateManager in each Spark executor and a StateManagerMaster in the Spark Driver. The StateManagers belonging to the same application have the same partitioners per RST which map the state entry's hash key to the partition ID. The Spark tasks running in an Executor query/update state by giving the key to the StateManager in the same Executor. The StateManager queries the IP and port of the StateManager storing the interested partition from either the StateMasterManager or the location cache, which helps to avoid remote location querying for every request. In the case of the inconsistency between the cache and the actual location (e.g. caused by the restarting after Executor failure), StateManager queries StateManagerMaster for the correct location and fix the location cache with the response. Besides partitioner, Merger also locates in each StateManager and is identical per RST. Merger resolves the conflict between the updates to the state entry. For example, in a streaming WordCount program, the merger accepts the update to the frequency for the same word by cumulating them. In contrast, in a stateful log analyzer continuously receiving the last active moment of the user, the merger resolves the conflicts following the last-writer-wins

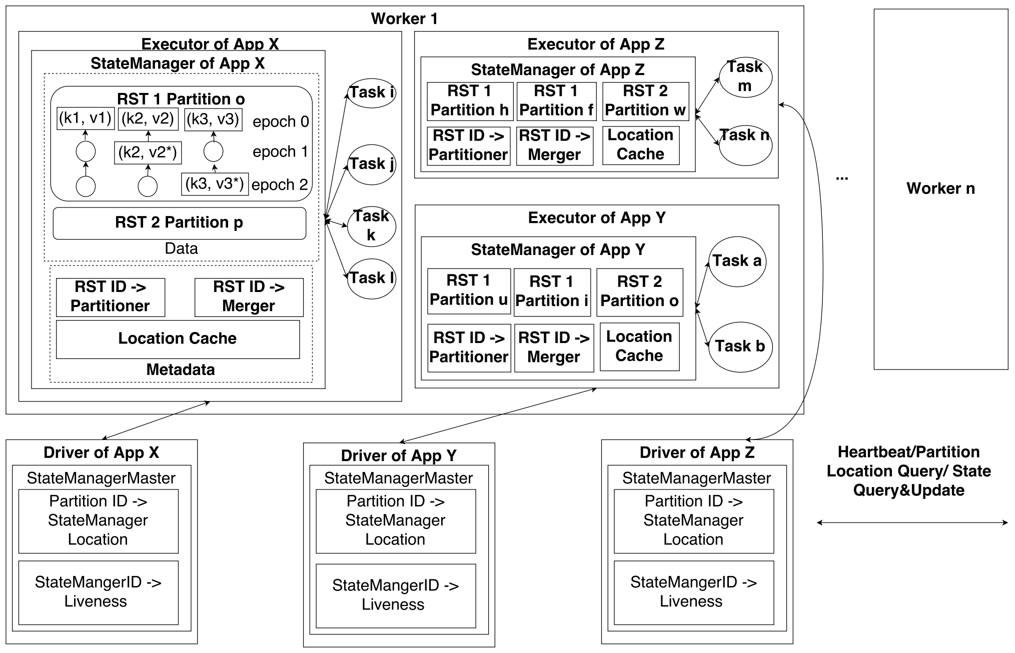


Figure 5.1 – Architecture of Spark with RST

stategy.

The design of RST partitions is inspired by the idea of persistent data structure [80]. Persistent data structure preserves the prior versions of data entries when they are modified. As shown in Figure 5.1, the partition *o* of RST 1 is updated by adding the new key-value pairs in the memory, as well as the pointers referring to the old version of RST partition. In the lastest version of Partition *o* of RST 1, only the pair (K3, V3) is updated to (K3, $V3^*$). When searching K3 in partition 3, RST would response with the latest value $V3^*$ by default. However, for other unchanged pairs, e.g. (K1, V1) and (K2, $V2^*$), it would recursively search in the older version of this partition along the pointers referring there. The circle in Figure 5.1 is the logical representation of the pointer referring to the older version of the state entries. In practice, a unified pointer is referring to the hash table in the last version.

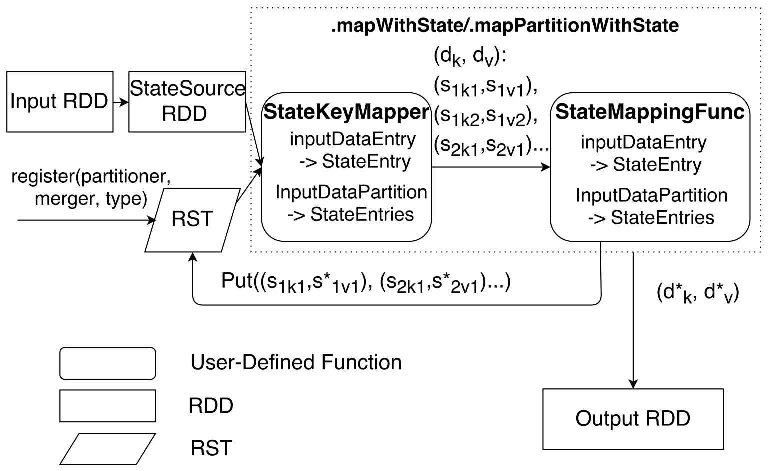

.mapWithState/.mapPartitionWithState
Input RDD
StateSource RDD
register(partitioner, merger, type)
RST
StateKeyMapper
inputDataEntry -> StateEntry
InputDataPartition -> StateEntries
(dk, dv): (s1k1,s1v1), (s1k2,s1v2), (s2k1,s2v1)...
StateMappingFunc
inputDataEntry -> StateEntry
InputDataPartition -> StateEntries
Put((s1k1,s*1v1), (s2k1,s*2v1)...)
(d*k, d*v)
Output RDD
User-Defined Function
RDD
RST

Figure 5.2 – Programming Model of RST

Persistent-style RST facilitates the fault-tolerance mechanism of RST. We represent the version of RST partitions by *epoch*, each of which is associated with the Spark job producing the state entries in this epoch. RST also allows the user to compress the dependency chain from the earliest to the latest version with the checkpoint. When recovering data, we first load the latest checkpoint and then replaying jobs generating the versions of the RST partition which are not covered in the checkpoint. We will have more details about how RST the recompute + checkpoint fault-tolerance mechanism in Section 5.2.4.

5.2.2 Programming with RST

One of the primary goals of RST design is to enable the user to compose stateful data analytic applications within Spark framework. As we stated in Section 5.1, the programming model in Spark is RDD-oriented. Therefore, RST's programming interface is also designed all around RDD.

Figure 5.2 illustrates the major components in the programming interface of RST. We introduce a new type of RDD into the design of RST, *StateSourceRDD*, which is built from the original RDD containing input data entries. StateSourceRDD provides two APIs, *mapWithState* and *mapPartitionWithState*. “mapWithState” maps each input data entry in StateSourceRDD and a specific state entry in RST to an output data entry in the output RDD. For example, when we calculate the frequency of each word in an RDD by aggregating with their existing frequency stored in RST, we will use mapWithState to establish this mapping from the input data entry to the state entry. Being different, “mapPartitionWithState” maps each partition of RDD as well as a set of state entries from RST to a partiton of the output RDD. The usage of “mapPartitionWithState” is feasible in the scenario that the output is derived from the computation over a set of input entries. For example, the machine learning model parameters are calculated by traversing the complete training set (or a subset of the training set) and applying the algorithms over all training samples. In Section 5.3, we will give two real-world applications showing the usage of these two APIs.

In the last paragraph of Section 2.1, we explained the formal definition of the stateful data analytic applications as $output = UDF(input, state)$. Therefore, there are two steps to perform stateful data analytics, mapping from input to state and apply UDF.

StateKeyMapper, as the first parameter taken by mapWithState and mapPartitionWithState, is to establish the mapping from input to state. There are two types of StateKeyMapper. The first category assumes that the data entry and the corresponding state entry is one-one mapping, and share the same hash key. In the above example of counting the words’ frequency, each (word, frequency) pair in RDD is mapped to a single state entry indexed by the same hash key, i.e. word. The other type maps an input data entry (for mapWithState) or a set of data entries (for mapPartitionWithState) to one or more state entries, and the input data and state entry do not need to share the same hash key. For example, in the recommendation system mentioned in [37], all products purchased by a customer (input data) is mapped to one or more state entries in the co-occurence matrix (state), which represents the products which are purchased by the same customer. In Figure 5.2, we show the case that a certain data entry (d_k, d_v) is mapped to a set of state entries $(s_{1k1}, s_{1v1}), (s_{1k2}, s_{1v2}), (s_{2k1}, s_{2v1})$, where s_{ikj} and s_{ivj} stand for the key and value of jth state entry in RST i.

To apply UDF, mapWithState and mapPartitionWithState receive *StateMappingFunc* as the second parameter. The runtime of RST produces the data entries in output RDD by applying StateMappingFunc on the input data entries and their corresponding state entries which are designated by StateKeyMapper. Within the StateMappingFunc, the user is entitled to query/update RST by calling get()/put(). As shown in Figure 5.2, StateMappingFunc not only produces the data entries in the output RDD but also push back the updated state entries, $(s_{1k1}, s^*_{1v1}), (s_{2k1}, s^*_{2v1})$, via put().

The following code snippet implements a stateful word counting program with RST's APIs. Comparing to the example in Section 2.1, we include the state into the word counting logic. In the following code, Line 1 registers RST with the cluster. The last parameter 0 indicates the type of StateKeyMapper, where 0 means that the input data entry and the state entry is sharing the hash key. Line 2 loads raw text corpus from the file system as an RDD and calculate the frequency of each word presented in the current RDD. Line 3 creates a StateSourceRDD based on RDD generated in Line 2. Because each input entry, i.e. (word, frequency) pair, is indexed by the same word and shares the same hash key with its corresponding state entry, we only need to define StateMappingFunc from Line 4 - 8, and use the default sharedHashKeyMapper in the last line, where we apply stateMapperFunc to StateSourceRDD and the newly registered RST to generate the output RDD.

```
val rstId = RST.register(partitioner, merger, type = 0)
val wordRawRDD = sc.textFile("inputFile").flatMap(_.split(' ')).
    map(word => (word, 1)).reduceByKey((freq1, freq2) => freq1 +
    freq2)
val wordCountStateSource = RST.buildStateSource(wordRawRDD)
val stateMappingFunc = (word: String, frequency: Int,
    existingFrequency: Int) => {
  val newFrequency = currentCount + frequency
  put(rstId, word, newFrequency)
  (word, newFrequency)
}
val outputRDD = wordCountStateSource.mapWithState(
    sharedHashKeyMapper, stateMappingFunc, rstId, epoch)
```

This example shows that the programming model of RST interacts with the original Spark's RDD abstraction seamlessly. Therefore, the user can program stateful data analytic application with the popular RDD-based APIs. While the above the example shows the basics of programming model of RST, we will have two more examples from the real world in Section 5.3.

5.2.3 Resilient State Table within Spark

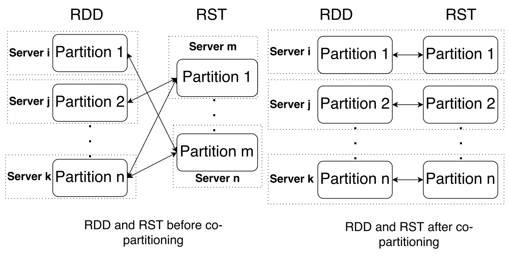
RST serves as a component of Spark instead of a new independent framework. The other challenge for RST to co-exist with the existing Spark components is to operate with the salient performance under the scheduling mechanism of Spark.

Given an available task slot in an executor, Spark decides which task shall be taken to fill the slot with the consideration of data locality. Spark assigns the tasks whose input RDD partition is managed by exactly the executor owning the current task slot with the highest priority so that it avoids the overhead to move the input partition around processes on the same machine and even across the network. Only when the executor with the input data partition is too busy and a timeout happens, the task can be scheduled to another place, and the runtime of Spark moves the input partition with it.

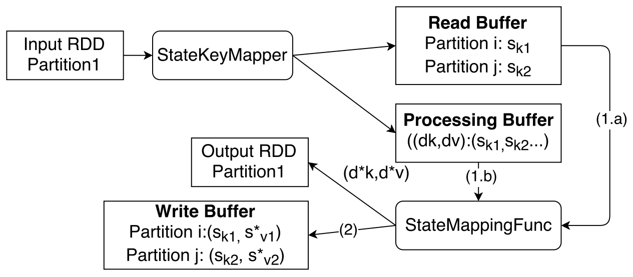
In the online data analytics with the computing state, we would like to achieve not only data locality but also "state locality", i.e. reducing the cost to access the state entries in remote servers. We study the solutions case by case.

The cases are determined by *StateKeyMapper*. As we introduced in Section 5.2.2, StateKeyMapper decides that the data entries in the StateSourceRDD's partition and the corresponding state entries in the RST partition are sharing hash keys or not. When the user register RST with the cluster, as we stated in the above WordCount example, the register() API receives a parameter, "type", which indicates if the input data and state entry share the hash key. The solution to reducing the network-involved overhead for remote state access differs accordingly.

Case 1: Shared Hash Key In the case of shared hash key, we use the identical partitioner for both the StateSourceRDD containing input data and RST containing the state entries, i.e. co-partitioning StateSourceRDD and RST [115]. After co-partitioning, the partitions of StateSourceRDD and RST matches one to one. Figure 5.3(a) shows the difference before and after co-partition. With co-partitioning, we co-locate the associated partitions of StateSourceRDD and RST in the same server. In the case of that Spark schedules task without data locality, i.e. when we need to move StateSourceRDD partition across the network, we



(a) Shared Hash Key between StateSourceRDD and RST: Co-partition of StateSourceRDD and RST



(b) Non-shared Hash Key between StateSourceRDD and RST: Batching Remote State Access

Figure 5.3 – Achieve State Locality and Reduce Remote State Access Under Spark Scheduling Policy

move the required RST partition along with the StateSourceRDD partition to pursue data and state locality.

Case 2: Non-Shared Hash Key If the shared hash key does not apply, the access to the state entries may target to the remote server, and the network-involved overhead for each state access will be prohibitively expensive. Spark handles the similar case in the context of RDD by moving RDD input partition along with the task. Unfortunately, we cannot adopt this method in RST since the access to a particular state entry may from different tasks distributing in multiple servers. Therefore, we batch the requests to query/update state entries to amortize the network cost over multiple state entry access. Figure 5.3(b) illustrates the general design of request batching mechanism in RST. When processing data in an InputRDD Partition, e.g. InputRDD Partition 1 in the figure, StateKeyMapper generates the hash keys of the corresponding state entries and save it in read buffer. In Figure 5.3(b), (d_k, d_v) is mapped to the state entries with the key s_{k1} and s_{k2}. When the size of processing buffer has achieved the threshold, the runtime of RST fetches all state entries indexed by the keys saved in the read buffer. StateMappingFunc is then applied with all read state entries

(Step (1.a)) and input data entries (Step (1.b)) to produce the output data entries ((d^*k, d^*v) in the figure). Simultaneously, in Step (2), the updated state entries ((s_{k1}, s^*_{v1}) and (s_{k2}, s^*_{v2})) are pushed to write buffer and send to the server storing the corresponding RST partition when the current data entries in processing buffer are all processed. A finer-grained request batching mechanism is supported for the following two cases. The first case is that each data entry only reads/writes a single state entry while they do not share the hash key, like the distributed kv workload in [37]. The second case is that each state entry just updates one or more state entries but do need to read, like the recommendation system introduced in [52]. In these two cases, the read and write buffer work in a per partition fashion, i.e. when the number of the requests targeting to a particular RST partition is beyond the threshold, the requests are batched and send to the remote StateManager.

5.2.4 Fault-tolerance

The key idea of the fault-tolerance design in RST is “recompute + checkpoint”, that means, to recover from data loss, RST provides the flexibility to both restarting the Spark jobs updating state entries and reloading checkpoint data. The rationale behind this design is to maximumly accommodate to the fault-tolerance mechanism of Spark. Spark achieves the fault-tolerance with “lineage” mechanism. For each RDD, Spark records its parent RDD and the function applied to the parent to produce the entries in the current RDD. In the case of data loss, the function is applied again for recovery. Eventually, the RDDs involved in a particular Spark job are associated with a dependency chain, which is “lineage”. RDD allows the user to reduce the length of the dependency chain by building a checkpoint.

Recover State With Recompute The basic idea to recover state entries of RST through recompute and fit Spark’s lineage mechanism is to associate each epoch of RST with a StateSourceRDD and StateKeyMapper/StateMappingFunc, and start the Spark job to perform StateMappingFunc over StateSourceRDD again so that the updates to the state entries are re-executed. We keep track the dependency between StateSourceRDD as well as StateKeyMapper/StateMappingFunc and RST in StateManagerMaster side, and StateManagerMaster triggers the Spark jobs to recover data. Since we keep multiple versions of the state entries in RST partition, the state entries can be recovered epoch by epoch.

The most challenge in recomputing is to ignore the duplicate updates, i.e. achieving "exactly-once" semantics. Figure 5.4 illustrates the scenario with a crashed StateManager and the potentially involved duplicate updates to the state entries caused by the restarted Spark jobs. In the beginning, we have two StateSourceRDDs updating RST at epoch 0 and 1 respectively. Each data entry in these two StateSourceRDDs updates both Partition 0 (managed by StateManager i) and 1 (managed by StateManager j) of RST. When StateManager i crashes, Partition 0 is re-assigned to StateManager j and is recovered from recomputing. Therefore, the updates to Partition 0 and Partition 1 are both reproduced, and we need to ignore the updates to Partition 1 but accept the ones to Partition 0.

The approach to ignore the updates from the Spark jobs recovering old epochs is straightforward. We only need to keep track the latest epoch of each partition and ignore the updates from the jobs from passed epoches. In Figure 5.4, because the latest epoch of Partition 1 of RST has been 1, all updates from the Spark job over StateSourceRDD at epoch 0 and 1 are ignored. In contrast, since the newly allocated Partition 0 in StateManager j is just initialized (the epoch is marked as "?"), the updates to it would be accepted.

The duplicate updates can also come from the restarted tasks (due to a transient error in the server or speculative execution [30]) belonging to the *running* job. We still take the scenario in Figure 5.4 as an example. During the running of the Spark job which takes the

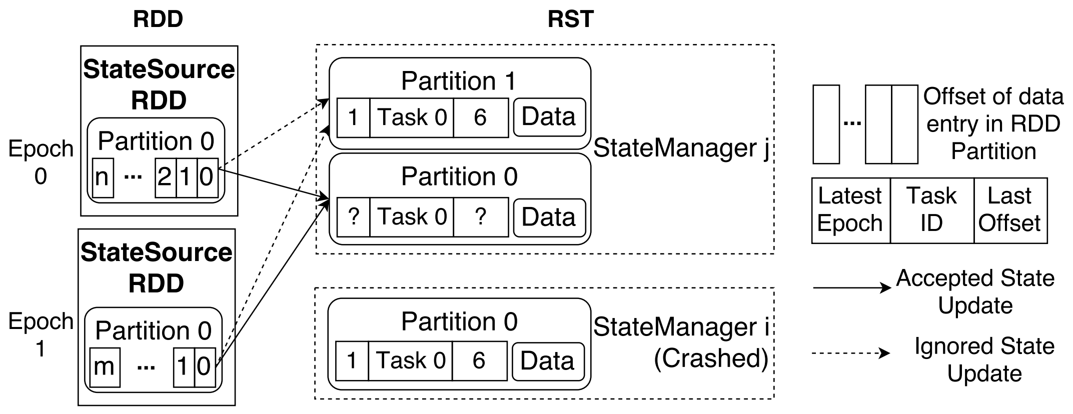


Figure 5.4 – De-duplicate Updates from Restarted Jobs and Tasks. We keep track the lastest epoch and the last offset of the data entry producing the state updates for each RST partition.

StateSourceRDD at epoch 1 as input, the task is restarted after processing 6 elements. The restarted task still starts with the first element. To ensure the correctness, we need to ignore all updates brought by the first six elements if the update is not idempotent. To achieve this, we make the StateManager manages the last offset of the data entry producing the updates to the state entries for each RST partition (e.g. Partition 1 of RST shown on Figure 5.4), and the runtime of RST encapsulates the update requests sent from the Spark tasks to StateManager with the offset of each input data entry. In this way, StateManager is able to identify whether the update request is duplicate.

Asynchronous Local Checkpoint The major challenge to the design of checkpoint is to apply the minimum interrupt to the undergoing processing. The synchronous global checkpoint adopted by systems like Naiad [77] stops processing across the cluster and brings low throughput when the state size is large. While the asynchrnous checkpoint taken in [87] eliminates the necessary to stop the processing, it imposes all nodes to discard the processing beyond checkpoint and roll back to the checkpointed state in case of failure.

To establish the asynchronous checkpoint, we build the checkpoint for the RST utilizing its persistent data structure feature. When the gap between the current epoch E_x and the last checkpointed epoch E_i reaches the threshold, all state entries between E_i and E_x serve as a read-only snapshot serving the undergoing Spark job at epoch E_{x+1}. The checkpoint process only saves the latest value of each state entry to the stable storage system and after finishing that, it compresses the memory usage of the state entries by removing all stale values. To avoid the global checkpoint, we build checkpoint for each partition with an independent file in the stable storage system, e.g. Hadoop Distributed File System (HDFS). The checkpoint for each partition is built independently, and the recovery is a per-partition process.

5.3 Applications

The programming model of RST integrates with Spark seamlessly and supports a broad range of important stateful data analytic applications. Here we showcase the flexibility of RST by building two applications, real-time web index updating 5.3.1 and asynchronous machine learning algorithm 5.3.2.

5.3.1 Refresh Index Against Fast-evolving Web Pages

We begin with the web index updating system based on RST. Internet search engines face the challenge to keep their indices up-to-date against the rapidly evolving web pages. In search engines, the web crawler adds the crawled pages to the "crawled" collection; the index updating program fetches the pages, calculates the score of each URL and triggers the crawler to conduct another round of crawling for the highly scored URLs [68]. To calculate the score of each URL efficiently, the state-of-the-art systems maintain the scores of URLs as computing state and incrementally update them instead of recomputing from scratch for every newly crawled URLs [85].

The major challenge for the index updating system comes from the requirement of low latency. To overcome the challenge, we integrate RST with DStream [116], the streaming computing model built on top of Spark's RDD in-memory abstraction. As introduced in chapter 3, DStream separates the continuous data streams into small chunks based on the arrival time, referred as "micro batches". All data arrived within a period presents as a micro batch and is delivered to Spark as an RDD. Since RST integrates with Spark's RDD-oriented programming model seamlessly, RST works with DStream smoothly. In practice, we apply mapWithState/mapPartitionWithState to every RDD in DStream.

The following Scala code snippet implements the web index program. We register RST with the cluster in Line 1. In this example, the StateKeyMapper directly uses the url as the key of the state entry, so that we register the type of StateKeyMapper as 0, indicating that the input data entry and the corresponding state entry is sharing the hash key. Line 2 creates a DStream pipeline which extracts links from crawled pages and calculate the current inLinks based on crawled info. Line 3 - 7 define the StateMappingFunc, which calculates the score of the URL based on its inLinks update and the existing inLinks in RST, and only returns the URL if the score is larger than a predefined threshold. In the last five lines, we apply StateKeyMapper and StateMappingFunc to each micro batch (represented as RDD) and only output the URLs whose scores are beyond the threshold via filter(). Since the data and state entry share the hash key in this example, we apply the default sharedHashKeyMapper. Note that since Spark 1.6, Spark Streaming (implementation of DStream) provides a API (named mapWithState) which handles the shared hash key case between data and state entry. In Section 5.4.1, we will compare the performance of Web Index updating program based on RST and Spark Streaming.

```
val rstId = RST.register(partitioner, merger, type = 0)
val inLinksForEachPage = createDStreamFromCrawledPages(...).map(
    _.extract_links).map(_.calculateInLinks).reduceByKey(_ + _)
val stateMappingFunc = (url: String, inLinks: Int,
    existingInLinks: Int) => {
  val score = scoringURL(inLinks, existingInLinks)
  put(rstId, url, score)
  if (score > threshold) url else None
}
inLinksForEachPage.foreachRDDWithEpoch { (rdd, epoch) =>
  val urlStateSource = RST.buildStateSourceRDD(rdd)
  val urlRDD = urlStateSource.mapWithState(sharedHashKeyMapper,
      stateMappingFunc, rstId, epoch)
  urlRDD.filter(is not None).saveToCrawlerQueue()
}
```

5.3.2 Asynchronous Machine Learning Algorithms

A large class of the popular machine learning problems is to minimize the prediction error. For example, if we were to predict the stock's price in the following days, the prediction error represents the deviation between the predicted values and the actual price. Machine learning algorithms take the large set of training samples (e.g. the historical data in the stock market) as input and output the predictive model consisting of a certain number of parameters. By filling the corresponding parameter values of a certain stock, e.g. price in the last two days, the model gives the prediction. Machine learning Algorithms iteratively determine the values of the model parameters by minimizing the objective function. The objective function usually presents as the form of:

$$F(w) = \sum_{i=1}^{n} l(x_i, y_i, w) + \omega(w) \tag{5.1}$$

where l is the loss function presenting how accurate the model (with the current parameter value w) can fit with the training samples (x_i and y_i). The second term is the regularization item which prevents the model being "overfitting" by penalizing the model complexity.

One of the most widely used algorithms in finding the optimal values of w is the *gradient descent*. Gradient Descent (GD) evaluates the derivative, or gradient of the current objective function against the complete training dataset and taking steps along with the direction represented by the derivatives. *Stochastic Gradient Descent* (SGD), the approximation variance of the original GD algorithm, moves a step for each training instance. A compromise between these two variance is the mini batch SGD, which moves a step by looking at a subset of the training dataset.

Parallel, Synchronous and Asynchronous SGD To tackle the large volume of training dataset, we usually parallelize the computation of SGD. Spark achieve parallelization by dividing the training dataset into multiple partitions and calculates the gradients for each partition in parallel. At the end of each iteration, Spark collects all these gradients to the Driver node, apply all gradients to update the model parameters, and finally broadcast the parameters to all Executors before moving forward to the next iteration [74]. This algorithm is usually described as *synchronous*. Recent work in the machine learning community propose to break the restriction of updating the parameters at each iteration, and allow the parameter solvers to work concurrently and rely on the *asynchronous* communication for the intermediate gradient steps [89, 51]. The rationale behind the asynchronous machine learning algorithms is that (1) the training of the model converges eventually in a statistical perspective; (2) by reducing the communication overhead, the total training time shall be lower than the strictly synchronous version. The original Spark with RDD cannot achieve asynchronous machine learning algorithm due to the lack of fine-grained state access.

Enabling Asynchronous SGD in Spark With RST, we are able to implement asynchronous SGD in Spark. The following code snippets realize an asynchronous SGD algorithm resembling the idea of Downpour SGD [29]. In Line 1, we register the RST with the cluster. The only difference with the mentioned earlier examples is that we pass the type parameter as 1, which indicates that the input data entry and the state entries do not share the hash key. Line 2 and 3 describe the creation of StateSourceRDD. We have a simplified StateKeyMapper defined in Line 4 which maps a complete partition to all parameters of the machine learning model. In practice, the parameter access pattern in machine learning algorithms exhibits the sparsity [64] so that we can utilize this matter of fact to reduce the communication overhead further. StateMappingFunc declared in Line 5 are performed over a collection of data and state entries, instead of a single instance. In StateMappingFunc, the algorithm iteratively updates the gradients over the trainingSetPartition and synchronize the local updates with the gradients in RST for every "maxCommRate" iterations. The final output of the algorithm is the trainingError in each partition. We apply StateKeyMapper and StateMappingFunc at Line 22 to train the machine learning model. To enhance the reliability of the algorithm, we can break the single StateMappingFunc into finer granularity where each job proceed for maxCommRate iterations so that the model parameters can be checkpointed for every maxCommRate iterations. However, this strategy applies more synchronization in the training process.

```
val rstId = RST.register(partitioner, merger, type = 1)
val trainingSet = loadLabeledPoints(...)
val stateSourceWithTrainingSet = RST.buildStateSource(
    trainingSet)
val stateKeyMapper = (trainingSetPartition: Iterator[
    LabeledPoint]) => (0 until lastParameterIndex)
val stateMappingFunc = (trainingSetPartition: Iterator[
    LabeledPoint], gradients: Array[Double]) => {
    val weights = new Vector
    for (i <- 0 until totalIterations) {
      if (maxCommRate reached) {
        gradients = get(rstId, allKeys)
        updateWeightWithGradients(weights, gradients)
      }
      for (trainingSample <- subset(trainingSetPartition)) {
        gradients = computeGradients(trainingSample, weights,
            gradients)
      }
      updateWeightWithGradients(weights, gradients)
      if (maxCommRate reached) {
        put(rstId, gradients)
      }
    }
```

```
      calculateTrainingError(trainingSetPartition)
  }
  val outputRDD = stateSourceWithTrainingSet.mapPartitionWithState
     (stateKeyMapper, stateMappingFunc, rstId, epoch)
```

In Section 5.4.1, we compare the performance of our algorithm and the synchronous SGD algorithm delivered in Spark's release. We observe that our algorithm exhibits a faster converge rate than the competitor.

5.4 Evaluation

We implement RST based on Apache Spark 1.6.0. We customize the implementation of OpenHashMapBasedStateMap in Spark to support the epoch-based operations of RST. We conduct the extensive experiments to evaluate the performance of RST. The experiments presented in this section consists of four groups. In the first group, we implement three real-world applications with Spark + RST and the original Spark framework respectively and compare the performance under various implementations (Section 5.4.1). The second group tests the scalability of RST under different system parameter setup, workload pattern and the increasing size of computing state (Section 5.4.2). The third group consists of the experiments showing how well RST integrates with Spark's scheduling policy (Section 5.4.3). The final group evaluates the fault-tolerance mechanism of RST on the impact to the undergoing processing and recovery time (5.4.4).

Testbed We run our experiments in a private cluster consisting of 5 servers with 44 cores. The Spark driver runs in a server with Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz (4 cores)/12GB RAM/ST1000DM003-1ER162 HDD. The executors run in 2 types of servers.

The first type contains 3 servers with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (8 cores)/16GB RAM/SAMSUNG SSD PM851 256GB, and the second type has only one server with Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz (20 cores)/32GB RAM/SK hynix SH920 512GB SSD.

If not mentioned specifically, we run the applications with five executors, each of which has 12GB memory and eight cores. We use 100 partitions in RST and batch the read/write requests to RST with the size of 1000.

5.4.1 Compare with Spark

In this subsection, our goal is to understand the question “What is the quantified benefit for introducing state management facilities to Spark?”. We implement the three example applications with RST + Spark and compare them with the implementations based on Spark’s RDD APIs.

Real-time Web Index Updating

The first application is the real-time web index updating which has been described in Section 5.3.1. To compare with the version built with Spark + RST, we implement it with mapWithState API provided in Spark Streaming. We test the implementations with the Web Data Commons-Hyperlink Graph dataset [92] (aggregated on subdomain/host level). The dataset contains 2 billion hyperlinks information across 100M subdomains. We extract 80% entries of the dataset in random and save to RST as the initial state, and use the hyperlinks info in the rest as the micro-batches with various size to test the system.

Figure 5.5 shows the average processing latency of two implementations. From the figure, we observe that two curves are nearly identical. The reason is that the mapWithState API

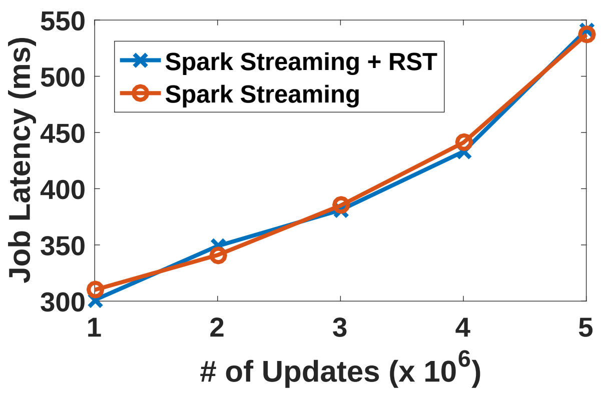


Figure 5.5 – Online Web Index Update Job Latency Comparison between Spark-Streaming-based and RST-based Stateful Implementation

implemented in Spark Streaming is based on OpenHashMapBasedStateMap which makes it equivalent to RST when input data and state entry share the hash key, and we perform co-partitioning to avoid remote access. As we explained in Section 5.3.1, the shared hash key is exactly the case of web index application. From this experiment, we conclude that *the performance of RST is no worth than the current local state management functionality offered in Spark Streaming.*

Recommendation System

The second application is the real-time recommendation system introduced in [52]. The real-time recommendation system collects all user ratings for the products, (userId, itemId, rating), within a time window and build a relevant matrix, M, the element of which $M_{i,j}$ stands for the number of times that item i and j are rated by the same user. As a result, the input data entry and the state entries are one-to-many mapping, i.e. a single (userId, itemId, rating) entry may trigger the update of multiple state entries ($M_{i,j}$).

We implement the application with RST and compare with the version implemented with

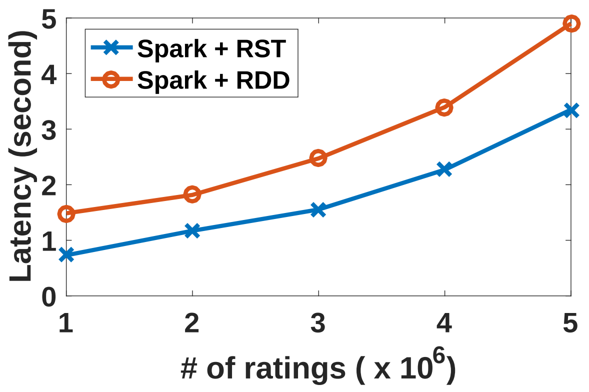


Figure 5.6 – Recommendation System Job Latency Comparison between Spark-based and RST-based Stateful Implementation

original RDD-oriented API. We extract a various number of ratings from the standardized dataset, MovieLens-20M [76], to simulate the user ratings within a time window. In this experiment, we set partition number of RST as 8. From the results in Figure 5.6, we see that the processing latency of RST-based version is much lower than the RDD-based one.

The performance gap comes from the overhead to create additional RDDs in RDD-based version. Without the support of RST, we have to build additional RDDs which contain all pairs of co-rated items and perform aggregation over the RDDs to count the number of co-ratings for each item pair. With RST, we simply send requests to update the entries in RST.

Asynchronous Machine Learning Algorithm

The third experiment we conduct is the asynchronous machine learning algorithm. We run the experiment with URL dataset which was used in [71]. The dataset contains 2,396,130 URLs and 3,231,961 attributes. We implement the Logistic Regression algorithm based on the SGD algorithm introduced in Section 5.3.2, and compare it with the Logistic Regression

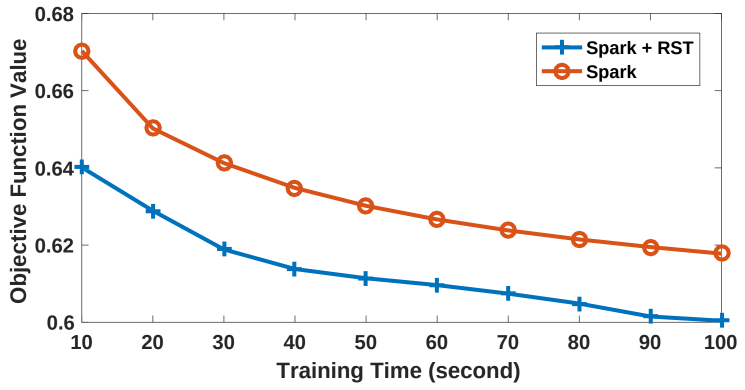


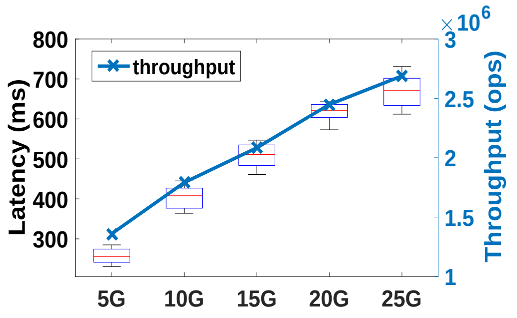
Figure 5.7 – Converage Rate Comparison between Asynchronous Machine Learning implemented with RST + Spark and original Spark, RDD Partition Number = 40, RST Partition Number = 200, Step Size = 1, regParam = 0.01

deliverd in Spark's machine learning library mllib [74].

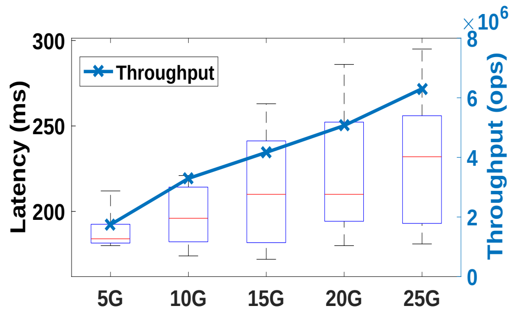
In the experiment, we set "maxSyncRate" as 10 . From Figure 5.7, we observe that the logistic regression algorithm implemented with RST reaches to the smaller loss value within the same amount of training time. The gaining performance comes from the reduced communication overhead and this observation goes along with the previous research work on asynchronous machine learning algorithms [29, 51, 42].

5.4.2 Test Scalability of RST

In this subsection, we study the performance of RST with the different setup. We use a synthetic benchmark, distributed key/value store implemented with RSTs because it is general to exemplify the algorithms which access state entries across the cluster, and used in other representative related work [37]. The figures presented in this section shows the latency distribution of all tasks and the overall throughput of the Spark job. Each task in the Spark job sends 50,000 state read/write requests.



(a) Non-shared Hash Key Between Input Data and State Entry



(b) Shared Hash Key Between Input Data and State Entry

Figure 5.8 – Throughput and Latency of of RST in Distributed Servers

We first test the scalability of RST when scaling to the distributed servers for the large size of the state. The experiment uses the workload with the read/write ratio as 4:1. We measure the aggregated throughput and task latency against different size of state and demonstrate the experimental results in Figure 5.8(a).

With the increasing size of the state, i.e. the increasing number of executors, we observe that the aggregated throughput increases close to linearly. When adding more executors to the application, the increased amount of requests is targeting to remote server acrossing the network. As a result, the latency of a single task increases accordingly but they still keep as sub-second (under 800 ms).

The next experiment we conduct is to understand the performance of RST with different workload pattern. We choose a moderate state size of 15GB. We create workload with various read/write ratio and show the measured task latency distribution and Spark job throughput in Figure 5.9. With the increasing of update requests, more requests need to be synchronized to prevent the race condition caused by the concurrent modification to the same RST partition, so the latency of the tasks increase slightly (within 100 ms).

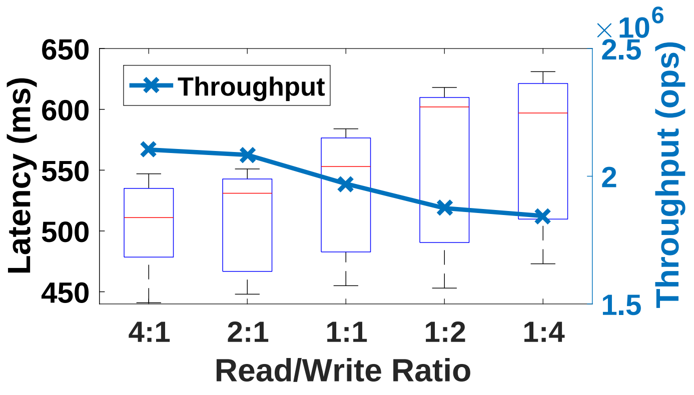


Figure 5.9 – Throughput and Latency of RST against Different Workload Pattern

These two experiments show that RST scales to the large state size and distributed environment. It can fetch and deliver the fresh results with low-latency and high throughput.

5.4.3 Test Integration of RST and Spark

In this part, we evaluate the integration of RST and Spark. Specifically, we test our strategies of (1) co-partitioning RST and RDD when there is shared hash key between data and state entries; (2) The impact of the batching size with the non-shared hash key between data and state entries.

Co-partitioning In Figure 5.8(a), each data entry may access a state entry stored in a remote server. To show the effectiveness of the co-partitioning strategy, we co-partition the RDD and RST in the cluster and conduct the test with the same setup again. We present the throughput and latency in Figure 5.8(b). We observe that without the remote access cost, the task latency is shortened by half, and the throughput increases with 2X accordingly.

Batching Size When co-partitioning is not feasible, we use request batching to reduce the overhead on the network. To understand how to choose a good batch size, we conduct

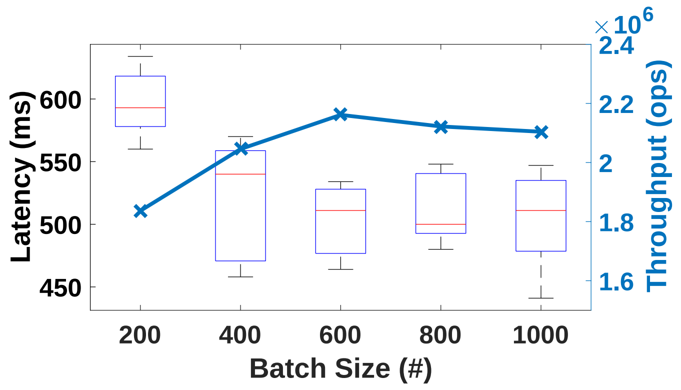


Figure 5.10 – Throughput and Latency against Different Batch Size.

the experiments with 15GB state size and 4:1 read/write ratio. We show the change of task latency and job throughput with various batch size in Figure 5.10. We found that with a small batch size (200), the task latency becomes high as we need to pay more on the network-involving overhead. When we increase the batch size, the task latency drops accordingly with the decreased network cost. However, if we continuous increasing the batch size (beyond 600), we found that the latency goes high again and the throughput degrades. The reason behind the phenomenon comes from two sides: 1) large batched requests take more time to transfer, and 2) with a large batched request object, the program spends more time on the serialization/deserialziation, which is recoganized as one of the major cost in data analytic systems [82].

5.4.4 Fault Tolerance of RST

We compare the asynchronous checkpoint algorithm of RST with its synchronous variance regarding the impact to the undergoing Spark tasks and show the results in Figure 5.11. To test the impact from checkpoint to Spark tasks under different state size, we group the

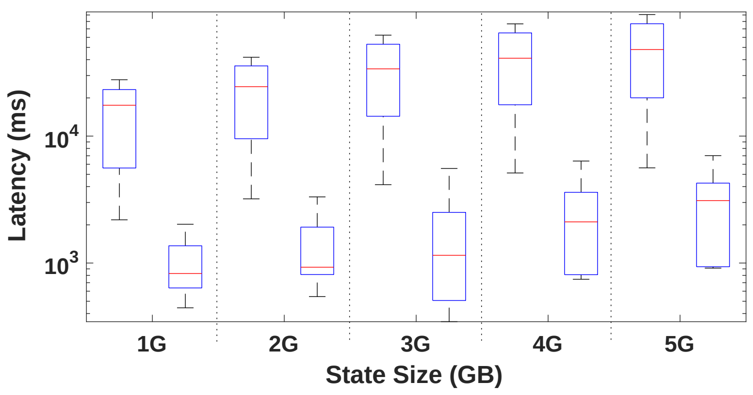


Figure 5.11 – Checkpoint Impact to the Task Latency. We group the experiments into 5 groups, each whcih corresponds to a particular state size. Within each group, the left bar plot stands for the synchronous algorithm and the right one is the asynchronous algorithm adopted by RST.

experiments into five groups, each of which corresponds to a state size, ranging from 1GB - 5GB. In each group presented in the figure, the left bar plots are the results of the synchronous checkpoint and the right is the asynchronous checkpoint. From the figure, we observe that the asyncronous checkpoint algorithm adopted in RST allows the concurrent execution of undergoing Spark job and checkpoint process. Therefore it is one order of magnitude faster than the synchronous algorithm.

We measure the recovery time of 5GB state with the global checkpoint, local checkpoint and recompute against different dependency length chain. For each epoch in the dependency chain, we run the Spark job consisting of 40 tasks each of which updates 50,000 (referred as recompute(s)) or 100,000 state entries (referred as recompute(l)). We shutdown one out of five executors to trigger the data loss. In Figure 5.12, we found that the global checkpoint algorithm has to roll back the whole system state. Therefore, it takes nearly 80s to finish the recovery process. In contrast, the local checkpoint only needs to recover the lost partitions, which shortens the duration of the recovery. In general, the time cost of re-executing a single Spark job is much less than recovering it from the checkpoint. With the larger Spark

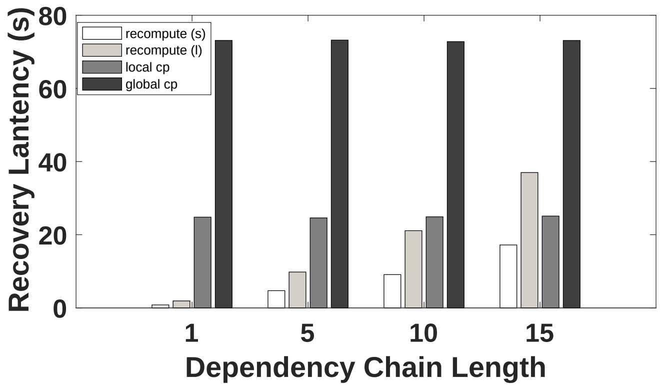


Figure 5.12 – Recovery Latency of Checkpoint and Recompute with 5GB State Against Different Dependency Chain Length.

job (100,000 per task), the time cost for recovering with recompute grows proportionally. When we increase the dependency chain length, the time cost for re-executing the Spark jobs increases accordingly and it eventually exceeds the time cost of recovering from the checkpoint when we have 15 epoches in the dependency chain with the larger Spark job. In this experiment, we demonstrate 1) RST's local checkpoint algorithm is faster than the global one used in other systems; 2) the recovery time cost with recomputing is determined by job granularity. As a result, we need to determine the checkpoint interval (maximum dependency chain length) according to the characteristics of the data analytic workload.

5.5 Related Work

State Management in Data Analytic Framework As we summarize in Figure 5.1, there have been some research efforts to achieve stateful data analytics. These efforts can be classified into two categories, the new frameworks and the refactored stateless frameworks. For the new frameworks, the representative work like Naiad [77] and SEEP [15] adopt graph-

and operator-oriented programming model respectively, but cannot scale to the large state size beyond the capacity of a single host. The other typical studies in this direction include SDG [37] and Piccolo [87]. Both of these two work adopt the imperative programming model and support the large state in distributed environment. However, we cannot directly borrow the ideas from these frameworks to RST as they have the significantly different design philosophy with Spark in various aspects, e.g. programming model, fault-tolerance, etc. The design of RST is orthogonal to SDG in that we share the same motivation to achieve stateful data analytics but we pursue to integrate with the existing frameworks seamlessly.

The other category which proposes to embed the stateful component/abstraction into the stateless frameworks keeps the capability with the existing frameworks like Hadoop. Some of these proposals have no support to fine-grained access to the state [11, 49, 13, 46, 116], because they have to cache the results from previous batching jobs and then derive the future results. A few of the frameworks in this category [38, 23, 106, 116] have the limited support of state management. They assume that the data and state entries share the same hash key, and an individual data entry can only involve a single state entry. As we introduced in Section 5.3.2, this does not fit in the scenario like machine learning. CBP [68] is a system to transform batching jobs into incremental computing jobs automatically, but cannot support low-latency and iterative computing.

Being different with all existing efforts, RST integrates with Spark seamlessly and keeps the functional programming model in Spark framework to avoid ship the user with another new framework and another newly-styled APIs. It scales to large state size and supports fine-grained and low-latency state access.

Stateful Data Analytic Applications within Stateless Framework The other family of approaches to achieve stateful data analytics in the stateless framework is to establish the communication channel between the computing tasks. PD^2F [124] and ASIP [42] fall into this class. However, these approaches focus on providing the per-application solution instead of a general framework like RST.

Failure Recovery The basic idea of failure recovery method for in-memory systems is to replicate checkpoint data to the disk of multiple nodes and recover from multiple disks in parallel. The difference of various approaches is on whether they have to stop the undergoing processing during checkpoint and the synchronization among the nodes during recovery. The systems with synchronous checkpoint [77] have to stop all processing during checkpoint, and the other systems with the global checkpoint requires rollback all nodes to the same checkpoint in case of failure. Being similar to the approach in SDG [37], we adopt the asynchronous local checkpoint in RST, which only roll back the data for lost RST partitions. RAMCloud [81] replicates data across cluster memory by replicating each write request. We are different with it in that we checkpoint the state data to stable storage while allowing the new requests to operate on dirty state.

5.6 Summary of this Chapter

In this chapter, we discuss how to enable the computing state management for online data analytics within Apache Spark, one of the most popular data processing frameworks. We introduce Resilient State Table (RST), which not only provides the scalable, efficient and reliable state management but also integrates seamlessly with Spark's memory abstraction as well as Spark's scheduling mechanism to efficiently perform stateful data analytic tasks. With Spark + RST, users can build online data analytic applications with their familiar Spark-styled APIs and run the applications within the Spark cluster instead of introducing another framework to the infrastructure and adopt another set of APIs.

The design principle represented in this chapter is valuable to a broader range beyond Spark community. Instead of inventing more data analytic frameworks and shipping them with various styled APIs to bother users, this chapter proposes the other practical direction for the system researchers to explore more probabilities, which is proceeding with more respects to the compatibility with the existing systems, especially those which have been adopted in a wide area.

Chapter 6

Conclusion and Future Work

In this chapter, we will first go through the motivations, design philosophies and the contributions in this thesis. We then propose several possible future research directions based on this thesis.

6.1 Conclusion

There has been a significant change in the development of our tools to handle the large volume of data. Just several years ago, the design of the data analytic applications mostly focused on the scalability, i.e. how to consume as many data as possible with the minimum amount of resources. The data analytic logic itself is more like a one-shot process. With the growing of the variety of data source and the speed of data generation, we observe that it is more and more urgent on developing data analytic applications which not only scale to the large data volume but also deliver the data analytic results against the fast evolving dataset promptly. We refer this type of data analytic applications as "Online Data Analytics".

We observe that there are two desired components in online data analytics. The first is an online data processing pipeline, which consumes the input data in an online fashion and connects the multiple stages as a complete pipeline. The second is the state management module which stores the intermediate state of the computing jobs and facilitates us to derive the results from the existing ones.

Consuming input data in an online fashion and pipelining the computing stages is critical and but also challenging. We observe that a lot of applications are built as batching-oriented and have explicitly separate Extract-Transform-Loading (ETL) and data analytics stages. These two design ideas bring the long latency to the data ingestion because we need to accumulate all data, wait for the periodic ETL jobs to start, and then perform the data analytic jobs to get the desired results. To address the issue, we develop Marlin.

Marlin is an online data processing pipeline developed in the context of content-based video indexing. It aims to serve the video-sharing platform and makes the user-uploaded videos searchable in real-time. It contains a streaming feature extractor which transforms the uploaded videos in real-time, and also an online-updatable content-based index to organize the videos according to their visual features. It addresses two major problems in building such a system. The first one is the low throughput caused by the load skew among the inputs. We employ a fair-queuing-based approach to model the videos as flow and intelligently adjust the processing order of the data to improve the system throughput. The second challenge is to connect the ETL and data analytic phases. We make the indexing structure incrementally constructed so that the feature of a video can be sent to the content-based index upon being available instead of being loaded in batches.

State management plays the significant role in online data analytics. Specifically, we need to ensure that the state entries are able to be queried/updated in online. Otherwise, we need to stop all undergoing processing to consume the modification to the state entries in batching or even reconstruct the whole state index to reflect the changes. The challenge arises when we operate with the high dimensional data. The increased dimensionality brings it difficult to store and analyze over the state entries in online.

We design a Parallel-Friendly State Management Layer for Online data analytic applications (PFO) to bring Locality Sensitive Hashing (LSH), which is proved to be the most promising tool for high-dimensional nearest neighbor search, into the online world. To fill the gap between the state-of-the-art LSH design and the requirement of the state management for online data analytics. We employ the hierarchical memory system to overcome the dilemma between the efficiency and capacity; we design an adaptive hash tree to avoid reconstructing the LSH index to consume updates; we intelligently partition state entries in the LSH-based index to facilitate the parallel processing.

After studying the design of the online data processing pipeline and the state management, and successfully porting two offline data analytic system to online, we consider the problem on the other side of the global picture, porting cost. There have been a tremendous amount of applications built on top of the frameworks which does not provide the facilities to support online data analytics. Porting them to the new frameworks/systems involves the huge overhead to rewrite programs and the additional infrastructure complexity brought by the new frameworks.

We propose Resilient State Table (RST), providing the fine-grained, low-latency and reliable state management for the online data analytics. Most importantly, RST seamlessly integrates with the programming model, scheduling mechanism of Apache Spark, which is one of the most widely adopted data processing frameworks. With Spark + RST, users can compose online data analytic applications with their familiar programming model and run the applications within Spark cluster instead of maintaining additional components in the infrastructure.

6.2 Future Work

Developing the guidance to move the offline data analytic applications/systems to the online scenario has the fundamental influence to building the efficient data infrastructure. As the future work, we will focus on the following three aspects.

First, we will explore the approach to improve the online data ingestion system with the deep learning technique [44]. To consume the data in an online fashion, it becomes difficult to adopt the algorithm/data structure involving high time/space complexity to compute, and we have to discard some of the data features when ingesting it. On the other hand, the emerging deep learning algorithms are pretty good at creating the efficient and effective data representation. By building data representation with deep learning, it theoretically possible to keep the data features with a compact representation which is easy to compute and store. However, it also raises several challenges, e.g. in the case of the changed data representation got from deep learning, how to re-index the existing data with the minimum interruption to the undergoing processing. We will explore the various possibilities and challenges in future.

The second future direction is to build the new system architecture to facilitate the application-specific optimization in state management for online data analytics. To achieve the optimal performance, many applications have to embrace the application-specific optimization designs. For example, Locality Sensitive Hashing serves the applications based on nearest neighbor search well but does not make contribute as much to the other fields. In future, we will study the design philosophy on the data analytic system to create a smooth path to integrate these application-specific designs to the a general data processing framework. It involves creating the new abstraction on the data analytic systems and developing expressive programming interfaces to enable the user to describe and integrate their design. The existing efforts on the new system abstraction focus on the data access layer, i.e. how to interpret data within the certain framework. For instance, the graph-based data processing engines interpret data as vertex and edges [43, 69]. When comes to the state management in online data analytics, more attentions go to the storage side, including the file system, database and the internal storage design in data processing frameworks. We will explore the new system abstractions in this direction.

Another possible direction is to combine the features of modern hardware to improve the performance of online data analytic applications. At the moment of the invention of MapReduce [30] as well as the following years, the RAM space is limited in an individual server. Consequently, the data processing frameworks born in those years are mostly designed to support computation with the external memory. With the development of modern hardware, the design philosophy of the systems shall change accordingly. In the recent layers, the server with the large RAM configuration becomes prevalent, so we witness the fast growth of the data processing frameworks, like Spark [100]. In the following years, new hardware will facilitate us to achieve more efficient data analytic system but requires us to adjust the design philosophy. For instance, with the development of Non-Volatile RAM [79], bringing NVRAM to the design of the hierarchical memory system of PFO and the in-memory RST will create a storage layer and requires us to adjust the design to fully utilize the hardware capability. The other example is that the development of Software Defined Network [36, 120, 119] bridges the applications and the network layer, and is potentially beneficial to reduce the network cost in the online data analytic applications by sharing the network information.

Nan Zhu

Education

— Bachelor, Nanjing Institute of Technology, 2005 - 2009

— Master, Shanghai Jiaotong University, 2009 - 2012

— PhD, McGill University, 2012 - Now

Community Service

— **Committer**, XGBoost, One of the most popular large-scale machine learning system in GitHub (`https://github.com/dmlc/xgboost/`)

— **Contributer**, Apache Spark, contributed 60 patches to Apache Spark (`https://github.com/apache/spark/`)

Publication List

[1] Nan Zhu, Lei Rao, Xue Liu, and Jie Liu. “Handling More Data with Less Cost: Taming Power Peaks in MapReduce Clusters”. In: *Proceedings of the Asia-Pacific Workshop on Systems*. APSYS ’12. Seoul, Republic of Korea: ACM, 2012, 3:1–3:6.

[2] Mingyuan Xia, Nan Zhu, Sameh Elnikety, Xue Liu, and Yuxiong He. "Performance Inconsistency in Large Scale Data Processing Clusters". In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13).* San Jose, CA: USENIX, 2013, pp. 297–302.

[3] Nan Zhu, Xue Liu, Jie Liu, and Yu Hua. "Towards a cost-efficient MapReduce: Mitigating power peaks for Hadoop clusters". In: *Tsinghua Science and Technology* 19.1 (Feb. 2014), pp. 24–32.

[4] Nan Zhu and Wenbo He. "ScalaSEM: Scalable validation of SDN design with deployable code". In: *34th IEEE International Performance Computing and Communications Conference, IPCCC 2015, Nanjing, China, December 14-16, 2015.* 2015, pp. 1–8.

[5] Nan Zhu, Lei Rao, and Xue Liu. "PD2F: Running a Parameter Server within a Distributed Dataflow Framework". In: *LearningSys at NIPS* (2015).

[6] M. Zhu, D. Li, F. Wang, A. Li, K. K. Ramakrishnan, Y. Liu, J. Wu, N. Zhu, and X. Liu. "CCDN: Content-Centric Data Center Networks". In: *IEEE/ACM Transactions on Networking* PP.99 (2016), pp. 1–14.

[7] Nan Zhu, Wenbo He, Xue Liu, and Yu Hua. "PFO: A ParallelFriendly High Performance System for Online Query and Update of Nearest Neighbors". In: *CoRR* abs/1604.06984 (2016).

[8] Nan Zhu, Wenbo He, Xue Liu, and Lei Rao. "Towards Full-fledged Stateful Data Analytics within Spark". In: *undersubmission* (2016).

Bibliography

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. “TensorFlow: Large-scale machine learning on heterogeneous systems, 2015”. In: *Software available from tensorflow. org* (2015).

[2] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. “A Foundation for Actor Computation”. In: *J. Funct. Program.* 7.1 (Jan. 1997), pp. 1–72.

[3] Apache. *Cassandra.* `http://cassandra.apache.org/`.

[4] Apache. *Commons Math: The Apache Commons Mathematics Library.* `https://commons.apache.org/proper/commons-math/`.

[5] Apache. *HBase.* `https://hbase.apache.org/`.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. “The Internet of Things: A Survey”. In: *Comput. Netw.* 54.15 (Oct. 2010), pp. 2787–2805.

[7] Bahman Bahmani, Ashish Goel, and Rajendra Shinde. “Efficient Distributed Locality Sensitive Hashing”. In: *Proceedings of the 21st ACM International Conference on Information and Knowledge Management.* CIKM ’12. Maui, Hawaii, USA: ACM, 2012, pp. 2174–2178.

[8] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. “Scaling up all pairs similarity search”. In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 131–140.

[9] Jon Louis Bentley. “Multidimensional Binary Search Trees Used for Associative Searching”. In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517.

[10] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. “The X-tree: An Index Structure for High-Dimensional Data”. In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB ’96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 28–39.

[11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. “Incoop: MapReduce for Incremental Computations”. In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC ’11. Cascais, Portugal: ACM, 2011, 7:1–7:14.

[12] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (July 1970), pp. 422–426.

[13] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. “HaLoop: Efficient Iterative Data Processing on Large Clusters”. In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 285–296.

[14] Janelia Research Campus. *Janelia Research Campus*. `https://www.janelia.org/`.

[15] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. “Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management”. In: *Proceedings of the 2013 ACM SIGMOD International Con-*

ference on Management of Data. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 725–736.

[16] Badrish Chandramouli, Justin J. Levandoski, Ahmed Eldawy, and Mohamed F. Mokbel. "StreamRec: A Real-time Recommender System". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: ACM, 2011, pp. 1243–1246.

[17] Ming-yu Chen, Lily Mummert, Padmanabhan Pillai, Alexander Hauptmann, and Rahul Sukthankar. "Exploiting Multi-level Parallelism for Low-latency Activity Recognition in Streaming Video". In: *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*. MMSys '10. Phoenix, Arizona, USA: ACM, 2010, pp. 1–12.

[18] Ye Chen, Dmitry Pavlov, and John F. Canny. "Large-scale Behavioral Targeting". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '09. Paris, France: ACM, 2009, pp. 209–218.

[19] Erkang Cheng, Liya Ma, Adam Blaisse, Erik Blasch, Carolyn Sheaff, Genshe Chen, Jie Wu, and Haibin Ling. "Efficient feature extraction from wide-area motion imagery by MapReduce in Hadoop". In: *SPIE Defense+ Security*. International Society for Optics and Photonics. 2014, 90890J–90890J.

[20] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John CS Lui, and Cheng He. "VENUS: Vertex-Centric Streamlined Graph Computation on a Single PC". In: *ICDE* (2015).

[21] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. "Managing Data Transfers in Computer Clusters with Orchestra". In: *Proceedings of*

the ACM SIGCOMM 2011 Conference. SIGCOMM '11. Toronto, Ontario, Canada: ACM, 2011, pp. 98–109.

[22] Cisco. *Visual Networking Index (VNI) Forecast*. `http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html#~overview`. 2014.

[23] Samza Community. *Apache Samza*. `http://samza.apache.org/`.

[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154.

[25] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. "Google News Personalization: Scalable Online Collaborative Filtering". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, pp. 271–280.

[26] Databricks. *Spark Summit 2015*. `https://spark-summit.org/2015/`.

[27] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. "Locality-sensitive Hashing Scheme Based on P-stable Distributions". In: *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. SCG '04. Brooklyn, New York, USA: ACM, 2004, pp. 253–262.

[28] Enron Email Dataset. *Enron Email Dataset*.

[29] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. "Large Scale Distributed Deep Networks". In: *Advances in Neural In-*

formation Processing Systems 25. Ed. by P. Bartlett, F.c.n. Pereira, C.j.c. Burges, L. Bottou, and K.q. Weinberger. 2012, pp. 1232–1240.

[30] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113.

[31] Biplob Debnath, Sudipta Sengupta, and Jin Li. "FlashStore: High Throughput Persistent Key-value Store". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 1414–1425.

[32] Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. "Modeling LSH for Performance Tuning". In: *Proceedings of the 17th ACM Conference on Information and Knowledge Management.* CIKM '08. Napa Valley, California, USA: ACM, 2008, pp. 669–678.

[33] Marina Drosou and Evaggelia Pitoura. "Search result diversification". In: *ACM SIGMOD Record* 39.1 (2010), pp. 41–47.

[34] Tamer Elsayed, Jimmy Lin, and Douglas W. Oard. "Pairwise Document Similarity in Large Collections with MapReduce". In: *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers.* HLT-Short '08. Columbus, Ohio: Association for Computational Linguistics, 2008, pp. 265–268.

[35] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol". In: *IEEE/ACM Trans. Netw.* 8.3 (June 2000), pp. 281–293.

[36] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The Road to SDN: An Intellectual History of Programmable Networks". In: *SIGCOMM Comput. Commun. Rev.* 44.2 (Apr. 2014), pp. 87–98.

[37] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. “Making State Explicit for Imperative Big Data Processing”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 49–60.

[38] Apache Flink. *Flink*. `https://flink.apache.org/`.

[39] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. “Locality-sensitive Hashing Scheme Based on Dynamic Collision Counting”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’12. Scottsdale, Arizona, USA: ACM, 2012, pp. 541–552.

[40] Jeremy Ginsberg, Matthew Mohebbi, Rajan Patel, Lynnette Brammer, Mark Smolinski, and Larry Brilliant. “Detecting influenza epidemics using search engine query data”. In: *Nature* 457 (2009). doi:10.1038/nature07634, pp. 1012–1014.

[41] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. “Broom: Sweeping Out Garbage Collection from Big Data Systems”. In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015.

[42] Joseph E. Gonzalez, Peter Bailis, Michael I. Jordan, Michael J. Franklin, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. “Asynchronous Complex Analytics in a Distributed Dataflow Architecture”. In: *CoRR* abs/1510.07092 (2015).

[43] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. “PowerGraph: Distributed Graph-parallel Computation on Natural Graphs”. In: *Pro-*

ceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30.

[44] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning". Book in preparation for MIT Press. 2016.

[45] Google. *MurmurHash3*. `https://code.google.com/p/smhasher/wiki/MurmurHash3`. 2011.

[46] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. "Nectar: Automatic Management of Data and Computation in Datacenters". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 75–88.

[47] Apache Hadoop. *Hadoop*. `https://hadoop.apache.org/`.

[48] Parisa Haghani, Sebastian Michel, and Karl Aberer. "Distributed Similarity Search in High Dimensions Using Locality Sensitive Hashing". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. Saint Petersburg, Russia: ACM, 2009, pp. 744–755.

[49] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. "Comet: Batched Stream Processing for Data Intensive Distributed Computing". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 63–74.

[50] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[51] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server". In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger. Curran Associates, Inc., 2013, pp. 1223–1231.

[52] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. "TencentRec: Real-time Stream Recommendation in Practice". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 227–238.

[53] Piotr Indyk and Rajeev Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality". In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. STOC '98. Dallas, Texas, USA: ACM, 1998, pp. 604–613.

[54] Instagram. *Celebrating a Community of 400 Million*. `http://blog.instagram.com/post/129662501137/150922-400million`.

[55] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72.

[56] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. "iDistance: An Adaptive B+-tree Based Indexing Method for Nearest Neighbor Search". In: *ACM Trans. Database Syst.* 30.2 (June 2005), pp. 364–397.

[57] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. “Chronos: Predictable Low Latency for Data Center Applications”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. San Jose, California: ACM, 2012, 9:1–9:14.

[58] Norio Katayama and Shin’ichi Satoh. “The SR-tree: An Index Structure for High-dimensional Nearest Neighbor Queries”. In: *SIGMOD Rec.* 26.2 (June 1997), pp. 369–380.

[59] Jan Kotek. *MapDB*. http://www.mapdb.org/.

[60] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. “GraphChi: Large-Scale Graph Computation on Just a PC”. In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 31–46.

[61] Martha Larson, Mohammad Soleymani, Pavel Serdyukov, Stevan Rudinac, Christian Wartena, Vanessa Murdock, Gerald Friedland, Roeland Ordelman, and Gareth JF Jones. “Automatic tagging and geotagging in video collections and communities”. In: *Proceedings of the 1st ACM international conference on multimedia retrieval*. ACM. 2011, p. 51.

[62] Yann Lecun. *MNIST dataset*.

[63] Philip L. Lehman and s. Bing Yao. “Efficient Locking for Concurrent Operations on B-trees”. In: *ACM Trans. Database Syst.* 6.4 (Dec. 1981), pp. 650–670.

[64] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. “Scaling Distributed Machine Learning with the Parameter Server”. In: *11th USENIX Symposium on Op-*

erating Systems Design and Implementation (OSDI 14). Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598.

[65] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 429–444.

[66] Yuanqing Lin, Fengjun Lv, Shenghuo Zhu, Ming Yang, Timothee Cour, Kai Yu, Liangliang Cao, and Thomas Huang. "Large-scale image classification: fast feature extraction and svm training". In: *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE. 2011, pp. 1689–1696.

[67] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Heng Tao Shen. "SK-LSH: An Efficient Index Structure for Approximate Nearest Neighbor Search". In: *Proc. VLDB Endow.* 7.9 (May 2014), pp. 745–756.

[68] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. "Stateful Bulk Processing for Incremental Analytics". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 51–62.

[69] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 716–727.

[70] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. "Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search". In: *Proceedings of*

the 33rd International Conference on Very Large Data Bases. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 950–961.

[71] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. "Identifying Suspicious URLs: An Application of Large-scale Online Learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: ACM, 2009, pp. 681–688.

[72] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. "Trash Day: Coordinating Garbage Collection in Distributed Systems". In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015.

[73] John MacCormick, Nicholas Murphy, Venugopalan Ramasubramanian, Udi Wieder, Junfeng Yang, and Lidong Zhou. "Kinesis: A New Approach to Replica Placement in Distributed Storage Systems". In: *Trans. Storage* 4.4 (Feb. 2009), 11:1–11:28.

[74] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. "MLlib: Machine Learning in Apache Spark". In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7.

[75] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. "CPHASH: A Cache-partitioned Hash Table". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: ACM, 2012, pp. 319–320.

[76] MovieLens. *MovieLens-20M*. `http://grouplens.org/datasets/movielens/20m/`.

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

[78] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. “CIEL: A Universal Execution Engine for Distributed Data-flow Computing”. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI’11. Boston, MA: USENIX Association, 2011, pp. 113–126.

[79] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. “Non-volatile Storage”. In: *Queue* 13.9 (Nov. 2015), 20:33–20:56.

[80] Chris Okasaki. *Purely Functional Data Structures*. New York, NY, USA: Cambridge University Press, 1998.

[81] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. “Fast Crash Recovery in RAMCloud”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal: ACM, 2011, pp. 29–41.

[82] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. “Making Sense of Performance in Data Analytics Frameworks”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 293–307.

[83] Cosimo Palmisano, Alexander Tuzhilin, and Michele Gorgoglione. “User Profiling with Hierarchical Context: An e-Retailer Case Study”. In: *Proceedings of the 6th In-*

ternational and Interdisciplinary Conference on Modeling and Using Context. CONTEXT'07. Roskilde, Denmark: Springer-Verlag, 2007, pp. 369–383.

[84] Jia Pan and Dinesh Manocha. "Fast GPU-based Locality Sensitive Hashing for K-nearest Neighbor Computation". In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* GIS '11. Chicago, Illinois: ACM, 2011, pp. 211–220.

[85] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications". In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation.* 2010.

[86] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. "Ringo: Interactive Graph Analytics on Big-Memory Machines". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1105–1110.

[87] Russell Power and Jinyang Li. "Piccolo: Building Fast, Distributed Programs with Partitioned Tables". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.* OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–14.

[88] Marc aurelio Ranzato, Y-lan Boureau, and Yann L. Cun. "Sparse Feature Learning for Deep Belief Networks". In: *Advances in Neural Information Processing Systems 20.* Curran Associates, Inc., 2008, pp. 1185–1192.

[89] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *Advances in Neural*

Information Processing Systems 24. Ed. by J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger. Curran Associates, Inc., 2011, pp. 693–701.

[90] Robbert van Renesse, Yaron Minsky, and Mark Hayden. "A Gossip-style Failure Detection Service". In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Middleware '98. The Lake District, United Kingdom: Springer-Verlag, 1998, pp. 55–70.

[91] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. *Recommender Systems Handbook*. 1st. New York, NY, USA: Springer-Verlag New York, Inc., 2010.

[92] Meusel Robert, Lehmberg Oliver, Bizer Christian, and Vigna Sebastiano. *Web Data Commons*. `http://webdatacommons.org/hyperlinkgraph/2012-08/download.html`.

[93] Rodrygo LT Santos, Craig Macdonald, and Iadh Ounis. "Intent-aware search result diversification". In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM. 2011, pp. 595–604.

[94] Lifeng Shang, Linjun Yang, Fei Wang, Kwok-Ping Chan, and Xian-Sheng Hua. "Real-time large scale near-duplicate web video retrieval". In: *Proceedings of the international conference on Multimedia*. ACM. 2010, pp. 531–540.

[95] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. "Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services". In: *12th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 15). Oakland, CA: USENIX Association, May 2015, pp. 351–366.

[96] M. Shreedhar and George Varghese. “Efficient Fair Queueing Using Deficit Round Robin”. In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication.* SIGCOMM ’95. Cambridge, Massachusetts, USA: ACM, 1995, pp. 231–242.

[97] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. “The Hadoop Distributed File System”. In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST).* MSST ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[98] Jingkuan Song, Yi Yang, Zi Huang, Heng Tao Shen, and Richang Hong. “Multiple feature hashing for real-time large scale near-duplicate video retrieval”. In: *Proceedings of the 19th ACM international conference on Multimedia.* ACM. 2011, pp. 423–432.

[99] Open Source. *OpenHashMap.* `https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/util/collection/OpenHashMap.scala`.

[100] Apache Spark. *Spark.* `https://spark.apache.org/`.

[101] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. “The 8 Requirements of Real-time Stream Processing”. In: *SIGMOD Rec.* 34.4 (Dec. 2005), pp. 42–47.

[102] Inc. Sun Microsystems. *Memory Management in the Java HotSpot.* 2006.

[103] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. “Streaming Similarity Search over

One Billion Tweets Using Parallel Locality-sensitive Hashing". In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 1930–1941.

[104] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. "Quality and Efficiency in High Dimensional Nearest Neighbor Search". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data.* SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 563–576.

[105] George Teodoro, Eduardo Valle, Nathan Mariano, Ricardo Torres, Wagner Meira Jr, and Joel H. Saltz. "Approximate Similarity Search for Online Multimedia Services on Distributed CPU—GPU Platforms". In: *The VLDB Journal* 23.3 (June 2014), pp. 427–448.

[106] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 147–156.

[107] Typesafe. *Akka.* `http://akka.io`.

[108] Rares Vernica, Michael J. Carey, and Chen Li. "Efficient Parallel Set-similarity Joins Using MapReduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data.* SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 495–506.

[109] Kilian Q Weinberger, John Blitzer, and Lawrence K Saul. "Distance metric learning for large margin nearest neighbor classification". In: *Advances in neural information processing systems.* 2005, pp. 1473–1480.

[110] David A. White and Ramesh Jain. “Similarity Indexing with the SS-tree”. In: *Proceedings of the Twelfth International Conference on Data Engineering.* ICDE ’96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 516–523.

[111] Huanmei Wu, Betty Salzberg, and Donghui Zhang. “Online Event-driven Subsequence Matching over Financial Data Streams”. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’04. Paris, France: ACM, 2004, pp. 23–34.

[112] Xiao Wu, Alexander G Hauptmann, and Chong-Wah Ngo. “Practical elimination of near-duplicates from web video search”. In: *Proceedings of the 15th international conference on Multimedia.* ACM. 2007, pp. 218–227.

[113] Alexander G. Hauptmann Xiao Wu Chong-Wah Ngo. *CC_WEB_VIDEO: Near-Duplicate Web Video Dataset.* `http://vireo.cs.cityu.edu.hk/webvideo/`.

[114] YouTube. *Statistics.* `http://www.youtube.com/yt/press/statistics.html`.

[115] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12).* San Jose, CA: USENIX, 2012, pp. 15–28.

[116] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. “Discretized Streams: Fault-tolerant Streaming Computation at Scale”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438.

[117] Jing Zhang, Xianglong Liu, Junwu Luo, and Bo Lang. "Dirs: Distributed image retrieval system based on mapreduce". In: *Pervasive Computing and Applications (ICPCA), 2010 5th International Conference on*. IEEE. 2010, pp. 93–98.

[118] Xiangmin Zhou and Lei Chen. "Monitoring near duplicates over video streams". In: *Proceedings of the international conference on Multimedia*. ACM. 2010, pp. 521–530.

[119] M. Zhu, D. Li, F. Wang, A. Li, K. K. Ramakrishnan, Y. Liu, J. Wu, N. Zhu, and X. Liu. "CCDN: Content-Centric Data Center Networks". In: *IEEE/ACM Transactions on Networking* PP.99 (2016), pp. 1–14.

[120] Nan Zhu and Wenbo He. "ScalaSEM: Scalable validation of SDN design with deployable code". In: *34th IEEE International Performance Computing and Communications Conference, IPCCC 2015, Nanjing, China, December 14-16, 2015*. 2015, pp. 1–8.

[121] Nan Zhu, Wenbo He, Yu Hua, and Yixin Chen. "Marlin: Taming the big streaming data in large scale video similarity search". In: *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*. 2015, pp. 1755–1764.

[122] Nan Zhu, Wenbo He, Xue Liu, and Yu Hua. "PFO: A ParallelFriendly High Performance System for Online Query and Update of Nearest Neighbors". In: *CoRR* abs/1604.06984 (2016).

[123] Nan Zhu, Wenbo He, Xue Liu, and Lei Rao. "Towards Full-fledged Stateful Data Analytics within Spark". In: *undersubmission* (2016).

[124] Nan Zhu, Lei Rao, and Xue Liu. "PD2F: Running a Parameter Server within a Distributed Dataflow Framework". In: *LearningSys at NIPS* (2015).