# Recurring Query Processing on Big Data

by

Chuan Lei

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

August 18, 2015

APPROVED:

Professor Elke A. Rundensteiner
Worcester Polytechnic Institute
Advisor

Professor Mohamed Y. Eltabakh
Worcester Polytechnic Institute
Co-Advisor

Professor Emmanuel O. Agu
Worcester Polytechnic Institute
Committee Member

Dr. Nesime Tatbul
Intel Labs/MIT CSAIL
External Committee Member

Professor Craig Wills
Worcester Polytechnic Institute
Head of Department

# Abstract

The advances in hardware, software, and networks have enabled applications from business enterprises, scientific and engineering disciplines, to social networks, to generate data at unprecedented volume, variety, velocity, and varsity not possible before. Innovation in these domains is thus now hindered by their ability to analyze and discover knowledge from the collected data in a timely and scalable fashion. To facilitate such large-scale big data analytics, the MapReduce computing paradigm and its open-source implementation Hadoop is one of the most popular and widely used technologies. Hadoop's success as a competitor to traditional parallel database systems lies in its simplicity, ease-of-use, flexibility, automatic fault tolerance, superior scalability, and cost effectiveness due to its use of inexpensive commodity hardware that can scale petabytes of data over thousands of machines. *Recurring queries*, repeatedly being executed for long periods of time on rapidly evolving high-volume data, have become a bedrock component in most of these analytic applications. Efficient execution and optimization techniques must be designed to assure the responsiveness and scalability of these recurring queries. In this dissertation, we thoroughly investigate topics in the area of recurring query processing on big data.

In this dissertation, we first propose a novel scalable infrastructure called *Redoop* that treats recurring query over big evolving data as first class citizens during query processing. This is in contrast to state-of-the-art MapRe-

duce/Hadoop system experiencing significant challenges when faced with recurring queries including redundant computations, significant latencies, and huge application development efforts. Redoop offers innovative window-aware optimization techniques for recurring query execution including adaptive window-aware data partitioning, window-aware task scheduling, and inter-window caching mechanisms. Redoop retains the fault-tolerance of MapReduce via automatic cache recovery and task re-execution support as well.

Second, we address the crucial need to accommodate hundreds or even thousands of recurring analytics queries that periodically execute over frequently updated data sets, e.g., latest stock transactions, new log files, or recent news feeds. For many applications, such recurring queries come with user-specified service-level agreements (SLAs), commonly expressed as the maximum allowed latency for producing results before their merits decay. On top of Redoop, we built a scalable multi-query sharing engine tailored for recurring workloads in the MapReduce infrastructure, called *Helix*. Helix deploys new sliced window-alignment techniques to create sharing opportunities among recurring queries without introducing additional I/O overheads or unnecessary data scans. Furthermore, Helix introduces a cost/benefit model for creating a sharing plan among the recurring queries, and a scheduling strategy for executing them to maximize the SLA satisfaction.

Third, recurring analytics queries tend to be expensive, especially when query processing consumes data sets in the hundreds of terabytes or more. Time sensitive recurring queries, such as fraud detection, often come with tight response time constraints as query deadlines. Data sampling is a popular technique for computing approximate results with an acceptable error bound while reducing high-demand resource consumption and thus improving query

turnaround times. In this dissertation, we propose the first fast approximate query engine for recurring workloads in the MapReduce infrastructure, called *Faro*. Faro introduces two key innovations: (1) a deadline-aware sampling strategy that builds samples from the original data with reduced sample sizes compared to uniform sampling, and (2) adaptive resource allocation strategies that maximally improve the approximate results while assuring to still meet the response time requirements specified in recurring queries.

In our comprehensive experimental study of each part of this dissertation, we demonstrate the superiority of the proposed strategies over state-of-the-art techniques in scalability, effectiveness, as well as robustness.

# Acknowledgements

The growth of my knowledge over the last few years culminating with my dissertation is to a huge part due to the inspiration and guidance I received from my advisor, Professor Elke A. Rundensteiner. She gave me the freedom to explore any topic in database research, provided sound directions at every turn, and the prompt feedback that pushed my research envelope. I have been fortunate to have her as my advisor. I express my sincere thanks for her support, advice, patience, and encouragement throughout my graduate studies. Her excellence in teaching as well as tackling complex research problems will always be my inspiration to continue my research in future.

I sincerely thank the members of my Ph.D. committee, Professor Mohamed Y. Eltabakh, Professor Emmanuel O. Agu, and Dr. Nesime Tatbul for providing me valuable feedback during the various milestones in my Ph.D. Their insight and critique helped me improve the contents of this dissertation. I would like to thank Professor Joshua D. Guttman for his guidance during my research qualification. I am thankful for the financial support I have received from my advisor Prof. Elke A. Rundensteiner and my department. My thanks also goes to the National Science Foundation (NSF) for providing funding for the computing resources used in my dissertation.

I would like to thank DSRG team members – in particular Lei Cao and Medhabi Ray for their insightful discussions. In addition, I would also like to thank Rimma Nehme and Karen Works for their help in my early Ph.D.

study. I very much appreciated the discussions as well as friendship of Chiying Wang, Zhongfang Zhuang, Xiao Qin, Dongqing Xiao, and all the other previous and current DSRG members.

I would like to thank my spouse Yuyu Feng for her patience, support and love during the past few years. Her confidence in me and encouragement was in the end what made this dissertation possible. My parents, Zidong Lei and Xiaoquan Wu, receive my most sincere gratitude. Their passion to achieve bigger and better things ingrained in me a drive to reach excellence.

# My Publications

## Publications Contributing to this Dissertation

### Part I: Supporting Recurring Queries in Hadoop

Part I of this dissertation addresses the problem of efficiently evaluating recurring queries over large-scale evolving data sources.

1. Chuan Lei, Elke A. Rundensteiner, and Mohamed Y. Eltabakh, *Redoop: Supporting Recurring Queries in Hadoop*, **EDBT**, 2014, pages 25-36.

   *Relationship to this dissertation:* In this work, we propose Redoop - an optimized framework to evaluate recurring queries. Chapters 3 to 8 in Part I of this dissertation are based on this work.

2. Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, and Mohamed Y. Eltabakh, *Redoop Infrastructure for Recurring Big Data Queries*, **PVLDB 7(13)** 2014, pages 1589-1592.

   *Relationship to this dissertation:* In this demo paper, we present the overall full-fludged infrastructure with native support for recurring big data queries. Redoop is a comprehensive extension to Hadoop that pushes the support and optimization of recurring queries into Hadoops core functionality. While backward compatible with regular MapReduce jobs, Redoop achieves an order of magnitude better performance than Hadoop for recurring workloads. Redoop employs innovative window-aware optimization techniques for such recurring workloads including adaptive window-aware data partitioning, cache-aware task scheduling, and interwindow caching mechanisms. This demo paper includes key elements of Parts I of this dissertation.

## Part II: Shared Execution of Recurring Workloads in MapReduce

Part II of this dissertation addresses the motivating need for accommodating hundreds or even thousands of recurring analytics queries that periodically execute over frequently updated data sets.

3. Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, and Mohamed Y. Eltabakh, *Shared Execution of Recurring Workloads in MapReduce,* **PVLDB 8(7)** 2015, pages 714-725.

   *Relationship to this dissertation:* In this work, we present Helix – a scalable multi-query sharing engine tailored for recurring workloads in the MapReduce infrastructure. It introduces a cost/benefit model for creating a sharing plan among the recurring queries, and a scheduling strategy for executing them to maximize the SLA satisfaction. Part II (Chapters 9-12) of this dissertation is based on this work.

## Part III: Fast Approximate Recurring Queries in MapReduce

Part III of this dissertation addresses the problem of processing recurring queries with approximation to meet the requirements of accuracy and responsiveness.

4. Chuan Lei, Elke A. Rundensteiner, and Mohamed Y. Eltabakh, *Fast Approximate Recurring Queries in MapReduce,* **in submission**.

*Relationship to dissertation:* In this work, we propose our Faro system, an approximate query engine for recurring workloads in the MapReduce infrastructure with proposed deadline-aware sampling strategy and adaptive resource allocation strategies that together improve the approximate results while assuring to still meet the queries' response time requirements. Part III (Chapters 13-17) of this dissertation is based on this work.

## Other Publications

The below listed publications are outcomes of my *Ph.D. Research Qualifier* in distributed continuous stream processing systems at WPI.

5. Chuan Lei, and Elke A. Rundensteiner, *Robust Distributed Query Processing for Streaming Data,* **ACM Trans. Database Syst.** 39(2): 17 2014.

6. Chuan Lei, Elke A. Rundensteiner, and Joshua D. Guttman, *Robust distributed stream processing,* **ICDE** 2013, pages 817-828.

7. Rimma V. Nehme, Karen Works, Chuan Lei, Elke A. Rundensteiner, and Elisa Bertino, *Multi-route query processing and optimization,* **J. Comput. Syst. Sci** 79(3) 2013, pages 312-329.

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

## 1.1 Background

The advances in hardware, software, and networks have enabled applications from business, government, to science to generate data at unprecedented volume, variety, velocity, and varsity—aka so called 'big data'. Some innovations in these domains are poised to be unleashed if only truly effective methods were available to analyze and discover knowledge from the collected data in a timely fashion. To keep pace, major research and development efforts are now being devoted to facilitate such big data analytics. This ranges from designing new processing infrastructures [1, 2, 3, 4], query languages [5, 6], query optimization algorithms [7, 8, 9], and workflow management systems [10, 11, 12, 13, 14], to scalable data mining tools [15, 16, 17, 18, 19, 20]. Among the proposed infrastructures, the MapReduce computing paradigm [1, 21] and its open-source implementation Hadoop [2, 22] have emerged as popular, widely used technologies for cloud-scale data analytics [23, 24, 25, 26, 27]. Hadoop's success as a competitor to traditional parallel database systems [28, 29] lies in its simplicity, flexibility, automatic fault tolerance, superior scalability, and cost effectiveness due to its use of inexpensive commodity hardware that can scale to petabytes of data over thousands of machines—while being open-source and thus inexpensive.

Existing techniques for big data analytics target batch processing of data [30, 31, 32, 33, 34, 35, 36, 37]. However, with the increasing variety of applications generating big data at an unprecedented velocity others aim to consume such data in a near instantaneous fashion to gain their competitive advantage. The required analytical needs far outpace this traditional offline batch processing mode. To achieve high responsiveness yet handle big data, we require technology that is highly optimized for these new demanding workloads. In particular, we now introduce a critical class of queries, called "recurring queries", prevalent in most cloud-based large-scale applications. Yet these queries are not naturally supported by state-of-the-art systems [38, 39, 40].

Recurring queries are common in applications from log processing [41], news feed updates [42], to social networks [42]. Such queries collect huge volumes of fresh data (in TBs per hour or day) to be continuously integrated into deep data analytics. As a result, these applications tend to re-execute over and over similar or even identical analytical queries on regularly refreshed subsets of a big data feed, e.g., the last 24-hour window of a click stream. Even more challenging, complex analytics warrant analyzing the data at *different time granularities* ranging from *offline batch-processing jobs*, *recurring jobs* that repeat periodically consuming the last n hours, days, or even a month's worth of data, to *responsive ad-hoc jobs* that rely on the delivery of faster results at the expense of accuracy and completeness.

As we will highlight in the motivating examples below, recurring queries are not only extremely commonplace across applications critical to our modern economic landscape, but they also form huge workloads characterized by combined characteristics from both real-time continuous queries and offline ad-hoc queries. These recurring workloads demand tremendous resources in terms of CPU processing time, responsiveness, and memory utilization, putting an unsustainable strain on modern infrastructures and computing costs of a company's budget. This warrants the need for the development of customized solutions that uncover and then leverage unique optimization opportunities tuned to recurring query models, as we explore in this dissertation.

## 1.2 Motivation

**Example 1.1 (Clickstream Analytics For Online Advertisement)** *One of the core applications of Internet-based companies is clickstream analysis, where companies (acting as brokers) build models, e.g., statistical linear regression or decision trees, that capture the relationship between companies hosting websites (called publishers), companies advertising and selling products on the publishers' websites (called advertisers), and the end-users visiting the websites and buying products. These predictive models are used for deciding, within a few milliseconds, which advertisement to display on a website upon observing a user's browsing history. The up-to-date maintenance of these models is typically achieved by recurring queries, e.g., a query executes every day to process the last week or two of data (in order of 100s of TBs) to update the predictive model within a user specified time. Companies like Amazon and eBay rely on the speed in which they can produce and apply the refreshed models to their product recommendations.*

In this dissertation, we set out to develop an infrastructure that supports recurring queries as first-class citizens in Hadoop without sacrificing any of its core processing paradigms.

**Example 1.2 (LinkedIn's News Feed Updates)** *On modern consumer websites, news feed generation is nowadays driven by online systems. In spite of the complexity of the involved analytical processing, it is still far from optimized due to the lack of superior technology. Online news services may be generated on a per-member basis based on the member's*

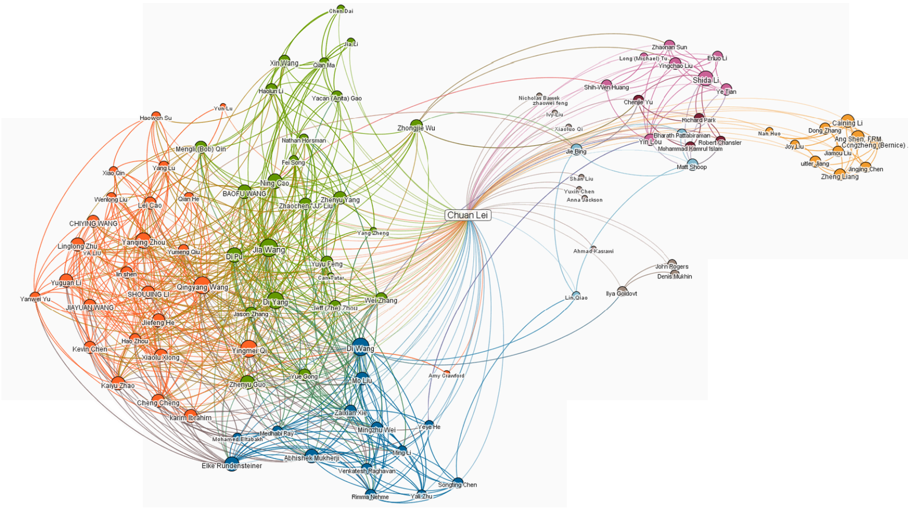


**Figure 1.1:** LinkedIn inMap

*interactions with other components in the system. For example, a LinkedIn member may receive periodic updates on their connection changes (as depicted in Figure 1.1). Computing these updates involves deep analytical processing of large-scale data sets across multiple sources. To generate an update highlighting the company in which most of a member's connections have worked in the past month requires joining the companys data of various profiles. The update is often delivered to the members by the end of each day or each week. Such updates could clearly be expressed by recurring queries running over evolving data sources. Freshness of results dictated by frequency of updating is to a large degree limited by the processing speeds and resource availability to churn through the data. Hundreds or even thousands of similar and possibly sharable recurring queries may be executed over evolving data on a multi-tenant system every day. Consider the following three recurring queries.*

- $Q_1$: A company recruiter attempts to find candidates that she may know based on her connections for job references. She wants to be updated about new potential candidates having sufficiently strong connections made in the past week (window

$w$). The query will be issued every morning (slide $s$), and the results will have the highest utility if returned within 5 minutes (Service Level Agreement constraint, SLA in short) such that she can refine her search and quickly identify potential candidates before moving on to other tasks.

- $Q_2$: A LinkedIn member is searching for openings from companies at which his connections work. He subscribes to a daily service from LinkedIn indicating that he wants to be alerted about attractive new positions shortly after they appear on the job market (say within 1 hour or latest by end of day).

- $Q_3$: LinkedIn data scientists design a mining application that detects emerging patterns from online discussion groups over the last 10 days (window $w$). They would like to find out the most sought-after skills required by mining job openings at the big data companies. The identified skills should be updated on a daily basis (slide $s$), and the results should be produced within 2 hours (SLA constraint) such that other predictive models can consume these results and decide on the recommendations to provide to LinkedIn service subscribers.

The above queries differ in their requirements and constraints, e.g., the SLA requirements vary significantly. In this dissertation, we tackle the problem of handling recurring queries with diverse window constraints and SLA requirements.

**Example 1.3 (Social Media Log Processing)** *As a third example, consider analytics on social media ranging from discovering emerging topics of interest, unusual events related to political unrest or other dangerous indicators, new trends in customers' behaviors, or geographical interest in a certain subject or product. Scalable log processing is critical for running large-scale web sites and offering advanced services. With the current log generation rates in the order of 1-10MB/s per machine, a single data center may collect 10s of TBs of log data per day [41]. For example, social-media networks such as*

*Facebook and Twitter process several TBs of log data per day by pulling continuously generated data from hundreds to thousands of machines, loading them into HDFS, and then running a large variety of data-intensive jobs on a large Hadoop cluster [43]. A representative recurring query that may be run periodically, e.g., every 12 hours or once a day, is to aggregate the log data from the recent past, e.g., last few days, over different dimensions, e.g., age, gender, or country to detect emerging patterns as illustrated in* $Q4$*. Such query may involve expensive join and grouping operations repeatedly applied over high-volume evolving and overlapping data sets.*

```
Q4: SELECT COUNT(connections)
    FROM Users
    WHERE graduation = '2008'
    GROUP BY location
    WINDOW = 6 DAYS, SLIDE = 12 HOURS
    WITHIN = 5 MINUTES
```

Computing the exact results for the above query consumes substantial computational resources over large-scale data sets. However, the relative popularity and overall trend (i.e., profile views rising) are often more important than the exact connection count. Thus, generating approximated results while saving resources and/or budgets paid to service providers is often a desirable goal. In this dissertation, we tackle the problem of strategically choosing an appropriate set of samples to maximize the accuracy of the result within a specified time deadline of the recurring query.

## 1.3 State-Of-The-Art Techniques

We briefly discuss the state-of-the-art and its respective shortcomings in meeting the requirements identified in Chapter 1.4, while a detailed discussion of related work is presented in Chapters 8, 12, and 17.

### 1.3.1 Recurring Query Support on MapReduce

Recently, extensions to Hadoop have been proposed, most notably, Nova [44] and Apache Oozie [45], to address some of the requirements of the above applications. However, they remain far from providing an end-to-end optimized system for supporting multi-granular recurring workloads. Apache Oozie [45] is a workflow scheduler that provides partial support by enabling developers to write scripts for automatic scheduling of jobs. Using this, end-users would no longer need to re-issue the recurring query over and over manually, but instead have it kicked off in an automated fashion. However, Apache Oozie does not provide any system-level optimizations for the executing queries. The Nova [44] system is one step closer to our objective as it offers incremental processing over new batches of data. However, Nova uses the Hadoop platform as a black-box system. It thus falls short in providing any critical system-level optimizations, from offering caching of intermediate data for reuse, cache-aware task scheduling, to adaptive or proactive processing needed to support variations in data selectivities or in input data rates.

Several recent systems have also been proposed to support in-memory processing on top of Hadoop including the M3R [46], SOPA [47], C-MR [48], and In-situ (iMR) [41] systems. In general, these systems focus on changing the disk-based processing inherent in Hadoop into memory-based real-time processing. However, they could fall short to the disk-based recurring queries due to the huge volume of the data source. In this dissertation, we first aim to provide support for recurring queries as first-class citizen in Hadoop.

### 1.3.2 Multi-Query Optimization on MapReduce

Quite often multiple recurring queries are initiated at overlapping time intervals and need to be processed over the same sub-stream of data. In such cases, it is possible that two or more queries need to perform the same processing over the same data. In MapReduce, such queries would be processed as independent jobs, thus resulting in redundant processing. As an example, consider two queries that need to scan the same query log, one trying to identify frequently accessed pages and another performing data mining on the activity of specific IP addresses. To alleviate this shortcoming, queries with similar tasks should be identified and processed together in a batch. In addition to sharing common processing, result sharing is a well-known technique that can be employed to eliminate wasteful processing over the same data. This problem of sharing the queries' execution has been recognized in the context of Hadoop in the recent literature [38, 40, 49]. However, these techniques neither consider the recurring query spectrum, multiple time horizons, nor proactive time slicing to address SLA concerns.

On the other hand, recent efforts in SLA-aware MapReduce techniques [50, 51, 52] either dynamically adjust resource allocation or exploit profiling to help jobs provision resources statically at startup. However, none of these efforts consider sharing among multiple map-reduce jobs. Moreover, they do not address challenges specific to recurring queries, e.g., understanding window semantics, incremental processing, and intermediate data reuse across the consecutive execution of a recurring query. All of these issues will be at the core of the dissertation.

### 1.3.3 Approximate Query Processing on MapReduce

Recurring queries are a typical requirement of applications that involve analysis of massive data sets, as in the case of scientific data. Instead of issuing recurring queries on the complete data set that would entail long waiting times and potentially non-useful results, it would be useful to execute such queries on small representative parts of the data set and return approximate results fast. However, MapReduce does not provide an explicit way to support quick retrieval of indicative results by processing on representative input samples.

Recent extensions to Hadoop, e.g., MapReduce Online [53], EARL [54] and ApproxHadoop [55], have been proposed to provide approximate results for these analytical applications on massive data sets to satisfy the time and resource constraints. EARL [54], utilizing uniform sampling, computes samples until a given level of accuracy is reached. ApproxHadoop [55] exploits input data sampling and task dropping for MapReduce jobs. MapReduce Online [53] focused on implementing online aggregation in MapReduce by producing aggregated results with its corresponding error bound and confidence.

However, none of these efforts target recurring queries. Thus they do not leverage the recurring query semantics to optimize the execution of this important class of queries, e.g., understanding window semantics, incremental sample updates, and progressive result refinement across consecutive executions of a recurring query. This may lead to inefficient sampling, wasting precious computational resources, and poor result accuracy. Hence, we address the imperative need of seamlessly integrating *approximate query processing* with *recurring query execution* in this dissertation.

## 1.4 Research Challenges Addressed in This Dissertation

Various challenges exist in improving the performance of recurring queries on MapReduce as illustrated in Figure 1.2. These challenges include how the evolving data sources are accessed by the recurring query processing system, how does the system effectively avoid redundant processing, does the system provide a way to terminate the execution

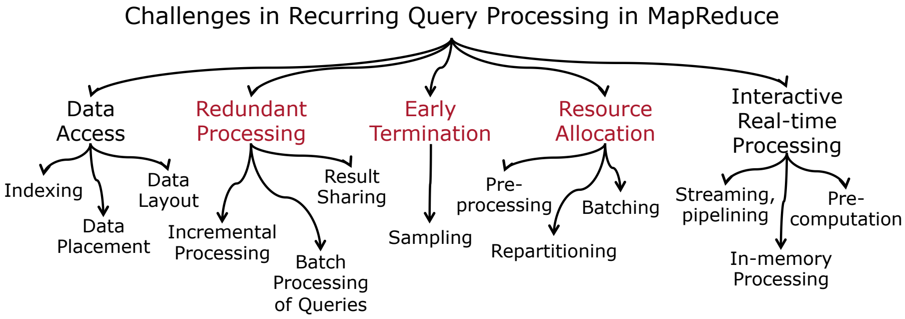


**Figure 1.2:** Challenges of MapReduce Improvements for Recurring Query Processing

early when certain criteria are met, how efficiently the system makes its resource allocation decision at runtime, is the system capable of supporting interactive real-time processing, and more. My dissertation mostly focus on tackling these three big challenges. In this dissertation, we tackle several key challenges in addressing the three research topics. We highlight these challenges as follows.

**Supporting Recurring Queries over Evolving Data Sources.** Recurring queries pose unique challenges because they inherit properties from both continuous queries (in stream processing systems) and ad-hoc queries (in batch processing systems). As motivated by the examples in Chapter 1.2 and in Figure 1.3, recurring queries are similar to continuous queries in that both are long-lived, re-execute periodically over the incoming data, have the notion of sliding windows, and process (possibly) large segments of overlapping data. On the other side of the spectrum, batch-processing systems, e.g., Hadoop [2], are well-designed for scalability and disk-based processing, both are common properties for recurring and ad-hoc batch queries.

However, recurring queries do not always mandate real-time millisecond processing which is the main focus of continuous query processing [56, 57]. Instead, they tend to have a larger granularity of execution, e.g., they may execute once every hour or every day. Also, they may return the results within a certain period of time, e.g., few minutes to a couple of hours. Hence, a query may remain idle for longer periods of time. Batch-

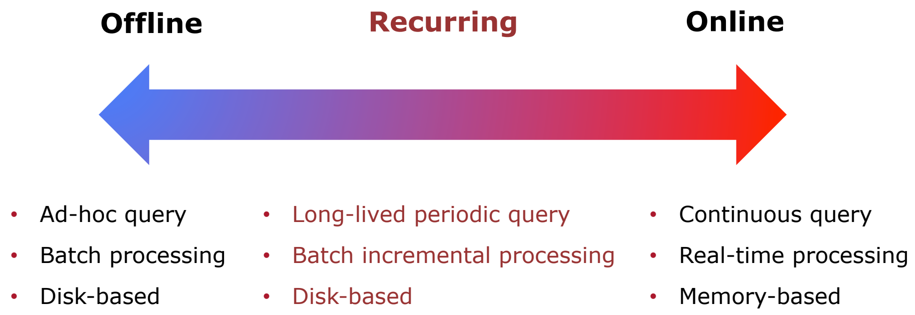


**Figure 1.3:** Large-scale Data Processing Spectrum

processing systems, on the other hand often lack the notion of recurring execution, sliding windows, and overlapping data sets. Hence, they fall short in providing dedicated support that takes advantage of the unique properties from incremental to overlapped processing.

It is apparent that unique optimization opportunities abound, ranging from detecting overlapping computations across consecutive executions to deploying caching techniques. Taking on such opportunities in a single system could reap huge benefits with respect to performance, cost, and computing resources, bringing challenging workloads into the realm of feasibility.

**Sharing the Processing of Multiple Recurring Queries.** Multi-recurring query sharing in MapReduce has been established as an NP-hard problem [40]. Our problem is more challenging due to the following combined characteristics:

• Unmatched Scope of Interest. The scope of interest (window $w$) and execution frequency (slide $s$) of recurring queries may not be well aligned. Hence sharing recurring queries involving identical (or similar) map-reduce jobs may not always be beneficial due to their differences in time and scope granularities.

• Variation in SLA Requirements. Each query has its own SLA parameter. This may prevent queries from fully sharing their execution with others—even if the other constraints are matching. That is, grouping queries without differentiating their SLAs may contradict their respective high-responsiveness requirements.

• Processing Evolving Data Sets. Existing shared execution techniques in MapReduce [38, 40] assume that the inputs to their map-reduce jobs are static. In contrast, recurring queries consume evolving data sets and do not exhibit this convenient property of static inputs. Since the volume of the newly arriving data for a given query can have significant fluctuations, the optimization and sharing techniques are further complicated by having to adapt over time.

**Approximate Recurring Query Processing.** As motivated in Chapter 1.2, there are two main challenges that must be overcome for efficient approximate recurring query processing as summarized below:

• Variances in Recurring Workloads. Under the recurring query model, the underlying data source is continuously evolving with new batches. Thus the sample should be maintained up-to-date with *continuous refinement*. However, rebuilding a new sample from scratch upon the evolving data would incur significant and at times unacceptable overhead, especially when the query recurrence is frequent. A common approach to circumvent this overhead is to incrementally update the current sample. However, variances of the input data sources over time can result in temporary load spikes. Hence it may not always be possible to update samples by the previously used sampling strategy within the query deadline.

• Resource Allocation for Different Sampling Purposes. Given recurring queries have deadlines by which to deliver the results, their approximate executions require a resource allocation strategy to decide how many resources to assign to each execution depending on the query deadline and the evolving data set. These executions may serve different purposes, such as sampling newly arriving data to update the existing samples or sampling certain portion of the existing data sets not yet sampled to refine the existing samples. Decisions in favor of one goal have ramification on the other, potentially forcing us to reconsider execution decisions for the rest of the to-be scheduled executions. As we

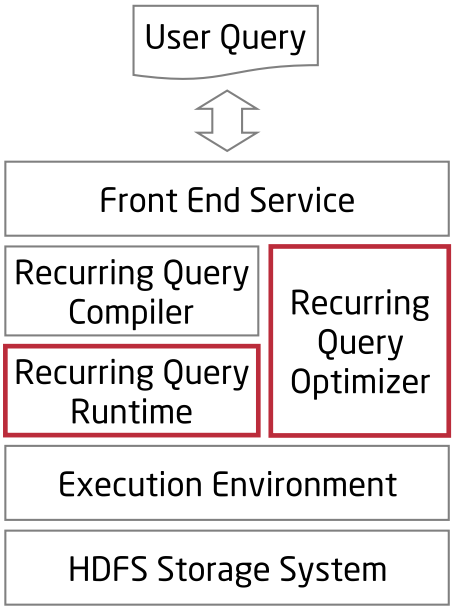


**Figure 1.4:** Recurring Query Processing Architecture

will show, the resource allocation problem in this recurring context is NP-complete and requires efficient heuristics to make appropriate decisions adaptively at run-time.

## 1.5 Proposed Solutions

In this dissertation, we thoroughly investigate solutions to address the challenges listed in Chapter 1.4 in the context of recurring query processing. We extend the state-of-the-art infrastructure to meet the needs derived from our motivating examples. As depicted in Figure 1.4, our main focus lies on the recurring query optimization and runtime within the recurring query processing architecture. The main contributions of this dissertation include the following.

**Supporting Recurring Queries in Hadoop.** We propose a new system called *Redoop* that is designed to support recurring queries. Redoop extends the Hadoop platform to exploit the optimization opportunities from recurring queries. This part of the dissertation work contributes to research in the recurring query processing in the following ways:

1. We establish the recurring query model to cover a wide spectrum of execution granularities. In particular, it is specified by a window size and execution frequency. Redoop is designed to efficiently handle recurring queries through a best-effort proactive execution mechanism, where it adaptively detects fluctuations in the data rate between different executions and proactively starts performing partial processing to deliver results.

2. We design adaptive window-aware partitioning techniques for splitting the input data into fine-grained data units (called panes) customized for effective window-centric data consumption. The adaptive partitioning reduces or even eliminates costs of the repeated reading and loading of partially overlapping panes across windows.

3. We provide techniques to cache the intermediate data at different stages of a MapReduce job and to create re-use opportunities across the subsequent execution of recurring queries. The caching mechanism significantly reduces I/O costs by avoiding unnecessary re-loading, re-shuffling, and re-computation of the overlapping data.

4. We propose an advanced window-aware task scheduler that exploits cache locality and resource usage in the system. This scheduler is tuned to maximize the utilization of the available caches and to balance the workload on each node to boost the system's performance.

5. Experimental Evaluation: We evaluate Redoop using real-world data sets on a variety of recurring workloads. Redoop outperforms Hadoop in all cases by a factor of up to 9 on average.

**Shared Execution of Recurring Workloads.** We propose a new system called *Helix*, the first MapReduce-based infrastructure for shared execution of recurring workloads under SLA-constraints. Given a workload composed of recurring queries and their associated window and SLA constraints, Helix constructs a global shared execution plan over the evolving data sources. Helix first deploys sliced window-alignment techniques to discover sharing opportunities among the recurring queries. It then models the problem as the stochastic knapsack problem with uncertain weights. Helix divides the optimization problem into two interleaved phases: (1) Creating a potential sharing plan that divides the queries into groups, and (2) Computing an execution scheduling for the groups within the given sharing plan. Helix iterates over these two phases and prunes the sub-optimal solutions as early as possible until it reaches an optimized shared execution plan for all recurring queries in one pass. This approach enables Helix to maximize the overall SLA satisfaction of the given recurring queries, while concurrently reducing the resources consumed due to shared execution. Our contributions in the area of multi-recurring query optimization thus are:

1. We formulate the problem of optimizing multiple recurring queries in MapReduce. We incorporate the queries' properties, e.g., window semantics, and SLA constraints, into the interleaved sharing and scheduling algorithms.

2. We propose techniques for solving the unmatched scope of interest problem over evolving data sources. We introduce the sliced window alignment strategy for preprocessing and partitioning the data into smaller segments. These techniques not only align queries for better sharing, but also reduce costs associated with the repeated loading costs among overlapping windows.

3. We develop an SLA-driven optimizer that generates an execution plan for a given set of recurring queries which maximizes the overall SLA satisfaction. The opti-

mizer exploits a Branch-and-Bound search strategy with various pruning strategies that effectively prune sub-optimal solutions as early as possible from the exponential search space, rendering the search tractable in practice.

4. We build the Helix prototype engine on top of the open-source Hadoop platform. Our experimental study using real-world datasets demonstrates that Helix consistently outperforms state-of-the-art techniques. In many cases, Helix achieves an order of magnitude improvement in satisfying the SLAs by leveraging recurrence specific sharing decisions.

**Approximate Recurring Queries Processing.** We propose a new system called *Faro* to provide fast approximate recurring query processing. Given a workload composed of recurring queries and their associated window constraints, Faro maximizes query precision by dynamically and adaptively making the best use of available resources within the given deadline specification. Faro collects offline execution statistics about the input data by piggybacking their computation with other tasks. Based on the statistics, Faro's deadline-bound sampling strategy aims to maximize result accuracy for query executions while best meeting the deadline requirements of recurring queries. Faro further adaptively allocates resources to approximation-based tasks to update existing samples and to progressively refine the result accuracy. Our key contributions include:

1. We formulate the problem of optimizing approximate recurring query processing in MapReduce. We incorporate the queries' properties, e.g., window semantics and time-based deadlines, into the adaptive approximation problem formulation.

2. We propose a deadline-bound sampling strategy for approximate recurring query processing. The proposed technique integrates equi-depth partitioning and multistage sampling to compute approximate results with error bounds. It not only pro-

vides maximum result accuracy for each individual recurring query execution, but also reduces the repeated sampling costs incurred by overlapping windows.

3. We develop new adaptive resource allocation mechanisms that progressively improve the existing samples when recurring queries have overlaps of inputs between consecutive executions. These mechanisms offer remarkable flexibility for choosing appropriate executions to refinement query results while saving computation resources for rebuilding samples without sacrificing result accuracy.

4. We build the Faro system on top of the open-source Hadoop platform. Our experimental evaluation using real-world data sets demonstrates that Faro consistently outperforms the state-of-the-art techniques by 14 fold. In many cases, Faro not only speeds up the execution, but also improves the results' accuracy due to the efficient utilization of resources.

## 1.6 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 first provides the background and preliminary materials needed for this dissertation. We then discuss in detail the three research topics of this dissertation, namely *supporting recurring queries in Hadoop*, *shared execution of recurring workloads*, and *processing of approximate recurring queries*, in Part I (Chapters 3-8), Part II (Chapters 9-12), and Part III (Chapters 13-17) respectively. The discussion of each of the three research topics includes the problem formulation and analysis, description of the proposed solution, experimental evaluation, and lastly a discussion of related work. Chapter 18 concludes this dissertation and Chapter 19 discusses possible future work.

# 2

# Preliminary

## 2.1 MapReduce Basics

**MapReduce** [1] is a framework for parallel processing of massive data sets. A job to be performed using the MapReduce framework has to be specified as two phases: the map phase as specified by a Map function (also called mapper) takes key/value pairs as input, possibly performs some computation on this input, and produces intermediate results in the form of key/value pairs; and the reduce phase which processes these results as specified by a Reduce function (also called reducer). The data from the map phase are shuffled, i.e., exchanged and merge-sorted, to the machines performing the reduce phase. It should be noted that the shuffle phase can itself be more time-consuming than the two others depending on network bandwidth availability and other resources.

In more detail, the data are processed through the following 6 steps as illustrated in Figure 2.1:

1. Input reader: The input reader in the basic form takes input from files (large blocks) and converts them to key/value pairs. It is possible to add support for other input types, so that input data can be retrieved from a database or even from main mem-

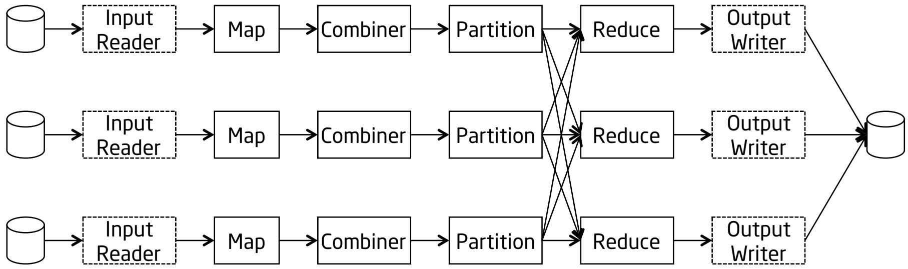


**Figure 2.1:** MapReduce Dataflow

ory. The data are divided into splits, which are the unit of data processed by a map task. A typical split size is the size of a block, which for example in HDFS is 64 MB by default, but this is configurable.

2. Map function: A map task takes as input a key/value pair from the input reader, performs the logic of the Map function on it, and outputs the result as a new key/value pair. The results from a map task are initially output to a main memory buffer, and when almost full spill to disk. The spill files are in the end merged into one sorted file.

3. Combiner function: This optional function is provided for the common case when there is (a) significant repetition in the intermediate keys produced by each map task, and (b) the user-specified Reduce function is commutative and associative. In this case, a Combiner function will perform partial reduction so that pairs with same key will be processed as one group by a reduce task.

4. Partition function: As default, a hashing function is used to partition the intermediate keys output from the map tasks to reduce tasks. While this in general provides good balancing, in some cases it is still useful to employ other partitioning functions, and this can be done by providing a user-defined Partition function.

5. Reduce function: The Reduce function is invoked once for each distinct key and

is applied on the set of associated values for that key, i.e., the pairs with same key will be processed as one group. The input to each reduce task is guaranteed to be processed in increasing key order. It is possible to provide a user-specified comparison function to be used during the sort process.

6. Output writer: The output writer is responsible for writing the output to stable storage. In the basic case, this is to a file, however, the function can be modified so that data can be stored in, e.g., a database.

As can be noted, for a particular job, only a Map function is strictly needed, although for most jobs a Reduce function is also used. The need for providing an Input reader and Output writer depends on data source and destination, while the need for Combiner and Partition functions depends on data distribution.

Hadoop [2] is an open-source implementation of MapReduce, and without doubt, the most popular MapReduce variant currently in use in an increasing number of prominent companies with large user bases, including companies such as Yahoo! and Facebook.

Hadoop consists of two main parts: the Hadoop distributed file system (HDFS) and MapReduce for distributed processing. As illustrated in Figure 2.2, Hadoop consists of a number of different daemons/servers: NameNode, DataNode, and Secondary NameNode for managing HDFS, and JobTracker and TaskTracker for performing MapReduce.

**HDFS** is designed and optimized for storing very large files and with a streaming access pattern. Since it is expected to run on commodity hardware, it is designed to take into account and handle failures on individual machines. HDFS is normally not the primary storage of the data. Rather, in a typical workflow, data are copied over to HDFS for the purpose of performing MapReduce, and the results then copied out from HDFS. Since HDFS is optimized for streaming access of large files, random access to parts of files is significantly more expensive than sequential access, and there is also no support

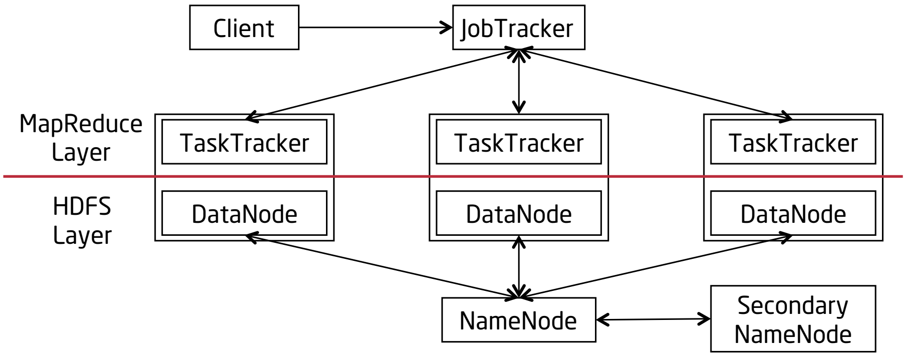


**Figure 2.2:** Hadoop Architecture

for updating files, only append is possible. The typical scenario of applications using HDFS follows a write-once read-many access model.

Files in HDFS are split into a number of large blocks (usually a multiple of 64 MB) which are stored on DataNodes. A file is typically distributed over a number of DataNodes in order to facilitate high bandwidth and parallel processing. In order to improve reliability, data blocks in HDFS are replicated and stored on three machines, with one of the replicas in a different rack for increasing availability further. The maintenance of file metadata is handled by a separate NameNode. Such metadata includes mapping from file to block, and location (DataNode) of block. The NameNode periodically communicates its metadata to a Secondary NameNode which can be configured to do the task of the NameNode in case of the latter's failure.

**MapReduce Engine.** In Hadoop, the JobTracker is the access point for clients. The duty of the JobTracker is to ensure fair and efficient scheduling of incoming MapReduce jobs, and assign the tasks to the TaskTrackers which are responsible for execution. A TaskTracker can run a number of tasks depending on available resources (for example two map tasks and two reduce tasks) and will be allocated a new task by the JobTracker when ready. The relatively small size of each task compared to the large number of tasks in total helps to ensure load balancing among the machines. It should be noted that while the number of map tasks to be performed is based on the input size (number of splits), the number of reduce tasks for a particular job is user-specified.

In a large cluster, machine failures are expected to occur frequently, and in order to handle this, regular heartbeat messages are sent from TaskTrackers to the JobTracker periodically, and from the map and reduce tasks to the TaskTracker. In this way, failures can be detected, and the JobTracker can reschedule the failed task to another TaskTracker. Hadoop follows a speculative execution model for handling failures. Instead of fixing a failed or slow-running task, it executes a new equivalent task as backup. Failure of the JobTracker itself cannot be handled automatically, but the probability of failure of one particular machine is low so that this should not present a problem in general.

**The Hadoop Ecosystem.** In addition to the main components of Hadoop, the Hadoop ecosystem also contains other libraries and systems. The most important in our context are HBase, Hive, and Pig. HBase [58] is a distributed column-oriented store, inspired by Google's Bigtable [59] that runs on top of HDFS. Tables in HBase can be used as input or output for MapReduce jobs, which is especially useful for random read/write access. Hive [60, 61] is a data warehousing infrastructure built on top of Hadoop. Queries are expressed in an SQL-like language called HiveQL, and the queries are translated and executed as MapReduce jobs. Pig [5] is a framework consisting of the Pig Latin language and its execution environment. Pig Latin is a procedural scripting language making it possible to express data workflows on a higher level than a MapReduce job.

## 2.2 MapReduce Sharing Techniques

We now briefly review sharing techniques among ad-hoc queries in MapReduce and discuss the associated key observations [38, 40]. For simplicity, we limit our example below to two jobs $J_1$ and $J_2$. However, the sharing principles are generally applicable across $n$

queries. In Chapter 10, we will explain why these techniques are not directly applicable to recurring workloads without customized optimizations.

**Sharing Map Input Scans.** For two jobs $J_1$ and $J_2$ to share their map input scan, typically the input files, the input key, and the value types of $J_1$ and $J_2$ must all be the same in MapReduce settings. This allows $J_1$ and $J_2$ to be combined into one integrated job that shares the map input scan for the two jobs. To distinguish between the map outputs for such two jobs, we attach tags to the map outputs $M_1$ and $M_2$ respectively. In the reduce phase, the key/value pairs are pushed to appropriate reduce functions according to their attached tags. When all values associated with a key have been consumed, we generate the results for the jobs associated with that key. In this scenario, the savings result from scanning and parsing the map input only once.

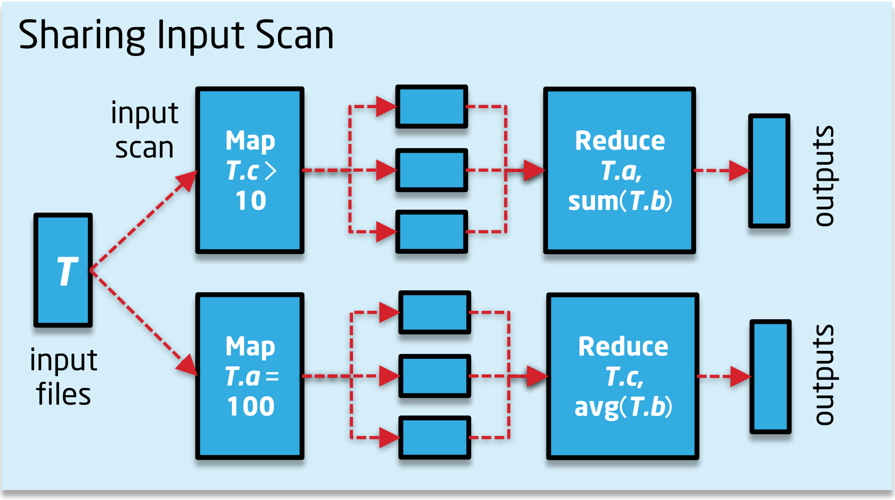


**Figure 2.3:** Share Input Scan

**Example 2.1** *Consider an input table $T(a, b, c)$, and the following queries:*

```
J1: SELECT   T.a, sum(T.b)
    FROM     T
    WHERE    T.c > 10
    GROUP BY T.a

J2: SELECT   T.c, avg(T.b)
    FROM     T
    WHERE    T.a = 100
    GROUP BY T.c
```

*The shared scan conditions are met. In this scenario, the savings result from scanning and parsing the input only once. Clearly, this sharing scheme can be easily extended to multiple jobs. Note that there is no sharing at the reduce stage. After grouping at the reduce side, each tuple is pushed to the appropriate reduce function. The size of the intermediate data processed is the same as in the case of two different jobs, with a minor overhead that comes from the tags.*

**Sharing Map Outputs.** Assume that in addition to sharing map input scans, the map output keys $K_1$ and $K_2$ are the same for both jobs $J_1$ and $J_2$. In that case, the map outputs for $J_1$ and $J_2$ can also be shared. Here map functions $map_1$ and $map_2$ are applied to each input tuple. Then the map output tuples produced by $map_1$ are tagged with $tag(1)$ only. If a map output tuple was produced from an input tuple by both $map_1$ and $map_1$, it is tagged by $tag(1, 2)$. In the reduce phase, tuples in each group are distributed to the appropriate reduce function according to their tags. For example, if the tag of the value is $tag(1, 2)$, we distribute the same value to both reduce functions of $J_1$ and $J_2$ separately. In this scenario, sharing map outputs reduces the total size of the map outputs and hence the I/O sorting costs and communication costs. At the reduce side, each group contains tuples belonging to both jobs, with each tuple possibly belonging to one or both jobs. The reduce stage needs to dispatch the tuples and push them to the appropriate reduce function, based on tag.

**Example 2.2** *Consider an input table $T(a, b, c)$, and the following queries:*

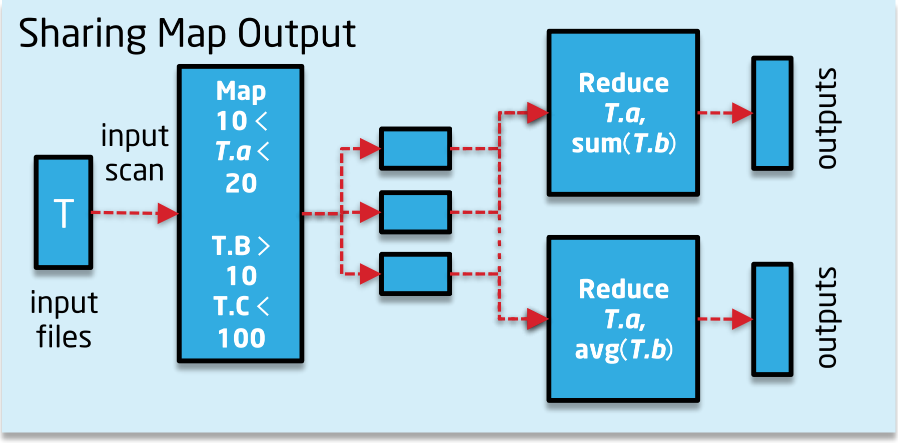


**Figure 2.4:** Share Map Output

```
J1: SELECT   T.a, sum(T.b)
    FROM     T
    WHERE    T.a > 10 AND T.a < 20
    GROUP BY T.a

J2: SELECT   T.a, avg(T.b)
    FROM     T
    WHERE    T.b > 10 AND T.c < 100
    GROUP BY T.a
```

*The map functions are not the same. However, the filtering can produce overlapping sets of tuples. The map output key (T.a) and value (T.b) types are the same. Hence, we can share the overlapping parts of map output. Producing a smaller map output results to savings on sorting and copying intermediate data over the network. This mechanism can be easily generalized to more than two jobs.*

**Sharing Map Functions.** Sometimes the map functions are identical and thus the map function can be executed only once. At the end of the map stage two streams are produced, each tagged with its job tag. If the map output is shared, then only one stream needs to be generated. Even if only some filters are common across both jobs, then it is possible to share parts of the map functions. Sharing parts of map functions involves identifying common subexpressions and filter reordering, both known to be hard problems.

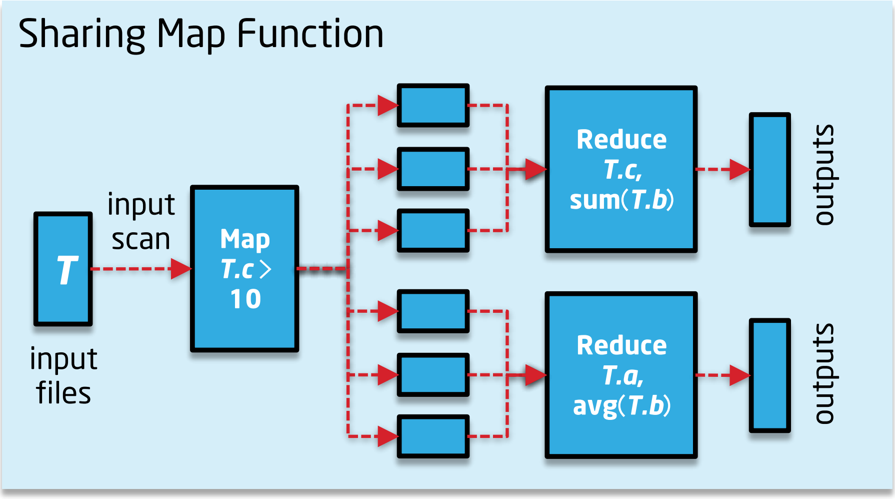


**Figure 2.5:** Share Map Function

**Example 2.3 (Aggregation)** *Consider an input table $T(a, b, c)$, and the following queries:*

```
J1: SELECT    T.c, sum(T.b)
    FROM      T
    WHERE     T.c > 10
    GROUP BY  T.c

J2: SELECT    T.a, avg(T.b)
    FROM      T
    WHERE     T.c > 10
    GROUP BY  T.a
```

*The map pipelines are identical. Note that in this case, by identical map pipelines we mean the parsing and the set of filers/transformations in the map function - the map output* *key and value types are not necessarily the same. If, additionally, the map output key and value types are the same, we can apply map output sharing as well. In our example, assuming that the second query groups by T.c instead of T.a, we would have the reducing pipeline similar to the previous example.*

**Example 2.4 (Partial Map - Aggregation)** *Consider an input table $T(a, b, c)$, and the following queries:*

```
J1: SELECT   T.a, sum(T.b)
    FROM     T
    WHERE    T.c > 10 AND T.a < 20
    GROUP BY T.a

J2: SELECT   T.a, avg(T.b)
    FROM     T
    WHERE    T.c > 10 AND T.c < 100
    GROUP BY T.a
```

*In this case, the map pipelines are not the same. However, some of their filters overlap. Also in this case, the key and value types of map output tuples are the same, and we can apply map output sharing. We remark that sharing parts of map functions has many implications. It involves identifying common subexpressions and filter reordering, which are hard problems.*

**Sharing Reduce Inputs.** This technique requires that the map output keys $K_1$ and $K_2$ are identical for both jobs $J_1$ and $J_2$. The key idea is to materialize the reduce input in the reduce phase of a job $J_1$ so that subsequently it can be reused also by the reduce phase of $J_2$. In this way, the sorting and communication costs required for processing the reduce input are eliminated when processing $J_2$. Reduce inputs can be shared by a sequence of executions of one single recurring query. In this case, the amount of savings is determined by the overlapping data across multiple consecutive executions.

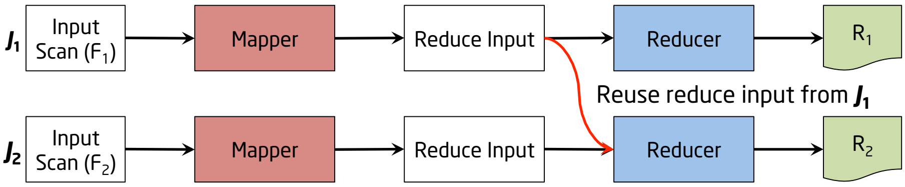


**Figure 2.6:** Share Reduce Inputs

## 2.3 MapReduce Sampling Techniques

Next, we review the two commonly used sampling techniques in MapReduce, namely *pre-map* and *post-map* sampling [54]. Each of these techniques has its own strengths and weaknesses as discussed next.

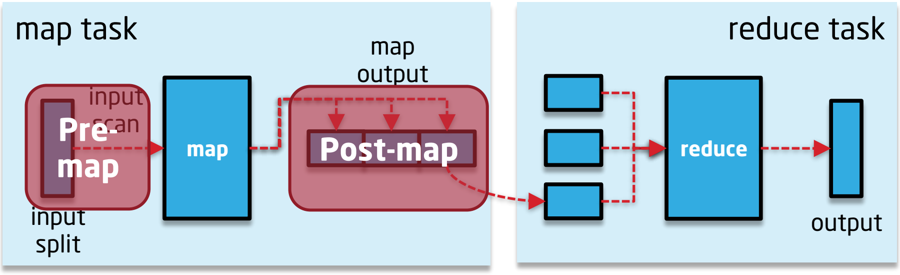


**Figure 2.7:** Sampling Approaches

In HDFS, an input file is divided into a set of blocks with each block being 64MB by default. When running a MapReduce job, these blocks can be further subdivided into Input Splits used as input to the mappers. Given such an architecture, sampling can be done before or after sending the input to the Mapper, referred to as pre-map and post-map sampling, respectively.

***Pre-map sampling*** (i.e., block-level) works by sampling a subset of blocks from the initial input file and then passing these blocks into the mappers. Because sampling is done prior to the data loading stage, the response time is greatly improved, with the potential downside of a slightly less accurate result. The reason for this is that tuples within one block may at times exhibit a correlation if data is clustered by the sampling attribute. However, it has been shown that block-level sampling may produce more accurate estimates than tuple-level sampling with the same costs [62]. In practice, the estimate of the number of the key value $\langle k, v \rangle$ pairs produced by the pre-map sampling approach is acceptable. Nevertheless, as explained below, the user has the flexibility to use post-map sampling if a more accurate final result is desired and a longer query execution can be tolerated by the query deadline. Algorithm 1 presents the HDFS sampling algorithm used in pre-map sampling.

**Algorithm 1** Pre-map Sampling

```
Input: $start$, $end$, $sample$
$start \leftarrow split.getStart()$
$end \leftarrow start + split.getLength()$
$sample \leftarrow \phi$
while $|sample| < n$ do
    $start \leftarrow$ pick a random start position
    if $start! =$ beginning of a line then
        $skipFirstLine \leftarrow$ true
        fileIn.seek($start$)
    $in$ = new LineReader($fileIn, job$)
    if $skipFirstLine$ then
        $start+ = in$.readLine($end - start$)
    $sample \leftarrow$ includeLineInSample()
    $skipFirstLine \leftarrow$ false
```

***Post-map sampling*** (i.e., tuple-level) works by reading and parsing the data file and then sending the selected $\langle k, v \rangle$ pairs to reducers. Each $\langle k, v \rangle$ pair is selected using random hashing that generates a pre-determined set of keys. All $\langle k, v \rangle$ pairs are stored on the mappers locally. When all data has been received, a subset of $\langle k, v \rangle$ pairs are randomly picked that satisfy the sample size and are sent to the reducer. Post-map sampling provides random access to the data in the original input file. This is beneficial because the majority of statistical estimates are based on the uniform random sample of tuples in the data. However, post-map sampling is not efficient since data is organized in blocks on HDFS. Retrieving a simple random sample of size $s$ for one mapper would cause the transmission of $s$ blocks in the network in the worst case. Namely, one tuple is sampled out of each block and none of $s$ blocks is stored locally on this particular mapper. Hence, post-map sampling is useful for application relying on an accurate estimate of the total $\langle k, v\rangle$ pairs at the expense of the increased execution costs. Post-map sampling is shown in Algorithm 2.

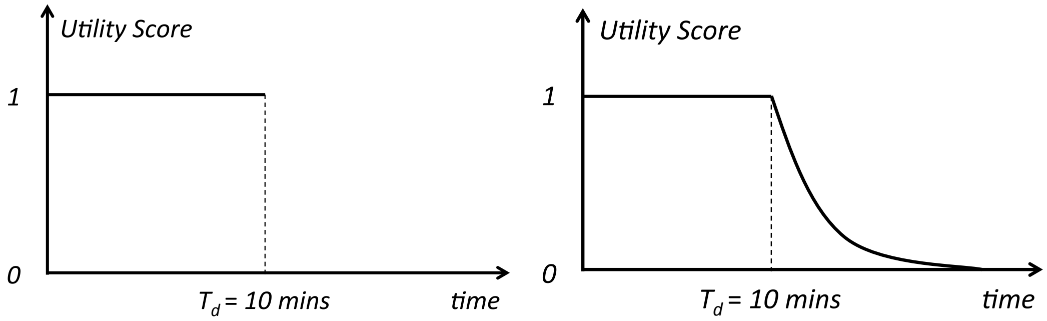
**Algorithm 2** Post-map Sampling

```
Input: $hash$, $timestamp$
$hash \leftarrow$ initialize the hash
$timestamp \leftarrow$ initialize the timestamp
while $input! =$ null do
    $key \leftarrow$ random key for input
    $value \leftarrow$ value for input
    $hash[key] \leftarrow value$
sendSample(hash(rand()%hash.size))
while true do
    if get_new_error_avg(timestamp) $>$ required then
        sendSample(hash(rand()% hash.size))
    else
        return
```

In summary, both sampling techniques have their respective advantages in MapReduce. We thus propose to exploit both techniques for approximate recurring query processing using a cost-driven approach (see Chapter 14).

## 2.4 Recurring Query Model

***Query Parameters.*** Recurring queries [63, 64] execute periodically over evolving diskresident data sets, i.e., data sets stored in HDFS. In each execution, a recurring query $\mathcal{Q}(\mathcal{W}, \mathcal{S}, \mathcal{D})$ applies its computations over a bounded subset of evolving data sets as determined by three configuration parameters window $\mathcal{W}$, slide $\mathcal{S}$, and deadline $\mathcal{D}$. The window $\mathcal{W}$ specifies the scope of data to process, while the slide $\mathcal{S}$ specifies the frequency of execution. For example, $\mathcal{Q}$ ($\mathcal{W}$ = 12 hours, $\mathcal{S}$ = 1 hour) specifies a recurring query that executes every hour and each time processes all data within the last 12 hours. The deadline $\mathcal{D}$ specifies a time limit for a query's execution to complete. *Deadline-bound recurring queries* are common in advertisement systems and web search engines [65, 66], where the query is spawned on a large data set and accuracy is proportional to the fraction of data processed [67, 68, 69]. Often an SLA parameter is associated with the deadline $\mathcal{D}$ as a time-based function $\vartheta(t)$. The SLA parameter indicates the merit of the query's results (a utility score) if delivered after the deadline. Figure 2.8 shows two sample SLAs. The function in Figure 2.8(a) indicates that the results produced after the deadline $T_d$ are useless to the application, i.e., its utility becomes zero. Figure 2.8(b) shows an SLA function that decreases the result's utility monotonically as the query execution exceeds $T_d$ = 10 minutes. In general, we support any non-increasing utility function as an SLA for recurring queries.



(a) SLA Function 1

(b) SLA Function 2

**Figure 2.8:** SLA Function

***Input Timestamps.*** Recurring queries process data that is updated in batches over time. Thus the timestamps associated with the data are important in our model. A data batch $f_i$ received at time $T_i$ is annotated by the time $T_i$. Time ranges covered by different batch files do not overlap. That is, the time ranges covered by the tuples in files $f_1$, $f_2$, ..., $f_n$, are in the range of $[T_0, T_1)$, $[T_1, T_2)$, ... , $[T_{n-1}, T_n)$ with $T_i < T_{i+1}$. Therefore, there is an order among the files, but no order constraints among the tuples within each file must hold. The above model is common in data analytics applications. For example, in log processing, the system may collect the log files every other hour from multiple machines, merge them without sorting, and upload the file into HDFS as a new batch. Between two consecutive executions $E_i$ and $E_j$ at times $T_i$ and $T_j$, where $T_i < T_j$, the system may receive multiple batches of data in the form of HDFS files, say $f_1$, $f_2$, ..., $f_n$ at times $T_1$, $T_2$, ..., $T_n$, where $T_i < T_1 < T_2 < ... < T_n < T_j$.

***Execution Model.*** The Redoop system [64] is proposed to treat recurring queries as first-class citizen in the MapReduce infrastructure. A recurring query $\mathcal{Q}(\mathcal{W}, \mathcal{S}, \mathcal{D})$ is registered in Redoop, where the $\mathcal{W}$ and $\mathcal{S}$ properties are defined as configuration parameters. Once registered, Redoop periodically triggers the execution of $\mathcal{Q}$ according to its $\mathcal{W}$ and $\mathcal{S}$ parameters. The evolving inputs of $\mathcal{Q}$ are consumed from a specific HDFS directory, while the outputs are also periodically produced to another HDFS directory. Given a recurring query $\mathcal{Q}(\mathcal{W}, \mathcal{S}, \mathcal{D})$, Redoop pre-processes the input data and subdivides the input files into smaller segments, called panes, with a refined granularity. The goal is that when the window of interest $\mathcal{W}$ slides by $\mathcal{S}$, any overlapping data segments between the two consecutive executions do not need to be processed again. Therefore, the pane-based partitioning divides a single query execution into a sequence of map-reduce jobs over non-overlapping pane inputs, each producing partial results. These partial results are then combined—using a user-defined function—to generate the final desired results. In order to do so, we assume that such recurring queries (e.g., aggregation and SPJ queries) are composable and can be incrementally computed. This execution strategy reduces the unnecessary I/O and CPU costs otherwise associated with repeated work across overlapping windows. Redoop also offers inter-window caching mechanisms that cache reduce input and output data for subsequent reuse. The cached data reduces redundant disk I/O operations. Although Redoop enables several unique optimizations to recurring queries, it is limited to processing the recurring queries in isolation, i.e., no sharing, and it also does not support SLA specifications. Our proposed Helix system further overcomes the limitations of Redoop by enabling the sharing among multiple recurring queries and specifying the queries' SLAs.

Both Redoop and Helix are limited to consuming all date tuples in the current window for each execution, i.e., no approximate results are produced based on sampled data, which may cause delay problems for many applications. Our proposed Faro system overcomes these limitations by enabling approximate processing over sampled data to satisfy the queries' deadline $\mathcal{D}$. Namely, the query execution in Faro should strive to maximize the accuracy of its result within a specified time of each execution.

# Part I

# Supporting Recurring Queries in Hadoop

# 3

# Redoop System Overview

Figure 3.1 illustrates the proposed architecture of Redoop as an extension of Hadoop. The sliding window semantics embedded in recurring queries can result in a significant overlap of data between consecutive windows [70, 71, 72]. Thus, we have designed an advanced task execution manager for Redoop to cache input data on local file systems of task nodes. The cached data is efficiently utilized to reduce redundant disk I/O operations at run-time. Redoop introduces an incremental processing model to allow task nodes to asynchronously execute any map or reduce task with incrementally evolving data between two query recurrences. Beyond the map/reduce task structure of Hadoop, Redoop adds four new components (depicted by the white boxes) along with adopting and extending several existing components from Hadoop (the light-gray boxes) in Figure 3.1.

**1. Window Semantic Analyzer** is the optimizer that, given the window constraints embedded in recurring queries, produces a data partition plan. That is, it produces a plan of subdividing input data sources into panes (i.e., separate HDFS files) with optimized granularity that can be most efficiently processed by map and reduce tasks. Such plan can also eliminate any unnecessary data re-processing caused by recurring queries (Chapter 4.1).

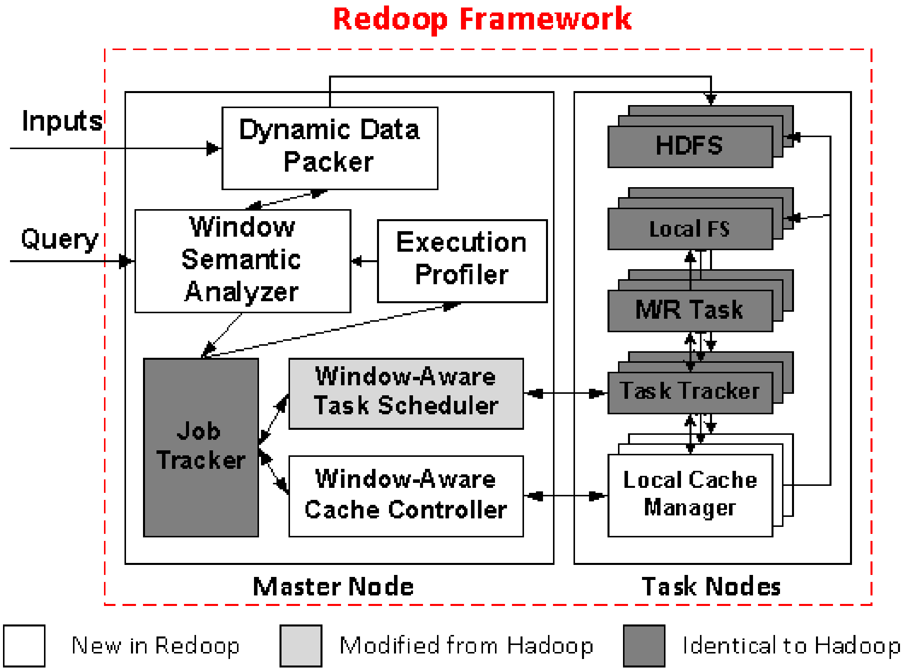


**Figure 3.1:** Redoop System Architecture

**2. Dynamic Data Packer** is the partition executor that implements the instructions encoded in the partition plan produced by the above optimizer. That is, it dynamically splits very large input data partitions into smaller panes (Chapter 4.2). The data packer piggybacks the pane creation step with the loading step, i.e., while a given input file is being loaded into HDFS, the data packer partitions the records to the corresponding panes.

**3. Execution Profiler** collects the statistics after the completion of each query recurrence, i.e., execution times of previous query recurrences. The profiler then transmits the statistics to the Window Semantic Analyzer such that the pane size can be adjusted in a timely manner during the subsequent input partitioning. The Window Semantic Analyzer, Dynamic Data Packer, and Execution Profiler together also determine Redoop's execution modes to tackle data fluctuations (Chapter 4.3).

**4. Local Cache Manager**, installed on each task node in Redoop, maintains the Redoop caches on the node's respective local file system. The Local Cache Manager sends its cache meta-data to the Window-Aware Cache Controller described below along with its heartbeat for global synchronization. The cache manager allows users to provide a purge policy and is responsible for purging the expired caches according to the prescribed policy and the purge notification received from the master node (Chapter 5.1).

**5. Window-Aware Cache Controller** is a new module housed on the Redoop master node that maintains window-aware meta-data of reduce input and output data cached on any of the task nodes' local file systems. This controller helps optimize query execution by providing information of window-dependent cache usage for run-time task scheduling decisions (Chapter 5.2).

**6. Window-Aware Task Scheduler**, an extension of the default Hadoop TaskScheduler, fully exploits the intermediate caches that reside on the local file system for incremental window-centric processing of input data. It also balances the workloads on each node based on the locality of prior caches. Exploiting existing caches and keeping the load balanced further improve the query processing performance (Chapter 5.3).

# 4

# Redoop Input Partitioning

Supporting recurring queries requires Redoop to understand the general notion of window semantics in recurring queries. This section first introduces the Window Semantic Analyzer for recurring queries used in Redoop, and then illustrates Redoop's dynamic data packer for input data pre-processing. Furthermore, load variances in evolving data over time require Redoop to adapt to these changes. Variance of the input data sources (in rate and/or in values) can at times result in temporary load spikes, with the data processing time significantly affected by the duration of the spikes. Worse yet, the cluster resources may not be efficiently utilized and the delayed query results may further slowdown other data analytics jobs that depend on the current query execution. To tackle such temporary load variances, an adaptive strategy is devised during the input partitioning.

## 4.1 Window Semantic Analyzer

The Window Semantic Analyzer takes as input a sequence of recurring queries with different window constraints. Its goal is to find an efficient strategy for partitioning the input data in a window-aware fashion to enhance the overall system's performance and to minimize any redundant processing or I/O operations. The partitioning strategy created by the Window Semantic Analyzer will be executed by the Dynamic Data Packer component. The advantage of partitioning the data into smaller panes [70] is that it gives the system the flexibility to create optimization opportunities between the overlapping data across consecutive windows, e.g., Redoop processes and shuffles each pane only once, caches the results on the local disks of the data nodes, and re-uses them repeatedly as needed based on the window semantics.

Next, we highlight the key challenges related to data partitioning.

**1. Overlapping Data Re-computation.** For consecutive executions of a recurring query, the plain Hadoop would re-load the overlapping data partitions from HDFS multiple times. And it is not only about re-loading, but the processing, shuffling, and sorting phases will be all repeated. These operations are very expensive and would consume significant system's resources.

**2. Redundant Data Loading.** Although smart caching would solve the problem highlighted above, it may not be sufficient if the cached data are large and not well-aligned with the window boundaries. In this case, unnecessary I/O operations may be inevitable. For example, assume a recurring query with $win$ = 4 hours and $slide$ = 3 hours, and the partitioning of data is performed based on its slide size, i.e., 3 hours chunks, and these partitions are cached on the local file system for future use. Then, in order to produce a correct output, the system has to retrieve the cached partition and then combine it with the newly arrived batch (which is 6-hour data in total). This is inefficient because only 1/3 of the cached partition is necessary in the second window. Therefore, partitioning based on the slide size is not always the best choice, and more dynamic partitioning is needed based on the available queries in the system.

Next, we discuss the Window Semantics Analyzer that tackles the second challenge, while Chapter 5 discusses our solution to the first challenge. The Window Analyzer takes the queries, the execution statistics from the Execution Profiler, and the HDFS block size (default 64MB) in the Hadoop configuration as input, and produces a partition strategy as its output, also called the *partition plan*. Algorithm 8 illustrates the strategy that we use to generate the partition plan. The key idea of the algorithm is to slice the window states into fine-grained disjoint panes based on the window constraints of individual data sources. This way the Redoop system executes window-centric operations over those panes in a finer-grained fashion.

In the algorithm, we use the greatest common divider (GCD) function to determine the logical data unit, which is henceforth referred to as $pane$ (Line 1). Given the logical $pane$, $fileSize$ is the expected size of the physical file that is to store the $pane$, incorporating the actual arrival rate of the corresponding data source (Line 2). Lines 3-8 choose the more effective method of representing the $pane$, considering the following two cases.

**1. Oversize Case**: One $pane$ corresponds to exactly one physical file (Line 4). And this file may have one or more splits (i.e., 64 MB chunks) on HDFS.

**2. Undersized Case**: Multiple $pane$s together correspond to one file (Line 7). Namely, one file contains multiple logical panes when the input data rate is not intensive.

Having such optimization on mapping logical data units to physical files, the Dynamic Data Packer can avoid creating many small files in Hadoop.

**Algorithm 3** Input Data Source Partitioning Algorithm

```
Input: Query $Q$, Data Source Statistics $S$, blockSize
Output: Partition Plan $PP$
1: $pane \leftarrow$ GCD($Q.win, Q.slide$)
2: $fileSize \leftarrow S.rate \times pane$
3: if $fileSize \geq blockSize$ then
4:     $PP \leftarrow$ ($pane$, 1, 1) // one file for one pane
5: else
6:     $paneNum \leftarrow \lfloor blockSize/fileSize \rfloor$
7:     $PP \leftarrow$ ($pane$, 1, $paneNum$) // one file for multiple panes
8: return $PP$
```

For example, a recurring query with its window constraints $win$ = 6 minutes and $slide$ = 2 minutes. Then the logical pane size is 2 minutes as a result of GCD (6, 2). Now consider the input rate of data source $News$ is 16MB/minute and the HDFS block size is default 64MB. In this case, the partition plan for $News$ is depicted in Figure 4.1, with each file denoted by the same color.

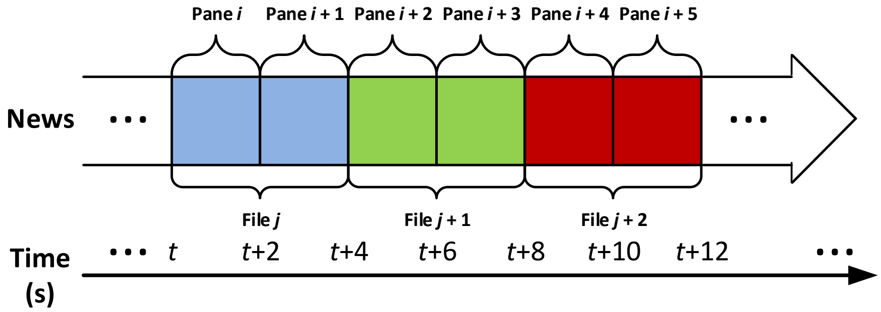


**Figure 4.1:** A Partition Example

## 4.2 Dynamic Data Packer

Given the above partition strategy, we describe how to encode the output panes to assure subsequent effective window-centric file access and processing. The Dynamic Data Packer takes as input: 1) the pane-based partitioning plan generated by the Window Semantic Analyzer, and 2) the external input data sources to be consumed. The main task of the dynamic data packer is to exploit the partitioning plan at run-time to pack the input data into panes and store them as physical files in HDFS. Note that the complexity of the pre-processing of the input files to create the panes would depend on the properties of the input files, e.g., sorted or unsorted, and the granularity of the pane sizes to create. For example, if the pane sizes are larger than the input files or the input files are sorted based on the records' timestamps, then the pre-processing involves only scanning the files to create the panes. Otherwise, it will involve a time-based partitioning to divide the records into the appropriate panes. The dynamic packing uses the following **naming convention** to distinguish between the two cases in Algorithm 8:

1. In the oversize case, one *pane* corresponds to exactly one file. The file name follows the format `S#P#`, where S stands for the data source and P for the pane identifier. For example, `S1P1` corresponds to the first pane in data source 1.

2. In the undersized case, multiple *panes* correspond to one file. Here the name follows the format `S#P#_#`, where `#_#` denotes the range of logical panes contained in a file. For example, `S1P1_4` indicates that this file contains the first 4 panes (i.e., panes 1, 2, 3, and 4) from data source 1.

We also introduce a special file header to boost performance for locating selected panes in case 2. Specifically, when a single file contains multiple logical panes, the entire file is not always required by an operation. Thus, a special header to such a file is designed to reduce the latency of finding the required logical panes. This is particularly effective when a file contains a large number of panes caused by a relatively low input rate over a given time period. In the cases where the number of data sources is very large, the Dynamic Data Packer could easily be de-centralized to a distributed design by adopting existing techniques, e.g., [73, 74, 75]. For example, a parallel data source splitting operator could be introduced that splits input data sources of high volume into massively parallel subdivided sources. For simplicity, we assume a centralized Dynamic Data Packer for the remainder of this manuscript.

## 4.3 Adaptivity in Input Data Partitioning

As described above, the Window Semantic Analyzer and Dynamic Data Packer together increase cache utilization and minimize the query processing time for a recurring query. However, the fluctuation in the data rate may cause a query execution to take much longer than expected and may not finish before the next execution (if started on the scheduled next slice). In this case, Redoop switches to a *proactive* processing mode, in which it will start processing the available data and creating partial results as soon as sufficient input data is available. This proactive approach does not guarantee the completion before the next execution, but it is a best-effort approach that can be very effective especially for fine-grained recurring queries with small *slide* parameters.

We propose an *adaptive pane-based partitioning* technique to adaptively partition a pane into sub-panes when faced with workload spikes. Clearly, a larger amount of input data will tend to increase the execution time. The core idea is to exploit the statistics collected from the Execution Profiler, i.e., the execution times and the amount of data processed in the previous executions, to adjust the pane size during the subsequent input data partitioning process. It has been shown that the input data size is one of the dominant factors determining the execution time of a MapReduce job [47]. Thus, the pane size in Redoop is determined by a series of observations of the job execution over time and the corresponding pane sizes. Our solution is to estimate the future behavior of input data sources based on these observations and then produce the pane-based partitioning plan accordingly.

We now describe the *estimation model*. The Execution Profiler, running as a separate thread, collects the statistics from previous executions and transmits them to the Window Semantic Analyzer. These statistics are a series of observations of the job execution time, denoted by $X_i$ for the $i$-th query recurrence. We utilize double exponential smoothing of previous recurrences to estimate the execution time of $i + k$-th query recurrence, denoted by $\hat{X}_{i+k}$. As statistics are collected, the value for the local mean level $L_i$ and trend $T_i$ of the execution time is periodically updated as follows:

$$L_i = \alpha \cdot X_i + (1 - \alpha)(L_{i-1} + T_{i-1}) \tag{4.1}$$

$$T_i = \gamma \cdot (L_i - L_{i-1}) + (1 - \gamma) \cdot T_{i-1} \tag{4.2}$$

The smoothing parameters $\alpha$ and $\gamma$ can be selected by fitting historical data (for details please refer to [76]).

Using the updated values of level $L_i$ and trend $T_i$, the execution time of $i + k$-th query recurrence is computed as:

$$\hat{X}_{t+k} = L_t + k \cdot T_t \tag{4.3}$$

If the Window Semantic Analyzer detects a potential execution time change by using the above equations, then the current pane size will need to change. Therefore, the Window Semantic Analyzer applies the scale factor (i.e., the ratio between the expected execution time and the previous one) to generate a new pane size for the input data partitioning. The new plan accommodating the data variation is then dynamically adopted by the data packer. If the new plan encodes a finer-granular data unit compared to the original partition plan, then Redoop system will automatically switch to the proactive processing mode for that query, i.e., the Window Semantic Analyzer will trigger the query execution as soon as the first data partition with the new pane size becomes available rather than waiting for the data of a complete window to become available. This separation of concern between the optimizer (Window Semantic Analyzer) that determines the partition plan and the executor (Dynamic Data Packer) that implements the chosen partition plan makes the input data partitioning adaptivity extremely light-weight. Namely, the system can simply plug in a new plan selected by the Window Semantic Analyzer in constant time.

Note that this proactive approach offers several advantages compared to using a fixed partitioning plan: 1) the granularity of job executions is decreased as sub-panes will be populated faster than entire panes or windows, 2) multiple sub-panes can be processed concurrently at arbitrary computing nodes to further distribute and parallelize the reduce computations, and 3) the overhead, namely, in maintaining statistics for average pane sizes over the recent data source history, is relatively small. As will be illustrated in the experimental section, the adaptivity mechanism can achieve up to 3x speedup compared to the base Redoop system without adaptivity.

# 5

# Redoop Window-Aware Caching

To reduce the unnecessary I/O costs resulting from the overlapping windows, Redoop's task nodes cache the input data partitions on their local file systems for subsequent reuse. Our Redoop maintains caches at two stages of a MapReduce job, reduce input and output. Both cached data need not to be loaded, processed or shuffled again with the same mapper across windows. Hence this reduces the processing time for recurring queries. To facilitate caching on local nodes, Redoop maintains additional data structures associated with these caches. Due to data sources being updated periodically, the local file system on task nodes cannot accommodate an unbounded number of historical caches. Thus, it is imperative to purge the expired caches in a timely manner without introducing additional overhead to the Redoop system.

## 5.1 Local Cache Registry

Given the above goals, we now present the meta-data structure (meta-data) on task nodes, which allows Redoop to maintain and use caches on each node. The cache data structure, called local cache registry, consists of three parts, namely, a pane id (`pid`) indicating

| pid | type | expiration |
|---|---|---|
| S1P3 | 1 | 1 |
| S2P4 | 2 | 0 |

**Table 5.1:** Examples of a Local Cache Registry

which pane is cached on the node, a cache type (type) indicating whether the cached pane is a reduce input cache or a reduce output cache, and a flag (expiration) showing whether that the cached pane is still going to be needed by any window operations in the Redoop system. This data structure provides location mapping so that a task node can extract a cache specific to a certain window range from its local file system and process it with respect to the corresponding reduce or finalize operation written by the application programmer. Table 1 shows the local cache registry containing two cache entries. S1P3 indicates that the pane is expired as a reduce output cache, and S2P4, on the other hand, is still being used as a reduce input cache by a recurring query.

Next, we characterize how the local cache registry is maintained under different operations during query processing.

**Adding New Entry.** When a new cache with pane id (pid) is created on a node, its expiration flag is set to 0 (i.e., not expired) and its type is set to 1 (2) if the cache is a reduce input (output) cache. The new entry is simply appended to the local cache registry on the node. The records for existing caches do not need to be changed. After adding a new cache entry into the registry, the node synchronizes and sends the local cache registry to the window-aware cache controller where local cache registries from all task nodes are consolidated. At this point, the TaskScheduler will consult the window-aware cache controller in order to use the newly registered cache in future map or reduce tasks.

Our design is compliant with the Hadoop framework in that the window-aware cache controller (Section 5.2) is responsible for deciding which map or reduce tasks to execute on which node based on the local cache registry information. This grants Redoop the ability to asynchronously extract data partitions (i.e., panes) from any cache and to execute the corresponding task. As consequence, task nodes are flexible to process any task as long as the data or cache is available, rather than waiting for slower nodes to finish a corresponding set of tasks.

**Updating Existing Entry.** When a cache is currently being used by a recurring query, the associated local cache entry needs not to be updated immediately. This avoids communication costs within the cluster. In contrast, the local cache registry is updated only when the task node receives a notification from the window-aware cache controller, which indicates the caches that have expired. Once received a notification, the local cache registry finds the matching cache entries and sets their `expiration` flags to 1.

Updating existing entries in a local cache registry is designed to handle cache purging. Due to the periodic updates on data sources, the local file system on task nodes cannot accommodate an unbounded number of historical caches. Thus, it is imperative to purge the expired caches in a timely manner. However, continuously scanning the local cache registry would introduce additional overhead to the Redoop system. Thus, we propose two light-weight yet efficient mechanisms to purge expired caches on task nodes, namely, $periodic$ and $on$-$demand$ purging. Periodic purging scans the local cache registry periodically based on a adjustable period threshold $PurgeCycle$ controlled by the Redoop administrator. During this scan, all caches with their `expiration` flag on will be purged during this scan. $On$-$demand$ purging instead is designed for an emergency. If the local file system is at risk of running out of space before the system begins the next periodic purging scan, then a $on$-$demand$ purging will be initiated, which deletes any expired caches from the file system instantaneously.

## 5.2 Window-Aware Cache Controller

To reduce I/O costs, Redoop caches the panes on the task node's local file system for subsequent reuse. To further accelerate processing, we introduce a dedicated component of the window-aware cache controller on the master node that is responsible for maintaining the cache information from all task nodes. Below we now describe how Redoop maintains and exploits these caches.

**Global Cache Management.** The window-aware cache controller maintains a summary of all local cache registries under its control. We now design a data structure called *cache signature* that consists of four parts, a cache id (`pid`), a node id (`nid`), a type bit (`type`), a ready bit (`ready`), and a per-cache bit-mask (`doneQueryMask`). Similar to the local cache registry, the `type` has a domain of 3 possible values: 0 (not available), 1 (reduce input cache), and 2 (reduce output cache). The `ready` column has a domain of 3 possible values: 0 (not available), 1 (HDFS available), and 2 (cache available). The `doneQueryMask` indicates which queries have finished their utilization of this cache. These signatures are very compact and easy to manipulate inside the cache controller. Table 2 shows an example of the cache signature with four cache entries.

Each bit in the `doneQueryMask` is associated with one distinct query. Whenever a cache being associated with a query but no longer utilized at that time, the corresponding bit is updated to 1. For ease of processing, the number of bits in the `doneQueryMask` indicator for each cache is identical. If the cache is not used by a given query at all, the corresponding bit is set to 1 automatically at initialization time. Once all bits in the `doneQueryMask` have been flipped to 1, this indicates that the cache will no longer be needed by any of the queries. Consequently, the master node sends a purge notification to the corresponding task nodes storing the cache. This node can be easily identified by the `nid` field. After receiving the notification, the local cache registry on the task node

| pid | nid | type | ready | doneQueryMask |
|---|---|---|---|---|
| S1P3 | 1 | 1 | 2 | 10011 |
| S1P3 | 1 | 2 | 2 | 10011 |
| S2P4 | 0 | 0 | 1 | 10111 |
| ... | ... | ... | ... | ... |
| S1P7 | 1 | 1 | 2 | 10011 |

**Table 5.2:** Example of Window-aware Cache Controller

| | S1P0 | S1P1 | S1P2 | S1P3 |
|---|---|---|---|---|
| **S2P0** | 1 | 1 | 0 | 0 |
| **S2P1** | 1 | 1 | 0 | 0 |
| **S2P2** | 0 | 0 | 0 | 0 |
| **S2P3** | 0 | 0 | 0 | 0 |

**Table 5.3:** Example of Cache Status Matrix

updates the `expiration` field to value 1. Thereafter, the task node purges the cache using either its periodic or on-demand purging policy so as to free the space on its local file system.

Next, we show how each bit of the `doneQueryMask` is updated according to cache status matrix (`Status`), a dynamically updated data structure. In addition, we also introduce the cache status matrix associated with each query that models their respective window constraints. Thus, there are up to $n$ cache status matrices, one for each of the $n$ registered queries. The cache status matrix is a multi-dimensional boolean array. Each dimension (column or row) refers to a series of panes within one data source. Each entry in the array is a boolean flag indicating whether the respective query operation has or has not been completed with the corresponding panes of the other dimensions. Querying and updating the matrix for a given cache signature is a very efficient lookup operation. Table 3 depicts an example of a cache status matrix for a binary join query. The extension to higher dimensions is straightforward.

Next we describe operations designed on this status matrix.

**Initialization.** The `Status` matrix is initialized when its associated query is added to the Redoop system. The number of dimensions of the `Status` matrix is determined by the number of data sources involved in this query. The entries for each dimension are directly derived from the window constraints on each source. For example, if the window size of input data sources $S1$ and $S2$ are both 4 hours and the pane size of these two data sources is 1 hour, then the `Status` is initialized with a 4 by 4 matrix, as shown in Table 3. All elements in `status` are initialized with zeros.

**Update.** Whenever a reduce task is completed, the element in `status` is located by the indices of the panes involved in the task. The value of the element is updated from 0 to 1, indicating this particular task is done. For example, as shown in Table 3, assuming that the reduce task joining `S1P3` with `S2P2` is completed, then the JobTracker triggers the update of the `status` matrix. In the matrix, the indices of $S1$ and $S2$ are 3 and 2 respectively. Then the value of the element `status[3][2]` is updated to 1.

**Expiration.** As described in Section 5.1, the Redoop system purges caches after their expiration. Checking whether or not a pane can be safely purged is not straightforward, as the expiration is determined by the panes in all other sources involved in the operation. For example, as shown in Figure 5.1, the query associated with this matrix is a binary join of $S1$ and $S2$. The pane `S1P1` expires once it completes joining with its corresponding pane partners from the matching data source $S2$. In this case, those range from `S2P1` to `S2P3`.

To efficiently detect the expired panes that can be safely purged from the system, Redoop computes a `lifeSpan` for each pane that indicates the range of panes from the other data sources with which each pane should be processed. Thus, a set of `lifeSpan` values is associated with a given pane, namely, one per matching join partner. Thus the cardinality of the `lifeSpan` set is $m$, where $m$ is the number of join partners in the operation. For a given pane `SiPx`, its `lifeSpan` on data source $S_j$ is computed as follows:

$$lifeSpan_{ij} = win_j + slide_i \cdot \lfloor slide_i / slide_j \rfloor \tag{5.1}$$

$$lifeSpan_{ij} = slide_j \cdot \lfloor slide_i \cdot \lceil win_i / slide_i \rceil / slide_j \rfloor \tag{5.2}$$

where $win_i$ ($win_j$) and $sld_i$ ($sld_j$) are the window and slide sizes of data source $S_i$ ($S_j$), respectively. We distinguish between two different cases, namely, when the pane is not within the overlapping part of consecutive windows (Equation 5.1), and when it is one of the overlapping panes of consecutive windows (Equation 5.2).

Given the `lifeSpan` of a pane, we propose an optimized approach to determine whether or not the pane is expired. Specifically, whenever an entry in `status` is updated to 1, we check if the corresponding pane of each data source is still being used by the operation on the window that the pane belongs to. If the pane does not belong to the current window of its data source, then we check whether or not all elements within its `lifeSpan` correspond to the value 1. Once all the elements within a pane's `lifeSpan` are set to value 1 (done), then the corresponding bit of `doneQueryMask` in the global cache registry can be updated accordingly, i.e., it also is set to 1. Namely, the pane is marked as expired with respect to its corresponding query.

In Figure 5.1(b), we show how this expiration mechanism applies to the `Status` matrix for a binary join query. If the current window of $S1$ consists of panes from `S1P5` to `S1P7`, then `S1P4` is considered expired for two reasons. First, it is no longer part of the current window of $S1$. Second, all panes of $S2$ within `S1P4`'s `lifeSpan` (panes `S2P3`, `S2P4`, and `S2P5`) have a value of 1. Thus, we can safely set the corresponding bit in `doneQueryMask` to 1.

**Purging Expired Elements.** To avoid an infinite growth of the `status` matrix, Redoop periodically purges the meta description about the expired panes to accommodate for new ones. Logically, purging is accomplished by shifting the array in each dimension from the high-index to the low-index side. The purging task is triggered periodically based on a configurable parameter $PurgeCycle$, a user-defined configuration parameter in Redoop. Its default value is the slide size $Slide$ of the data source in each dimension. During the shifting, we scan each element in each dimension in ascending order by pane id until an element indicates that the task has not yet been done. Thereafter, we can safely remove the consecutive “done” panes and in their place insert the same number of new panes into the matrix with an initialized value zero.

Figures 5.1(b) and (c) illustrate this shifting process using an example. Assuming that the window constraints on $S1$ and $S2$ are the same ($win$ = 3 mins and $slide$ = 2 mins). Then the default shifting period is 2 minutes (i.e., $PurgeCycle$ = $slide$). Thus, the `Status` matrix purges expired elements every 2 minutes. In Figure 5.1(b), we start scanning the matrix from `S2P1` horizontally. The first four elements in row 1 indicate that the corresponding pairs of panes have been processed with respect to their `lifeSpan`. For example, the `lifeSpan` of `S2P2` and `S2P3` are 3 and 5 panes, respectively. Elements corresponding to their `lifeSpan` have value 1. Thus, we can safely shift up the first 4 rows in the matrix and insert 4 new panes in $S2$'s dimension, namely, from `S2P8` to `S2P11`. The same process applies to $S1$ dimension as well. As a result, 4 new columns, from `S1P8` to `S1P11`, are inserted into $S1$'s dimension. Note that, the element of (`S1P5`, `S2P5`) is not removed even though its value is 1, because neither `S1P5` nor `S2P5` have completely exhausted their set of tasks with other panes within their `lifeSpan`, respectively. For example, as indicated in Figure 5.1(b), the elements of `(S1P5,S2P6)` and `(S1P5,S2P7)` are both still 0. Therefore, the shifted matrix `status` is updated only as depicted in Figure 5.1(c).

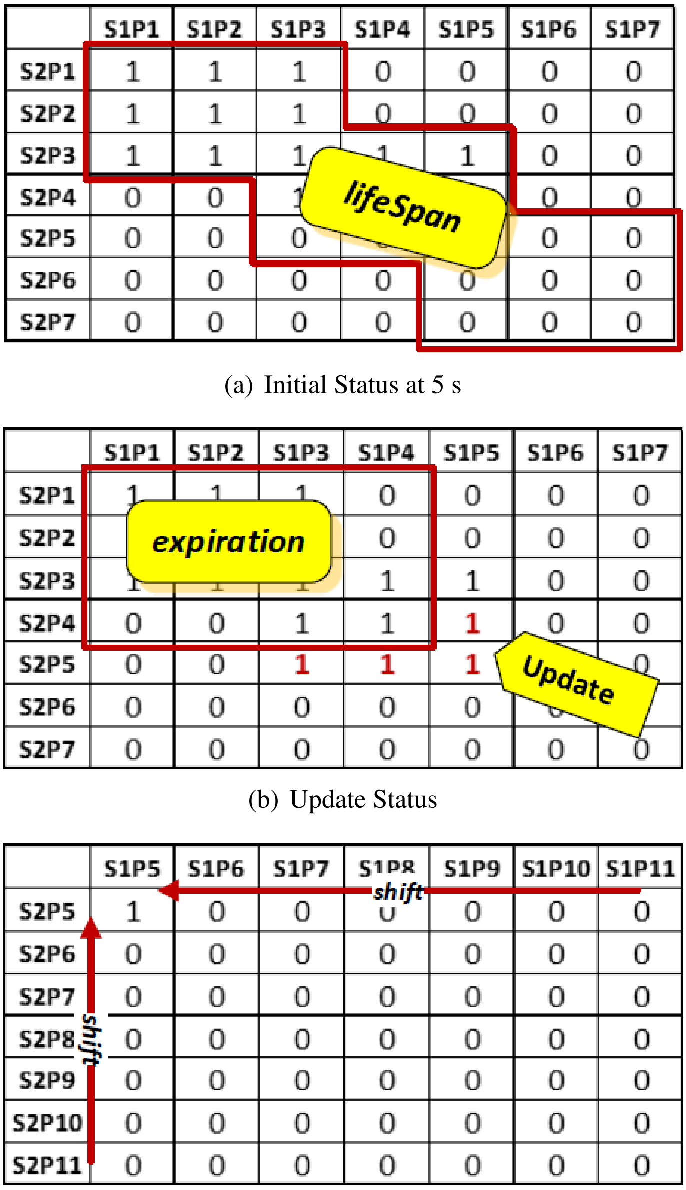
The design of the cache status matrix is compact, as the system only keeps one such data structure for each query. Moreover, all operations on this `status` matrix introduce minimum overhead to the window-aware cache controller. Specifically, the costs of matrix initialization, update, and expiration are linear in the size of matrix. The shifting operation's costs are identical to those of the matrix initialization costs in the worst case. Also, shifting is only triggered periodically. Thus, the maintenance of this cache status matrix is negligible.

## 5.3 Cache-Aware Task Scheduling

The goal of the Redoop's scheduler is to schedule tasks that exploit the window-centric cached partitions as much as possible, reducing redundant work across window panes. For example, Figure 5.2 is a sample schedule for a query joining data sources $S1$ and $S2$. To improve performance, window-centric partitions from $S1$ and $S2$ are cached and reused in the recurring query processing.

Two task nodes are involved in this job. The scheduling of window 1 in Redoop is no different than in Hadoop: the map tasks are arbitrarily distributed across the two task nodes as are the reduce tasks. In the join step of window 1, the input states are `S1P1` and `S2P1`. Two map tasks are executed, each of which loads appropriate window partitions from input files into the local file system. As in the original Hadoop architecture, the map and reduce tasks are executed with each processing the input data according to hash values on the join attribute.

The scheduling of the join step of window 2 of data source $S1$ can take advantage of the cache on data source $S2$ produced by window 1: the map task that processes the specific data partition `S2P1` is thus scheduled on the task node where that data partition was processed for window 1. That is when its cache already resides as determined by previous task scheduling decisions.



| | S1P1 | S1P2 | S1P3 | S1P4 | S1P5 | S1P6 | S1P7 |
|---|---|---|---|---|---|---|---|
| S2P1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| S2P2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| S2P3 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| S2P4 | 0 | 0 | 1 | [illegible] | [illegible] | 0 | 0 |
| S2P5 | 0 | 0 | 0 | 0 | [illegible] | 0 | 0 |
| S2P6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2P7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(a) Initial Status at 5 s

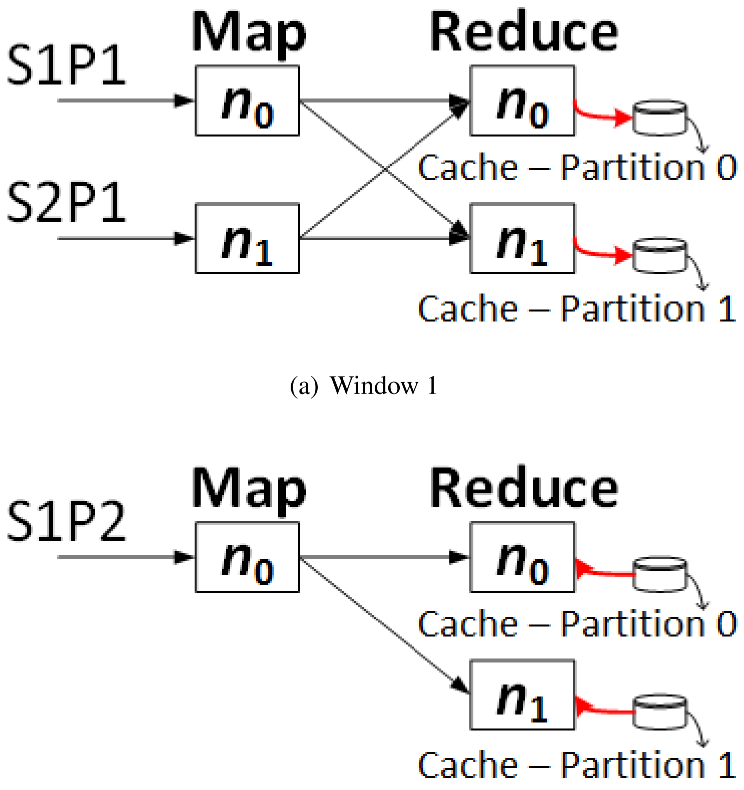
| | S1P1 | S1P2 | S1P3 | S1P4 | S1P5 | S1P6 | S1P7 |
|---|---|---|---|---|---|---|---|
| S2P1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| S2P2 | [illegible] | [illegible] | [illegible] | 0 | 0 | 0 | 0 |
| S2P3 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| S2P4 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| S2P5 | 0 | 0 | 1 | 1 | 1 | [illegible] | 0 |
| S2P6 | 0 | 0 | 0 | 0 | 0 | 0 | [illegible] |
| S2P7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(b) Update Status

| | S1P5 | S1P6 | S1P7 | S1P8 | S1P9 | S1P10 | S1P11 |
|---|---|---|---|---|---|---|---|
| S2P5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2P6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2P7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2P8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2P9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2P10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2P11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(c) Shift Status at 7 s

**Figure 5.1:** Operations on Cache Status Matrix



(a) Window 1

(b) Window 2

**Figure 5.2:** Cache-Aware Task Scheduling

The schedule in Figure 5.2 provides the ability to reuse historical data cached on the local file system. There is no need to re-compute these map outputs nor to communicate them to the reducers. In window 1, if the reducer input partitions 0 and 1 are stored on nodes $n_1$ and $n_2$, respectively, then in window 2, these partitions need not be loaded, processed, nor shuffled again. In that case, in window 2, only the new data partitions need `S1P2` to be processed. With this strategy, the reducer input now physically comes from two different sources: the output from the mappers (i.e., for the new input data) and the local file system (i.e., for the caches of previous panes).

In order to maximize the cache utilization in Redoop, we assume that the number of reducers in a given job does not change over time. Moreover, the partitioning functions used between mappers and reducers are fixed. Two separate lists, $mapTaskList$ and $reduceTaskList$, are introduced for each type of tasks. This separation into two lists helps the scheduler find the appropriate task to schedule, depending on the type of the available task slot on the task nodes. Both map and reduce task lists are associated with the window-aware cache controller. Specifically, whenever a `ready` bit of a data partition in the window-aware cache controller turns to 1 (available in HDFS), then the task using this data partition is added to the map task list. This new entry indicates that a map task can now be scheduled because its data partition has newly arrived. If the `ready` bit of a data partition turns to 2 (cache available 1), then it will be matched up with the other panes (which also have cache available) based on its `lifeSpan` with respect to the other data sources. Then, the cache pairs are added to the reduce task list. For example, whenever `S2P4`'s ready bit in Table 2 becomes 2, it will be paired with `S1P3`. As a result, the cache pair (`S1P3`, `S2P4`) is inserted into the reduce task list. However, `S2P4` would not be paired with `S1P7` since `S1P7` is beyond `S2P4`'s life span, assuming the window and slide sizes of $S1$ are 3 minutes and 2 minutes (i.e., pane size is $gcd(3, 2)$ = 1 minute), respectively.

The cache-aware task scheduler also tries to balance the workloads on each node when it decides on the task assignment. That is, if the scheduler assigns the map or reduce tasks only based on the locality of prior caches, the nodes storing these caches could be overloaded quickly. Thus, our scheduler combines two metrics to improve Redoop's performance: the load in each node and the affinity between the newly arrival data and the cached data on each node. The first metric is used to assign a task to the node with more resources available, and the second one tries to additionally exploit the locality of cache. The combined metric for task assignment is shown below:

$$node = arg \min_{i \in N} \left[ Load_i + C_{task,i} \right] \tag{5.3}$$

where $Load_i$ indicates the current load on the $i$th node, and $C_{task,i}$ denotes the I/O cost for a given task. We plug in the cost model proposed by Li et al. in SOPA [47] to calculate $C_{task,i}$, as the I/O cost is shown to be the dominant cost. Given a task $task$ to schedule, $C_{task,i}$ is lower for the nodes where the required data is cached, and it is higher for the rest of the nodes. The node with the minimal value from Equation 5.3 is selected. For example, if all task slots of a node have been taken by map or reduce tasks, the scheduler assigns the new task to a different node even if a fully loaded node has the desired cache available. In this case, the cache would not be used for this particular task and the task execution is identical to a regular map or reduce task.

Algorithm 4 describes our proposed cache-aware task scheduling algorithm. In the beginning of a recurring job, the tasks are scheduled exactly the same as what would have been done by Hadoop (Lines 2-5). The master node gets the cache information from the window-aware cache controller. Thereafter, the scheduler consults the two map and reduce task lists to assign any map or reduce task to an available node. If the map task list is not empty (Line 6), the scheduler selects a node to assign the task from this map task list in FIFO order (Lines 7-9). Then the scheduler updates the associated information of the data partitions that participated in the scheduled task, such as the local cache registry, the window-aware cache controller, and the map task list (Lines 10-12). These update operations follow the logic described in the above subsections.

If the reduce task list is not empty (Line 13), the scheduler selects the appropriate node by using Equation 5.3 for a reduce task from the $reduceTaskList$. The selection takes both workload balancing and cache utilization on a node into account. (Lines 14-16). Specifically, the scheduler tries to schedule the task in which both data partitions are available as caches. If not, the scheduler prefers the task containing at least one partition that is available as cache. Once selected, the scheduler removes the scheduled task from the list and updates the window-aware cache controller accordingly (Lines 17-18).

**Algorithm 4** Cache-Aware Task Scheduling Algorithm

```
Input: Node $node$, Map$\langle Node, List\ \langle Partition \rangle\rangle$ $cache$
boolean $start$ = true
if $start$ = true then
    Partition $p$ = defaultSchedule($node$);
    $cache$.get($node$).add($p$);
    $start$ = false;
else if !$mapTaskList$.isEmpty() then
    Partition $p$ = $mapTaskList$.get(0);
    $node$ = selectNode(M, p);  // select a node for a map task
    schedule($p$, $node$);
    $cache$.get($node$).add($p$);
    $mapTaskList$.remove(0);
    updateCache($p$);    // update cache associated information
else if !$reduceTaskList$.isEmpty() then
    Partition $p$ = $reduceTaskList$.get(0);
    $node$ = selectNode(R, p);  // select a node for a reduce task
    schedule($p$, $node$);
    $reduceTaskList$.remove($index$);
    updateCache($p$);    // update cache associated information
```

# 6

# Redoop Implementation

This section presents the Redoop implementation details.

**Window Controller and API.** As Figure 3.1 depicts, Redoop's window-aware cache controller is added as a new component to the Hadoop architecture. For this, the task scheduler is modified from Hadoop's classes JobInProgress and TaskScheduler, respectively. Specifically, the cache-oriented TaskScheduler fully exploits the cache information to schedule when and which task to assign to the available task slot.

Additionally, Redoop extends the Hadoop API to facilitate client programming. Concretely, the programmer specifies a recurring query by providing:

1. The computation performed by each map and reduce in the recurring query's body. These functions have exactly the same interfaces as they do in Hadoop.

2. The window constraints of $win$ and $slide$ associated with data sources. The constraints are used to initialize the window-aware cache controller and to associate the input files with each pane.

3. To specify the inputs and output of each execution, the programmer implements two functions: GetInputPaths(int recurrence, int win, int slide) and GetOutputPaths(int recurrence, int win, int slide), where recurrence indicates the number of times that a

window has slided. Specifically, the returned input file paths consists of two kinds of input data, newly arrival data, and cached input data from prior query recurrences. The returned output file path is expected to have a unique output path for future query executions.

4. The application-specific incremental computation function finalization(int[] recurrences) (discussed in Chapter 4.1) expresses different patterns of processing, such as stateless or stateful incremental. Our Redoop returns the required intermediate data files according to the recurrence numbers.

**Caching.** We have implemented Redoop's caching mechanism by modifying the classes MapTask, ReduceTask, and TaskTracker. In map or reduce tasks, Redoop creates a directory in the local file system to store the cached data. The directory, under the task's working directory, is tagged with the cache name (following the naming convention described in Section 4.2). With this approach, a task accessing the cache in the future can access the data for a specific window and pane as needed. After the recurring query finishes, all files storing cached data are removed from HDFS.

**Purging.** Redoop conducts the window and pane purging in a distributed fashion. After the reduce phase of the specified window operation, a ReduceTask updates the lineage (`status`) of the window/pane cache used by that operation. Then, the host TaskTracker sends the updated lineage information back to JobInProgress. JobInProgress collects the updated information of the locally updated lineage of each window/pane cache. If all boolean bits in one lineage (bitmask) of a window/pane have been marked as 1, then denotes that the window/pane does not need to further match with other data sources. Then JobInProgress will raise a ”window/pane expire” event to notify the window/pane purging. Otherwise, JobInProgress simply keeps the cache for possible utilization in the operations.

**Failure Recovery.** Redoop retains the desired fault-tolerance properties of Hadoop through the following mechanisms:

1. Redoop inherits Hadoop's strategy for task failures: a failed task is restarted some number of times before it causes the job to fail. Intermediate data from unexpired panes is maintained on local disks to recover from a task failure.

2. In the case of a slave node failure, all work assigned to this node is rescheduled. Further, any data held on the failed node must be reconstructed. The additional failure situation is introduced by the caches, since they are written only to local disks rather than being replicated to the HDFS. However, rebuilding these lost caches on the failed node in Redoop is naturally achieved by re-executing the tasks hosted by the failed node. During these task re-executions on other nodes, the caches are re-constructed on their latest hosts accordingly without incurring any additional costs.

3. In addition to the cache re-construction, a task failure or slave node failure also triggers rollbacks on the data structures associated with the cache, such as the window-aware cache controller and the map/reduce task lists. Specifically, if a cache is lost, the `ready` bit of the associated pane must be changed to 1 (HDFS-available). The scheduled tasks, using this cache, must be removed from the $ReduceTaskList$ immediately by the $TaskScheduler$. Thereafter, a new task should be inserted into the $MapTaskList$ to re-construct this cache as in Hadoop failure recovery is completely transparent to user programs.

# 7

# Experimental Evaluation

The goal of this experimental study is to show that the Redoop framework achieves its goals. Namely, we will show that: (1) Redoop seamlessly supports window-based recurring query processing over data sources, (2) Redoop outperforms Hadoop system significantly with caching enabled, and (3) Redoop is effective even under input data fluctuations due to adaptivity support.

## 7.1 Experiment Setup & Methodologies

We implemented the Redoop framework as an extension to the open-source Hadoop. We enriched the Hadoop framework by integrating our techniques for recurring queries. All the experiments were conducted on a shared-nothing compute cluster of 30 slave nodes and a single master node. Each server consisted of one quad-core Intel Core Duo 2.6GHz processors, 8GB RAM, 76GB disk, and interconnected using 1Gbit Ethernet. Each server ran Linux (kernel version 2.6.18), Java 1.6, Hadoop 0.20.2. Each worker was configured to run up to 6 map and 2 reduce tasks concurrently. The sort buffer size was set to 512MB, and speculative execution was turned off so to boost performance. The replication factor was set to 3 unless stated otherwise.

**Datasets.** We use two real datasets. The WorldCup Click (WCC) dataset (236GB) [77] records all 1.3 billion requests made to the 1998 World Cup Web site. The second dataset is from high velocity sensor data (FFG) collected from a football field of the Nuremberg Stadium in Germany [78]. The FFG dataset (26GB) is repeatedly used for each window.

**Metrics & Measurements.** We measure the most common metric for data management systems, namely the processing time. Our results are the average over 10 runs. While the experiments are reported using time-based sliding windows, count-based windows provide similar results.

**Methodology.** We evaluate the performance of the Redoop system for two key computational tasks, namely, recurring aggregation and join queries. Both are implemented in Redoop and in Hadoop (using the traditional driver approach). For each type of query, we compare the performance of both systems by varying the most important parameters of the window-based recurring queries. Specifically, our experiments vary the factor called $overlap$, which corresponds to the ratio between slide size $slide$ and window size $win$, to measure scalability and efficiency of Redoop pane-based caching on high volume data sources. Moreover, we measure how well the Redoop system copes with the input data fluctuations. Last but not least, we demonstrate Redoop's fault tolerance by conducting experiments with slave node failures.

## 7.2 Effect of Pane-based Caching

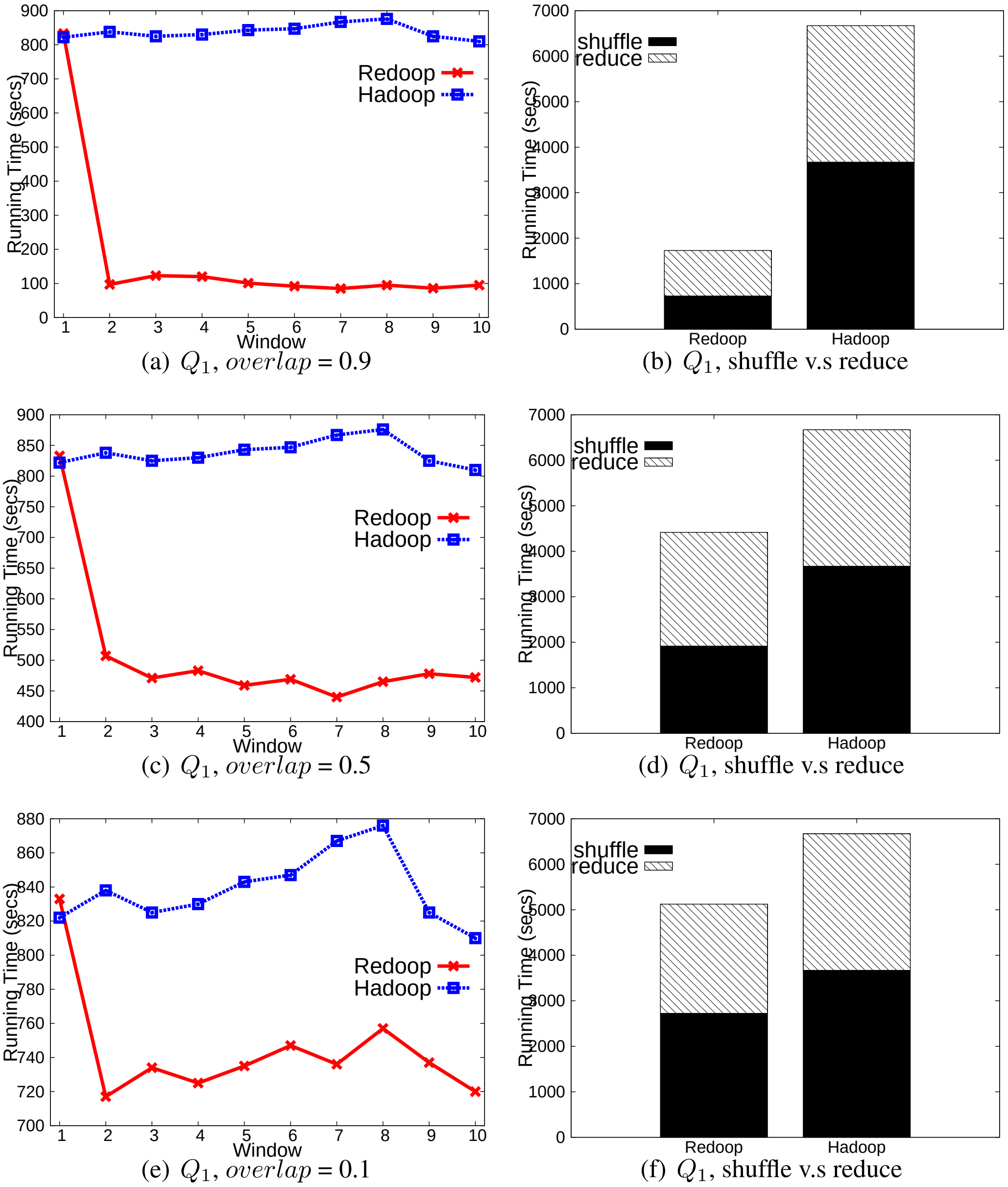
Our incremental processing experiments evaluate the Redoop system's performance by varying the factor $overlap = \frac{win-slide}{win}$, a ratio between the slide and window size. This ratio represents the portion of the newly arriving data tuples in the window after the window slides. Thus, the higher the ratio is, the greater the amount of data shared between consecutive windows.

### 7.2.1 Aggregation Query Evaluation

We run an aggregation query over the WCC dataset that ranks the movements of players. Figure 7.1 shows the results for Redoop and Hadoop. Overall, as the figures show, for an aggregation job, Redoop significantly improves on the run-time when the cache is enabled. We now describe these results in more detail.

**Overall response time.** In this experiment (Figure 7.1), Redoop significantly outperforms Hadoop. Redoop's task scheduling and pane-based caching substantially reduce aggregation time. The running time depicted in Figures 7.1(a), 7.1(c), and 7.1(e) (left column) demonstrate the response time of the aggregation query for 10 windows. In all three figures, for the initial window, both Redoop and Hadoop need to process the whole window full of tuples and thus they achieve similar performance. Hadoop is slightly faster because it does not cache the data produced by the mappers. For the subsequent sliding steps (windows 2-10), Redoop benefits from the pane-based caching, resulting in a significant advantage over Hadoop. Reusing the cached results of previously processed data, Redoop only needs to process the newly arriving tuples after the window slides, and then merge all intermediate results. Figure 7.1(a) achieves the best improvement (lowering the response time by a factor of 8 on average) among the three settings, because its $overlap$ = 90%. Namely, 90% data tuples in each window are cached on the Redoop local file system from each previous window processing step.

**Time distribution.** To better understand Redoop's improvements to each processing phase, we also compared the cost distribution of the aggregation across the Shuffle and Reduce phases. Figures 7.1(b), 7.1(d), and 7.1(f) (right column) show the sum of the cost distribution of the aggregation for 10 windows. The Y axis shows the time spent on each phase. In both Redoop and Hadoop, reducers start to copy data immediately



(a) $Q_1$, $overlap = 0.9$

(b) $Q_1$, shuffle v.s reduce

(c) $Q_1$, $overlap = 0.5$

(d) $Q_1$, shuffle v.s reduce

(e) $Q_1$, $overlap = 0.1$

(f) $Q_1$, shuffle v.s reduce

**Figure 7.1:** Aggregation Query Performance

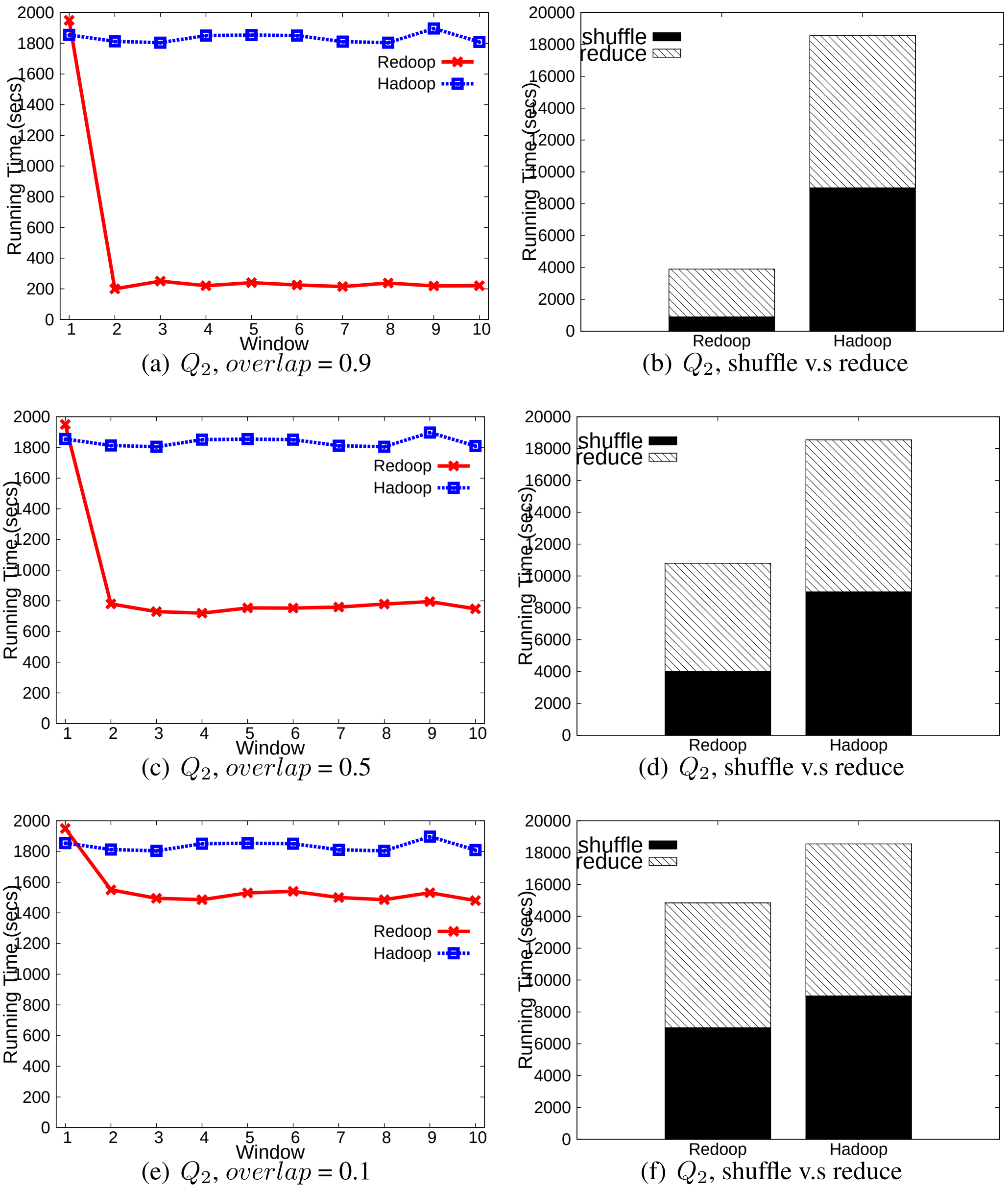
after the first mapper completes. “Shuffle time” is normally the time between reducers starting to copy map output data, and reducers starting to sort copied data; shuffling is concurrent with the rest of the unfinished mappers. The reduce time in the figures is the total time a reducer spends after the shuffle phase, including sorting and grouping, as well as accumulated Reduce function call times. Considering all three charts, we conclude that Redoop outperforms Hadoop in both phases.

Both the “shuffle” and “reduce” bars of Redoop are shorter compared to Hadoop, because Redoop takes advantage of the cached data. Also, the aggregation between new data tuples and cached data is pane-based rather than tuple-based since the data in caches is already aggregated by previous processing. Thus, the cost becomes negligible in Figure 7.1(b). By contrast, for Figure 7.1(f), the cache does not help much, because the $overlap$ (10%) is low. The results in Figure 7.1 clearly demonstrate the effectiveness of our incremental design gained by using pane-based caching.

### 7.2.2 Join Query Evaluation

We run a join query on the FFG dataset with the same system settings as above. Overall, as Figure 7.2 shows, for a join job, Redoop achieves better performance by winning almost an order of magnitude in the best case scenario with the cache enabled, as described below.

**Overall response time.** Redoop’s performance shows a similar pattern with the join query on the FFG dataset. The running times in Figures 7.2(a), 7.2(c), and 7.2(e) demonstrate the processing time of the join query for 10 windows. Again, for the initial window, both Redoop and Hadoop need to process the whole window full of tuples and thus achieve similar performance. For the subsequent sliding steps (windows 2-10), Redoop benefits from the pane-based caching, resulting in a significant performance advantage over Hadoop. Figure 7.2(a) achieves 9 fold performance improvement by taking advan-



(a) $Q_2$, $overlap = 0.9$

(b) $Q_2$, shuffle v.s reduce

(c) $Q_2$, $overlap = 0.5$

(d) $Q_2$, shuffle v.s reduce

(e) $Q_2$, $overlap = 0.1$

(f) $Q_2$, shuffle v.s reduce

**Figure 7.2:** Join Query Performance

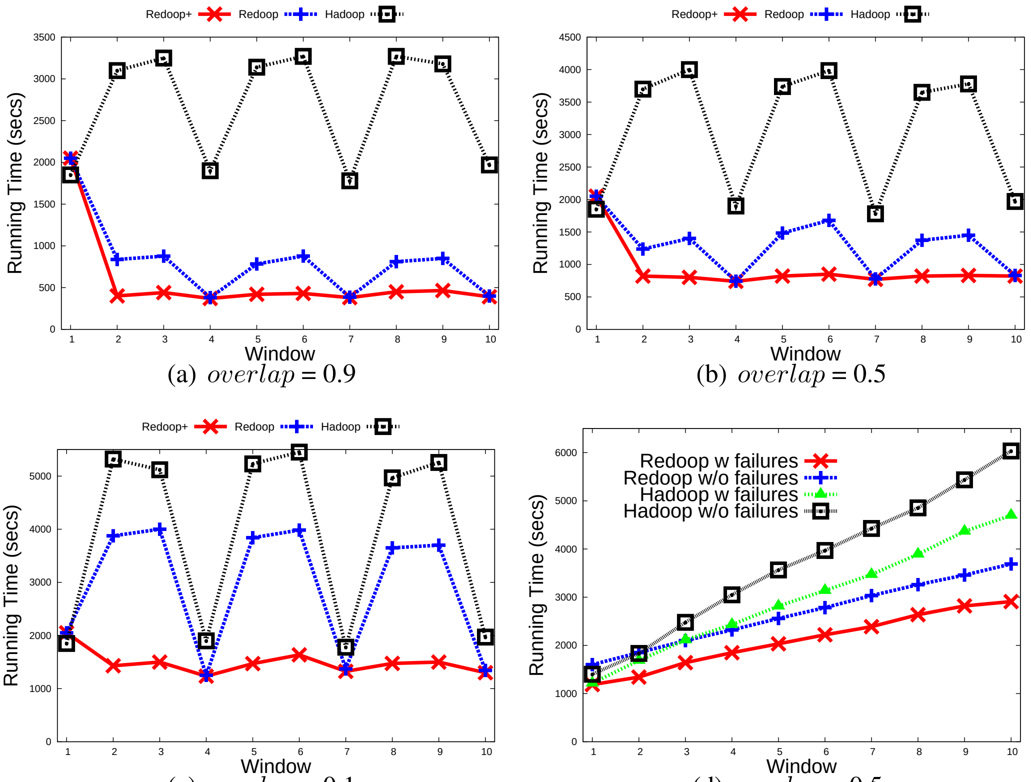
tage of reusing a huge portion of its cache data ($overlap$ = 90%).

**Time distribution.** Similarly, we compared the cost distribution of the join processing across the Shuffle and Reduce phases. Figures 7.2(b), 7.2(d), and 7.2(f) show the sum of the time distribution of the join for 10 windows corresponding to different $overlap$ settings. As expected, both the "shuffle" and "reduce" bars of Redoop are shorter compared to Hadoop, because it takes advantage of the cache data. In particular, the reducers in Redoop only need to process the incremental inputs and produce new results which are combined with the cached reducer outputs from last occurrence to form the final results. In contrast to the results of the aggregation query, the time distribution is much different. Figure 7.2(b) shows that the reduce phase is the dominant time. The reasons include the join selectivity, the implementation of the join operation, etc. However, our pane-based caching is orthogonal to these factors and the possible optimization techniques for these factors are out of the scope of this paper. What is interesting to observe is that by exploiting pane-base caching for recurring queries, Redoop achieves an up to 9 times performance gain compared to the basic Hadoop.

### 7.2.3 Effect of Adaptive Input Partitioning

In this experiment, we study the effectiveness of our adaptive strategy that handles input data fluctuations. For comparison, we again use Hadoop as baseline. We measure the processing time for 10 windows. The workload used in this experiment varies periodically. Specifically, windows 1, 4, 7, and 10 have the normal workloads. The workloads of the rest of the windows are doubled. Figure 7.3(d) shows the processing time for 10 windows with three different settings ($overlap$ = 90%, 50%, and 10%).

For very large overlaps as shown in Figure 7.3(a), Hadoop is outperformed by the adaptive Redoop and even Redoop. The reason is that when having workload spikes, adaptive processing as exploited in Redoop, smooths out the doubled workloads by start-



(a) $overlap = 0.9$

(b) $overlap = 0.5$

(c) $overlap = 0.1$

(d) $overlap = 0.5$

**Figure 7.3:** Adaptive Input Partitioning & Fault Tolerance

ing the query execution earlier with the newly arrival data at a finer granularity (i.e., sub-panes). Redoop without exploiting such adaptive strategy would waste time on waiting for more data than what it could possibly handle at a time. Worse yet, Hadoop uses the default batch processing strategy, which only starts processing data whenever the window slides. On the other hand, the adaptive strategy avoids creating too many small sub-panes by exploiting the estimation model discussed in Section 4.3.

However, as the overlaps grow, we observe an interesting behavior by Redoop without adaption. In Figure 7.3(c), we see that Redoop only has slight gain over Hadoop. This time, the amount of data and thus computation needed become more significant in a shorter time period. Neither Hadoop nor Redoop without the adaptive strategy can handle such high workload spikes. On the contrary, we observe that the adaptive Redoop starts query execution earlier rather than waiting until all data had been received. This gives excellent results, even outperforming the basic Redoop by 2.7 times on average during the workload fluctuations. In summary, the above results demonstrate the effectiveness of the adaptive input partitioning strategy in Redoop.

### 7.2.4 Fault Tolerance

Redoop and plain Hadoop will have the same behavior with respect to a slave node failure. Thus, in this section, we focus on cache failure where the cached data is lost from a given node. As middle ground, we use a FFG dataset to run an aggregation query with $overlap$ = $50\%$. We inject cache removals at the beginning of each window, and plot the running time in Figure 7.3(d), where Redoop(f) and Hadoop(f) correspond the cases when task failures happen, and Redoop and Hadoop to the cases without failure. As shown in Figure 7.3(d), Hadoop(f) has the worst performance as we expected. On the contrary, the accumulative running time of Redoop(f) is still much shorter than that of Hadoop. The reason is that Redoop caches the intermediate data in a fine-grained unit (i.e., pane) rather than at the granularity of the whole window. Thus, even some cached intermediate data is lost due to failures, Redoop can still exploit the rest of the caches during its task execution.

Notice that Redoop(f) has a small loss for the first two windows. This is attributed to the cold start of the query processing on Redoop - similar to Figure 7.1 with the same trend. What happens is that with more windows, a larger number of panes are cached. In short, the advantage of pane-base caching remains apparent for Redoop even with task failures.

# 8

# Related Work

**Hadoop Extension.** MapReduce [1] and its open-source implementation Hadoop [2] have emerged as a popular model for large-scale distributed data processing in shared-nothing clusters. However, it has been recognized that Hadoop is not a high performance system as it lacks critical system-level optimizations [79]. Hence, several extensions have been proposed to improve Hadoop's performance for different query types, e.g., SQL-like queries [5, 60], online processing [80], and iterative queries [81]. However, none of these systems support or optimize the data-intensive recurring queries addressed in this paper.

While some studies such as HaLoop [81] and Twister [82] have utilized disk-based caches to improve the performance of Hadoop, their domain of queries, which is the recursive and iterative queries, is different from ours. These systems identify and then maintain invariant data during subsequent recursions. HaLoop [81] caches each stage's input and output to save I/Os during iterations. The major difference between the aforementioned approach and Redoop is that we use a well-understood set of principles from window semantics [70, 71, 72] to provide an end-to-end optimization for supporting recurring queries. Similar to HaLoop, Twister [82] extends MapReduce to preserve data across iterations, in which mappers and reducers are long running processes with distributed memory caches. However, Twister's architecture between mappers and reducers is sensitive to failures. Also the memory cache suffers from potential scalability issues.

ReStore [49] extends Pig to reuse intermediate results of the MapReduce workflows. However, ReStore does not directly support recurring queries. It neither understands recurring queries nor provides caching options for input data at the infrastructure level. In contrast, our work focuses on supporting window-based recurring queries by providing window-aware caching over overlapping data across windows. Moreover, Redoop enables fine-grained control over these caches by efficiently maintaining and exploiting their meta-data.

Nova [44] is the closest work to the Redoop system in that Nova supports the convenient specification and processing of incremental workloads on top of Pig/Hadoop. However, Nova acts as a middle-ware layer on top of Hadoop that is treated as a black-box system. Thus, Nova can identify which incremental files (deltas) to process in each execution, but it cannot exploit the optimization opportunities offered by Redoop including adaptive data partitioning, caching of the intermediate data to avoid redundant shuffling, cache-aware task scheduling to utilize cache locality, and adaptivity to the data arrival rate.

**In-Memory Hadoop.** Several recent systems have been proposed to support in-memory processing on top of Hadoop including the $M3R$ [46], $SOPA$ [47], C-MR [48], and In-situ (iMR) [41] systems. In general, these systems focus on changing the disk-based processing inherent in Hadoop into memory-based real-time processing, and hence they cannot be applied to the disk-based recurring queries. For example, SOPA [47] replaces the MapReduce I/O intensive merge sort by hash-based in-memory processing. However, not being aware of overlapping windows, SOPA does not provide caching across MapReduce jobs. M3R [83] builds a main-memory implementation of MapReduce and places constraints on the type of supported jobs imposed by available memory size. C-MR [48] supports stream processing by eliminating disk buffers. However, C-MR focuses on aggregation queries over a single stream, rather than providing a general solution of supporting window-based queries. Worse yet, C-MR keeps all intermediate workflows in a shared in-memory buffer. Consequently, no fault tolerance is provided by C-MR. In-situ (iMR) [41] uses the MapReduce programming interface to deploy a single MapReduce job onto an existing data stream management system (DSMS) to support count- or time-based sliding windows. However, iMR only optimizes log processing applications rather than the more general recurring queries that are our focus. Worse yet, iMR computing nodes are fixed to specific map or reduce jobs and thus are unable to benefit from any form of load balancing and cannot adapt to workload or resource volatility.

**Distributed stream processing systems.** Distributed stream processing systems have been proposed to enable scalable and distributed execution of the continuous queries over data streams [57, 84, 85, 86]. However, since these systems are optimized for main-memory processing with real-time response, they are not suitable for the disk-based data-intensive recurring queries. First, they will not scale well to the sheer volume of data that must be processed by our target applications. And second, continuously maintaining the data in memory would waste significant system resources during the inactive periods between recurring query executions. That is why Redoop addresses several challenges that do not apply to streaming systems such as disk-based caching and recovery mechanisms for cache failure, and cache-aware scheduling of tasks.

# Part II

# Shared Execution of Recurring Workloads in MapReduce

# 9

# Helix Window Alignment for Recurring Queries

Before introducing the proposed techniques for the shared execution among multiple recurring queries, we first formally define our overall problem.

**Definition 9.1** *Given a workload of recurring queries $Q = \{q_1, q_2, \ldots q_n\}$, where each recurring query $q_i(w_i, s_i, \vartheta_i)$ is defined with the three constraints window size ($w$), slide ($s$), and SLA function ($\vartheta$), the* ***SLA-aware multiple recurring query optimization problem*** *is to find a shared execution strategy $\mathcal{E}_{shared}$ of the workload that maximizes the cumulative utility score of the queries in $Q$. Our goal is to*

$$Maximize : \sum_{i=1}^{|Q|} uScore(q_i, \vartheta_i(t), \mathcal{E}_{shared}) \tag{9.1}$$

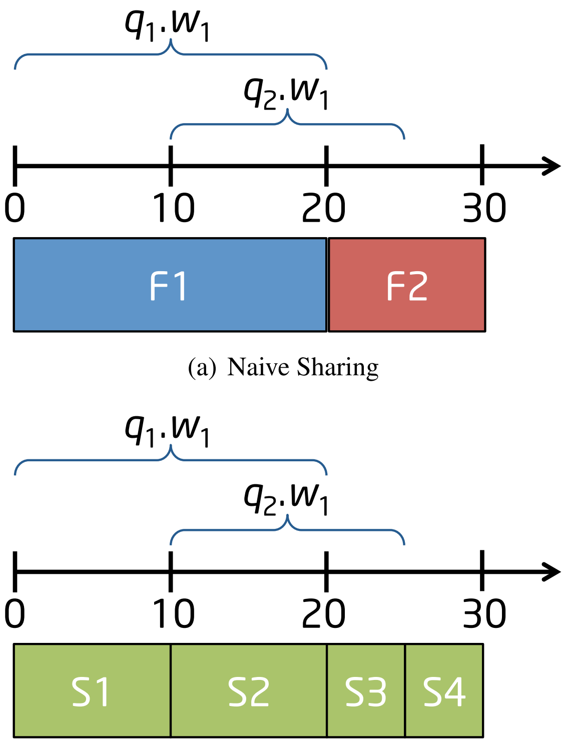
*where $uScore$ is defined as the utility score for $q_i \in Q$ computed from its SLA function $\vartheta(t)$, assuming that the results are generated at time instant $t$.*

Finding the optimal solution for this problem is prohibitively expensive as discussed in Chapter 10. Orchestrating a shared execution of recurring queries with different SLAs is a stochastic knapsack problem, where the stochasticity comes from the fact that the utility score of a query is variable and depends on all previous decisions. Any solution to this problem would need to consider the unique characteristics of recurring queries.

In this chapter, we present a new technique for the shared execution among multiple map-reduce recurring queries, called the *Sliced Window Alignment (SWA)*. The goal of SWA is to tackle the problem of different window constraints among queries, and thus create more opportunities for fine-grained sharing among recurring queries. Our approach first identifies the differences among the scope of interest of each query, and then partitions each of the input data sources into non-overlapping slices. The query processing over the slices produces partial results that can be used to answer multiple queries with little overhead. We first analyze the issues caused by the differences among the scope of interest of query executions (Chapter 9.1). And then, we propose a logical window slicing approach that partitions recurring query windows into aligned slices (Chapter 9.2). Since in Hadoop's context the slices will map to HDFS files, then special considerations needs to be taken into account to avoid creating many small files, which is not optimal for Hadoop. Therefore, in Chapter 9.3, we present the mapping procedure from the logical sliced window to physical HDFS files.

## 9.1 Alignment Problem with Diverse Window Constraints

Given a set of recurring queries with varying window constraints, i.e., window ($w$), slide($s$), and start time ($start$), these parameters may not be well aligned. In this case, sharing work among recurring queries—even if they otherwise have identical tasks (e.g., map input scan, map task, etc.)—may not be beneficial due to their different time scope granularities. Shared query execution without proper data preparation may result in inefficiencies caused by problems of redundant data loading and/or repeated partial data



(a) Naive Sharing

(b) Sharing with SWA

**Figure 9.1:** Relation between Windows

re-computation, as illustrated in the following example.

**Example 9.1 (Naive Sharing)** *Assume that two queries $q_1$ and $q_2$ consume the same input data source and have identical map outputs, with $q_1.w$ = 20, $q_2.w$ = 15, $q_1.s$ = 10, $q_2.s$ = 10, $q_1.start$ = 0, and $q_2.start$ = 10 (Figure 9.1(a)). Assume the input files [0-20] ($F1$), and [20-30] ($F2$) are received in two batches at time $T$ = 20 and $T$ = 30, respectively. Although $q_2$ can share its execution (i.e., partial input scan and the associated map output) with the first execution of $q_1$ over the input file [0, 20], the remaining data [20-25] for the execution of $q_2$ would still need the data from [10-20] in order to produce complete results for $q_2$. In this case, the input file [0-20] has to be loaded and processed again for $q_2$, causing redundant data loading and repeated computations. Moreover, the execution of $q_2$ would also need the data from [20-25] in the second input file [20-30],* *incurring unnecessary data loading from [25-30]. These operations are very expensive and would consume significant system resources.*

**Example 9.2 (Optimized Sharing)** *The ideal case would be to partition the received input files into smaller slices [0-10], [10-20], [20-25], and [25-30] (Figure 9.1(b)). The slice [10-20] serves both queries, while the slice [0-10] and [20-25] only serve $q_1$ and $q_2$, respectively. In this case, each slice with appropriate time granularity would be processed only once - serving both jobs rather than causing unnecessary data loading and computation.*

## 9.2 Aligning Queries in Sliced Windows

We now explain how to best partition an input data source for evaluating recurring queries that reduces both the required I/O operations and computation costs. The idea of partitioning a data stream into slices was first introduced by Li et al. [70] in the pane-based window approach. In this approach, all slices, also called panes, are of equal size. While this may be appropriate for partitioning the input data source for a single recurring query, it is not the most effective approach in the multi-query scenario. Instead, we propose to partition a data source into possibly unequal slices for multiple queries with varying window constraints. We define the sliced window as follows.

**Definition 9.2** *A window $W$ of size $|W| = w$ can be decomposed into $m$ slices $p_i$ with $p_1$.start = 0, $p_i$.end = $p_{i+1}$.start, and $p_m$.end = w. Each slice $p_i$ has size $r_i = (p_i.end - p_i.start)$ ∀ 1 ≤ $i \leq m$. The slices of W are denoted as $W(r_1, \ldots, r_m)$, and the ending position of the $i$-th slice $p_i$ is defined relative to the start of $W$ as $p_i$.end = $r_1 + \ldots + r_i$.*

We start with $Q$, a set of $n$ recurring queries that access the same input data sources, where each query $q_i$ in $Q$ has different window sizes $w$, slides $s$ and logical start times

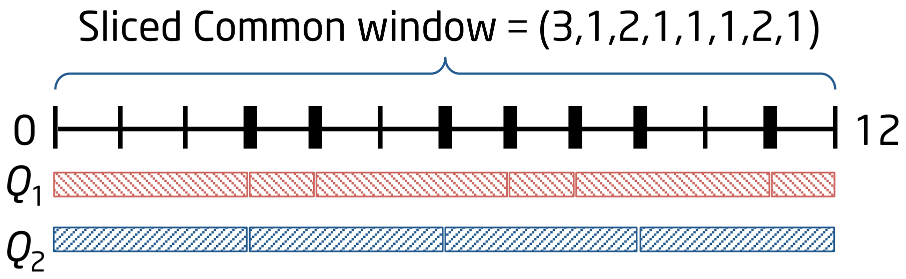
$start$. For simplicity, here we first assume that the start time is the same across all queries. Later, we will relax this constraint. With varying window sizes and slides, we need to find a single *common window* that consists of at least one or multiple consecutive executions of each query $q_i$. Thus, the period of this common window is the lowest common multiple ($LCM$) of the slide $q_i.s$ of each query.

**Example 9.3** *Given two queries $q_1$ and $q_2$, with $q_1$ ($w = 7$, $s = 4$) and $q_2$ ($w = 9$, $s = 6$). We note that $q_1$ and $q_2$ have different slides, namely, 4 and 6. We thus stretch them respectively by factors of 3 and 2 to produce a common window of period 12.*

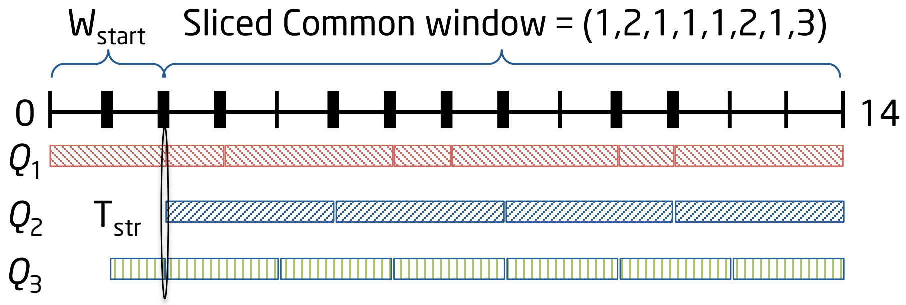
Knowing the size of the common window, we adopt the *paired window* approach [87] to partition it with unequal slice sizes according to the window sizes and slides of $q_i \in Q$. A sliced window of period $s$ is partitioned into a pair of slices, i.e., $p_1 = w \bmod s$ and $p_2 = s - p_1$. Partitioning a window into two slices never creates more slices than the pane-based window approach. Thus it is always better than, or at least as good as, pane-based windows in the context of MapReduce systems. The reason is that in general it may not always be beneficial to execute a job with very small files. The detailed reasons will be explained in Chapter 9.3.

**Example 9.4** *Here we continue using Example 9.3 to demonstrate our solution. The paired windows for $q_1$ and $q_2$ are (3,1) and (3,3), respectively. Then the common window $W = 12$ can be partitioned into either (3,1,3,1,3,1) based on $q_1$'s paired window or (3,3,3,3) based on $q_2$'s paired window. Lastly, we combine these two partition plans together to produce the sliced window $W$ = (3,1,2,1,1,1,2,1), which can serve as the common executions of both queries. The thicker bars show the boundaries in the common sliced window $W$ as shown in Figure 9.2(a).*

We now relax the constraint of identical logical start times of queries in $Q$. Namely, we allow queries with arbitrary logical start time. First, we sort all queries by their logical



(a) Common Sliced Window



(b) Start time

**Figure 9.2:** Sliced Window Example

start time in an ascending order. Then we define the time period from the start time of the first query and the start time of the last query as $W_{start}$. The period $W_{start}$ can then be partitioned based on each query's $q_i.start$. The remaining part of the window of each query becomes the new window size $w'$, i.e., $q_i.w'$ = $q_i.w + q_i.start - W_{start}$. The new $q_i.w'$ with $q_i.s$ are used to generate the common sliced window as described above.

**Example 9.5** *Now assume we have three queries $q_1$, $q_2$, and $q_3$, with $q_1$ ($w = 9$, $s = 4$, $start = 0$), $q_2$ ($w = 9$, $s = 6$, $start = 2$), and $q_3$ ($w = 5$, $s = 2$, $start = 1$). First, we have three queries $q_1$, $q_3$, and $q_2$ in an ascending order of their start time. Thus the period $W_{start}$ is 2, and this time period is partitioned into 2 slices due to $q_3.start = 1$. The remaining part of the window of each query is $q_1.w' = 7$, $q_2.w' = 9$, and $q_3.w' = 4$. With these new window sizes $q_i.w'$ and slides $q_i.s$, we have a common sliced window as shown in Figure 9.2(b)*

Algorithm 5 corresponds to the above SWA approach in pseudo-code. In general, our SWA approach is a combination of building a common sliced window for all queries and using unequal slide sizes. The solution produced by the SWA algorithm is a set of boundaries that partition the common window into unequal slices. By processing at the sliced level of input data, we create more opportunities for fine-grained sharing among recurring queries. Moreover, having the input data in such fine-granularity can alleviate or even avoid redundant data loading and repeated partial data re-computation.

**Algorithm 5** Sliced Window Alignment Algorithm

```
Input: A set of recurring queries $Q$
Output: A set of ending positions $pos$
$tLast \leftarrow$ max($q_i.start$)
for $q_i \in Q$ do
    $pos.add(q_i.t_{str})$
    $q_i.w' \leftarrow q_i.w + q_i.start - tLast$
$CommonWin \leftarrow$ LCM($q_i.s$)
for $q_i \in Q$ do
    $q_i.bList \leftarrow$ slice($q_i, CommonWin$)
    for $b \in q_i.bList$ do
        if $b \notin pos$ then
            $pos$.add($b$)
return $pos$
```

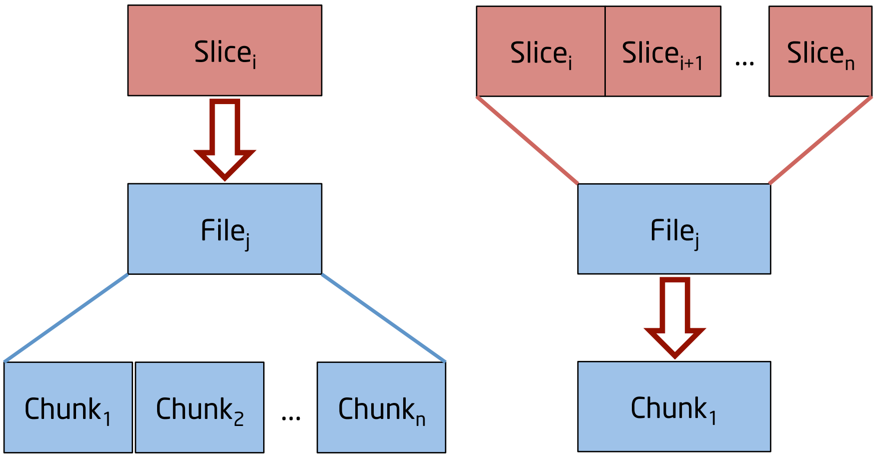
## 9.3 From Slices to Physical Files

After computing the optimal slicing boundaries for input pre-processing, we need to store the data within each slice into physical HDFS files. However, this mapping is not straightforward because the size of each slice varies depending on the actual arrival rate of the corresponding data source. And thus, slices may generate many small HDFS files, which is not optimal for Hadoop's execution. The reason is that HDFS is optimized for processing large data files [64]. Therefore, reading through small files may cause lots of seeks and communications from datanode to datanode to retrieve each small file. If the file is very small and there are many such files, then each map task processes very little input yet imposes extra bookkeeping overhead. Such overhead may offset the potential computational savings from our sliced window alignment solution. To avoid such scenario, we propose a strategy that maps a common sliced window plan to physical files in HDFS. The decision chooses the most effective method of representing the slices according to a predefined minimal granularity. The following two cases show the options that Helix adaptively chooses from.

Note that the complexity of the pre-processing of the input files to create the panes would depend on the properties of the input files, e.g., sorted or unsorted, and the granularity of the pane sizes to create. For example, if the pane sizes are larger than the input files or the input files are sorted based on the records' timestamps, then the pre-processing involves only scanning the files to create the panes. Otherwise, it will involve a time-based partitioning to divide the records into the appropriate panes.

First, one slice corresponds to exactly one physical file (depicted in Figure 9.3(a)). Depending on the chunk size on HDFS (e.g., default size 64MB), this file may have one or more chunks. On the other hand, multiple slices together may correspond to one physical file (show in Figure 9.3(b). Namely, when the input data rate is not intensive, multiple logical slices are mapped to one physical file by a partition executor that implements the instructions encoded in the common sliced window produced by the SWA approach. The data records are hashed to their corresponding slices based on their timestamps during the loading time. The partition executor piggybacks the slice creation step with the loading step, i.e., while a given input file is being loaded into HDFS, the partition executor partitions the records on-the-fly to the corresponding slices.

We also introduce a special file header to boost performance for locating selected slices in the second case. Specifically, when a single file contains multiple logical slices, the entire file is not always required by an operation. Thus, a special header for such a file



(a) Individual Slice

(b) Merged Slices

**Figure 9.3:** Mapping Slice Decision to Physical Files

is designed to reduce the latency of finding the required logical slices. This is particularly effective when a file contains a large number of slices caused by a relatively low input rate over a given time period. Having such optimization on mapping logical data units to physical files, Helix avoids creating excessively small files in Hadoop.

# 10

# Helix Shared Recurring Query Optimization

In this chapter, we describe how to find a shared execution plan for a given set of recurring queries $Q$. Each of these queries retrieves its inputs in the form of sliced windows described in Chapter 9.3. The shared execution plan should maximally satisfy the SLA associated with each query $q_i \in Q$, i.e., maximizing the sum of the utility score of $Q$. The problem has two dimensions: (1) identifying the sharing groups, i.e., the queries that will be grouped together to share their executions, and (2) identifying the execution order among these groups. What makes our optimization problem challenging is not only that the solution to each of these sub-problems is NP-hard, but also that they are interdependent, as illustrated in the following example.

**Example 10.1** *Assume that the utility score for all three queries $q_1$, $q_2$, and $q_3$ follows a SLA function, the score is 1 if the queries finish before time $T_d$, and 0 otherwise. Assume that $\{\{q_1, q_3\}$,$\{q_2\}\}$ is the best grouping solution with respect to computational savings, e.g., $q_1$ and $q_3$ share a lot of their computations. However, sharing $q_1$ and $q_3$ would result in missing $q_1$'s deadline due to the data-intensive tasks involved in $q_3$. The estimated* *utility score of group $\{q_1, q_3\}$ in turn would be 1. Therefore, $q_2$ would be scheduled ahead of $\{q_1, q_3\}$ because it has shorter execution time compared to group $\{q_1, q_3\}$. This way only $q_1$'s deadline will be missed and hence the total utility score of executing all three queries would be 2. On the contrary, if we choose to share $q_1$ and $q_2$ together (although their amount of sharing can be small), then the utility score of this group will be 2 since the execution time can meet the SLA functions from both queries. In this case, $\{q_1, q_2\}$ is scheduled ahead of $q_3$, since $q_3$ has a relatively loose deadline. Therefore, the total utility score would be 3 in total.*

In brief, we may need to change the execution order of query groups according to different grouping decisions in order to achieve a better shared execution plan. In the following, we define the concepts of *shared grouping* and *execution ordering*.

**Definition 10.1 (Shared Grouping)** *Given a set of recurring queries $Q = \{q_1, q_2, \ldots, q_n\}$, a set of query groups $G = \{G_1, G_2, \ldots, G_k\}$, where each $G_i$ is a subset of Q, is called a shared grouping solution $GS$, if it satisfies the following two conditions:*

*(1) $G_i \bigcap G_j = \emptyset$, $\forall i, j : 1 \leq i, j \leq k, i \neq j$;*

*(2) $\bigcup G_i = Q$, namely, the union of all $G_i$ forms the entire set of recurring queries $Q$.*

For example, the grouping $\{\{q_1, q_3\}, \{q_2\}\}$ in Example 10.1 is a valid shared grouping, while the grouping $\{\{q_1, q_3\}, \{q_2, q_3\}\}$ is invalid due to the overlapping $q_3$ between the two query groups. The shared grouping does not yet specify the order of execution among its groups, which will be defined next.

**Definition 10.2 (Execution Order)** *In a shared grouping solution $GS$, the start execution time of $G_i$ is denoted by $t_i^{start}$, and its end execution time is denoted by $t_i^{finish}$. We assume that each query group will use all of the available resources to finish as early as possible. Therefore, a valid execution order, denoted as $EO = \langle G_i \rightarrow G_j \rightarrow \ldots \rightarrow G_x \rangle$, is a sequential ordering for $G_i \in GS$.*

For example, $\langle\{q_1, q_3\} \rightarrow \{q_2\}\rangle$ or $\langle\{q_2\} \rightarrow \{q_1, q_3\}\rangle$ are both valid execution orderings.

*Problem Complexity.* Sharing among $n$ map-reduce jobs without taking into account the recurring query constraints has been shown to be an NP-hard problem [38, 40]. Adding the execution ordering problem of the shared query groups to the optimization problem introduces a second dimension, which turns the shared execution of recurring workloads into a bilinear optimization problem. As we illustrated in Example 10.1, if we change which queries to share, then this also affects the overall ordering of shared query execution, and vice versa. The bilinear nature of the problem renders exhaustive techniques, like brute-force and greedy optimization, infeasible. Solving the ordering problem independently of the sharing problem will result in sub-optimal solutions for this bilinear problem. In essence, the shared execution of recurring workloads problem is equivalent to the stochastic knapsack problem [88] with uncertain weights, where the stochasticity comes from the fact that the utility score of a query is variable and depends on all previous decisions.

Worse yet, in order to decide on the best execution ordering within a shared grouping solution, accurate progress estimations for map-reduce jobs are required to estimate when each group ends, and thus compute its expected utility score. Progress estimation in MapReduce is in itself a challenging task due to the factors of distributed processing, concurrency, failures, data skew, and other issues. This problem has received relatively limited attention, e.g., ParaTimer [89], which attempted to estimate the progress of ad-hoc map-reduce jobs.

In the proposed technique, we will explore a branch-and-bound (B&B) optimization strategy that solves a wide variety of combinatorial problems. Conceptually, B&B systematically enumerates a lattice-shaped search space, where each node represents a possible shared grouping. In each node, all possible execution orderings are consid-

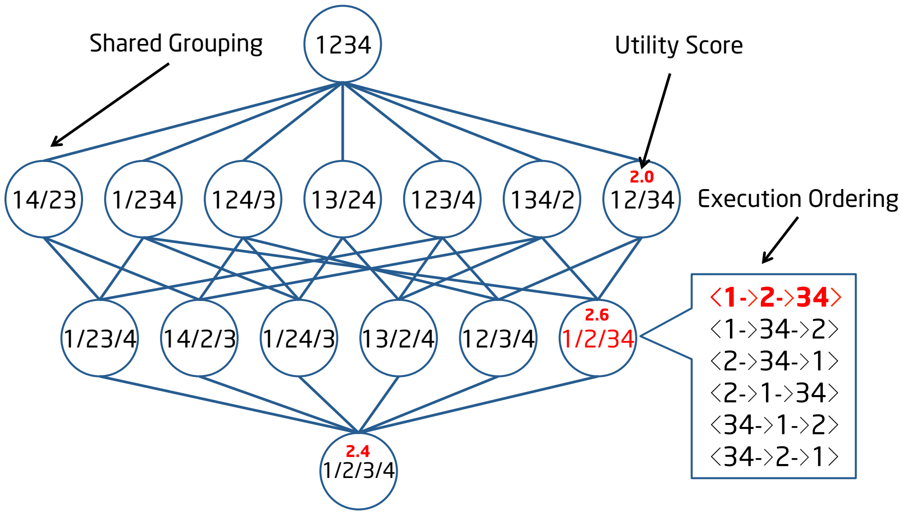


**Figure 10.1:** Lattice-Shaped Search Space

ered. Figure 10.1 demonstrates a lattice-shaped search space for a set of queries of size $n = 4$. Node "1/2/34" corresponds to a shared grouping that consists of three query groups $\{\{1\},\{2\},\{3,4\}\}$. This particular shared grouping is associated with a list of 6 execution orderings, in which $\langle\{1\},\{2\},\{3,4\}\rangle$ is assumed to achieve the highest utility score among 6 different orderings. This score, 2.6 in this case, is attached to the node "1/2/34". Clearly, a brute-force searching algorithm for the entire space is prohibitively expensive. Our proposed B&B algorithm efficiently traverses this search space using two strategies as pruning functions to effectively discard the sub-optimal candidates at the sharing group level and the execution order level, respectively.

## 10.1 Optimization Strategies

In the following subsections, we present two strategies that help pruning the exponential search space and reaching a solution in a practical way.

### 10.1.1 Sharing Strategy

The goal of the proposed sharing strategy is to efficiently produce a good shared grouping solution $SG$. This solution can then be used as a bound to evaluate how good other candidates are and to prune sub-optimal candidates as early as possible.

The sharing strategy first identifies all query groups from the given set of recurring queries $Q$. Each group is associated with a weight which represents the benefit of exploiting the sharing techniques described in Chapter 2.2 versus not exploiting them w.r.t the utility score gain. The benefit value $bVal$ is obtained as follows:

$$bVal = \frac{\vartheta(t_{shared})}{Cost_{shared}(G_i)} - \frac{\sum_{j=1}^{|G_i|} \vartheta(t_j)}{\sum_{j=1}^{|G_i|} Cost(q_j)} \tag{10.1}$$

where $Cost_{shared}(G_i)$ and $\sum_{j=1}^{|G_i|} Cost(q_j)$ denote the estimated costs of executing shared query group $G_i$ and the total costs of executing each query in $G_i$ in isolation, respectively. $\vartheta(t_{shared})$ and $\sum_{j=1}^{|G_i|} \vartheta(t_j)$ denote the utility score of the shared execution of $G_i$, i.e., all queries in $G_i$ end at time $t_{shared}$ and the total utility score by running these queries in a non-shared fashion. The intuition behind Equation 10.1 is to show the utility benefit per unit of cost between the shared and non-shared executions. A higher $bVal$ indicates that the sharing in $G_i$ is rewarding and should be given a higher priority. To avoid reinventing the wheel, we exploit the cost model established in [40] to obtain the costs of shared execution of $G_i$. This cost model takes into consideration all MapReduce sharing techniques to estimate the costs.

Conceptually, the next step is to form all possible shared groupings based on the identified query groups, calculate the total $bVal$ of each query group, until we select the shared grouping with the largest total $bVal$ as the final solution. Given Definition 10.1, and the benefit model of each candidate, our shared grouping problem can be mapped to a well-known graph problem, i.e., finding the Maximum Independent Set.

**Definition 10.3 (Problem Mapping)** *Given a set of all possible query groups, we define an undirected graph $G = (V, E)$, where $v_i$ denotes a query group, and an edge $e(v_i, v_j)$ denotes that $v_i$ and $v_j$ do not meet Definition 10.1. Now, our goal is to find a maximum independent vertex set $V_i$, among all possible $V_i \subset V$, where no vertices in $V_i$ are connected, with the largest overall $bVal$, i.e., $max_{Vi}(\sum_{v_i \in V_i} bVal(v_i))$ with $\bigcup V_i = Q$.*

The maximum independent set is known to be a NP-hard problem [90]. Clearly, it is prohibitively expensive to create all possible shared groupings and to select the one with largest total $bVal$. Therefore, we now propose a greedy search strategy to find a good shared grouping solution (shown in Algorithm 6).

**Algorithm 6** *SharedGrouping()* Algorithm

```
Input: A set of query groups $G$
Output: A shared grouping solution $sol$
for $G_i$:$G$ do
    if $G_i.bVal < 0$ then
        $G$.remove($G_i$) // remove groups with negative benefits
sort($G$) // sort groups by $G_i.bVal$
$sol \leftarrow \phi$
for $G_i$:$G$ do
    if !isOverlapping($G_i$, $sol$) then
        $sol$.add($G_i$) // add query group $G_i$ to the solution
return $sol$
```

The time complexity of Algorithm 6 depends on the number of query groups $|G|$. Suppose there are $n$ candidates, the sorting function can finish in $O(nlogn)$ time. The time complexity of conflict check depends on the size of $sol$ set, which is at most $n$. Thus, the upper bound complexity of conflict check for $n$ candidates is $O(n^2)$. However, the solution set $sol$ would not be large in practical due to the sharing conflict check as discussed above. Thus the worst case time complexity of our greedy algorithm is $O(n^2)$.

The solution produced by Algorithm 6 serves as an initial and reasonable guideline about which queries should be shared. The overall SLA satisfaction of such shared grouping solution serves as a bound for B&B algorithm as well. More specifically, any other shared grouping solutions should have better or at least identical SLA satisfaction in order to be considered as a final solution for our shared execution problem. In Chapter 11.3, our experimental results illustrate that this strategy effectively helps the B&B algorithm find the optimized shared execution plan for given recurring workloads.

### 10.1.2 Ordering Strategy

In addition to the sharing strategy, our Helix optimizer also uses the ordering strategy. Based on the shared grouping (e.g., Node {{1},{2},{3,4}} in Figure 10.1) produced by Algorithm 6, we make a decision on the global execution ordering $EO$ of its groups (e.g., $\langle\{q_1\} \rightarrow \{q_2\} \rightarrow \{q_3, q_4\}\rangle$ in Figure 10.1). If the global execution ordering $EO$ based on the given shared groups is found to be sub-optimal (described in Section 10.1.4), then the Helix's optimizer will repeatedly request other shared groupings. The ordering solution for a given shared grouping can be easily computed, provided that the utility scores of all recurring queries are known at given time $t$. This is a big advantage of recurring queries that Helix will make use of. In the recurring workload context, all queries periodically execute over the same evolving data sources. Thus, Helix deploys a monitoring and profiling techniques that tracks the consecutive execution of each recurring query, and builds a profile for each query. The profile contains statistics for each execution, e.g., the execution time, the amount of data processed, the number of mapper and reducers used. According to these statistics, the estimation accuracy of the current execution as well as the associated utility score can be gradually improved.

The ordering strategy exploits the observation that if a sequence of recurring queries or query groups are to be executed, executing the one with the higher utility score first and also the lower execution costs first likely results in higher global utility scores. The benefits come mainly from the fact that choosing such query can return a higher utility score per unit of cost, $\vartheta_i(t_0)/Cost(q_i)$, in which $\vartheta_i(t_0)$ is the initial utility score of $q_i$. In order to efficiently produce the solution, we exploit greedy approximation algorithm to solve the relaxed problem of stochastic knapsack problem [88]. In this case, all queries and query groups are sorted in decreasing order of utility score per unit of cost ratio. The ordering strategy then computes the total utility score achieved by this order. For instance, if $q_1$, $q_2$, and $q_3$ have exactly the same costs, but their utility scores at a given time $t$ are 1, 0.2, and 0.6, respectively, then the ordering should be $q_1$, and then $q_3$ followed by $q_2$.

The ordering strategy is not reducing the solution space by omitting sub-optimal decisions, but instead it gives us a greedy direction on how to start exploring the solution space.

### 10.1.3 Helix Algorithm

Now we present our Helix optimization algorithm. The goal is to produce a globally efficient shared execution plan, given a recurring workload $Q$ with resources $R$. Before presenting the algorithm, we will discuss how to use the solution produced from the above two strategies as bounds to safely prune the sub-optimal candidates.

### 10.1.4 Pruning in B&B

As already explained in the previous two strategies, we have two initial bounds for the B $\&$ B method, a bound for sharing groups and a bound for execution ordering. Given that both bounds can be produced by fast run-time algorithms, they can be used as effective approximations towards the final solution. The key idea of the proposed B&B algorithm is that if the initial bounds are better than the upper bound (optimistic bound) for the current candidate under consideration, then this candidate—and all its sub-solutions—can be safely discarded from the search. If a candidate has a higher utility score compared to the initial bounds, then they will be replaced by this new candidate. Hence, these two bounds record the minimum upper bound seen among all candidates examined so far. Next, we introduce two lemmas that guarantee the safe pruning for the B&B method. Thus the B&B method is guaranteed to find the optimal solution of shared execution of recurring workloads.

**Lemma 10.1** *Given an execution ordering of a subset of query groups $G_i \in SG$ and the utility score associated with this ordering (denoted as $uScore_{G_i}$), and the rest query groups $G_j \in SG$, if the highest utility score of $G_j$ (denoted as $uScore_{G_j}$) plus $uScore_{G_i}$ is less than the utility score of the ordering bound $uScore_{OB}$, then the candidates constructed by combining the existing ordering of $G_i$ and all permutations of the execution ordering of $G_j$ can be pruned safely.*

*Proof:* Due to the monotonic decreasing property of SLA functions, for any recurring query, we have

$$\vartheta(t_i) \geq \vartheta(t_j), (t_i < t_j) \tag{10.2}$$

where $t_i$ and $t_j$ are two time points. Hence the highest utility score of a query $q_i$ that starts its execution at time $t$ can be:

$$uScore^H(q_x, \vartheta) = \vartheta(t + e) \tag{10.3}$$

where $e$ is the estimated execution time of $q_i$. Therefore, $uScore_{G_j}$ at time $t$ is

$$uScore_{G_j} = \sum_{x}^{|G_j|} uScore^H(q_x, \vartheta) \tag{10.4}$$

where $|G_j|$ is the number of remaining query groups not yet inserted into order. Here we assume that all remaining query groups can be executed concurrently at time $t$. Any other execution orderings of the remaining query groups delays the execution of at least one query group, which in turn reduces the total utility score. Finally, if $uScore_{G_i}$ + $uScore_{G_j} \leq uScore_{OB}$, all possible execution orderings formed by the remaining query groups can be safely pruned. ■

**Example 10.2** *Assume that we have 5 recurring queries and the query groups are* $\{q_1, q_2\}$, $\{q_3\}$, $\{q_4\}$, *and* $\{q_5\}$. *We further assume that the highest utility score seen so far is 4.2 (i.e., the ordering bound). Assume that the utility score of the execution ordering of* $\langle\{q_3\} \rightarrow \{q_4\}\rangle$ *is 1, and the highest utility score of the other two query groups, i.e., assuming they will execute at the same time, is 3. Then the highest utility score achievable (1+3=4) is still less than the ordering bound (4.2). In this case, we do not need to examine the detailed execution ordering among* $\{q_1, q_2\}$ *and* $\{q_5\}$, *and all shared groupings resulted from their different execution ordering, i.e.,* $\langle\{q_3\} \rightarrow \{q_4\} \rightarrow \{q_1, q_2\} \rightarrow \{q_5\}\rangle$ *and* $\langle\{q_3\} \rightarrow \{q_4\} \rightarrow \{q_5\} \rightarrow \{q_1, q_2\}\rangle$, *can be pruned safely.*

**Lemma 10.2** *Given a shared query group, if the utility score of this query group at time* $t$ *is less than the one of executing all queries in this group in a sequential order, all candidates that contain this query group can be safely pruned.*

*Proof:* The Lemma 10.2 is self-explanatory. Let $uScore_{G_i}^{share}$ denote the utility score of the shared query group. Let $uScore_{G_i}^{nshare}$ denote the utility score of all queries running separately in a sequential order. Let $uScore_{Q-G_i}$ denote the utility score that could possibly be achieved from the remaining queries. If $uScore_{G_i}^{share} < uScore_{G_i}^{nshare}$, we have

$$uScore_{G_i}^{share} + uScore_{Q-G_i} < uScore_{G_i}^{nshare} + uScore_{Q-G_i} \tag{10.5}$$

Thus the query grouping candidates consisting of this shared query group can never achieve higher utility score than the ones without it. Thus, the candidates that contain this shared query group can be safely pruned. ■

**Example 10.3** *Assume that we have 4 recurring queries $q_1$, $q_2$, $q_3$, and $q_4$. If the utility score of a shared query group $\{q_1, q_2\}$ is less than executing $q_1$ and $q_2$ in a sequential order, grouping $q_1$ and $q_2$ together is not beneficial with respect to the utility score, even they may have significant computational savings. Any solution candidates with group $\{q_1, q_2\}$ should be pruned from consideration.*

### 10.1.5 Branch and Bound Algorithm

Now we present our B&B method for determining shared execution plans of recurring workloads. The algorithm uses "nodes" to keep intermediate states. There are three types of nodes: the solution node, the live nodes and the dead nodes. A solution node contains a solution to the problem with highest utility score seen so far. The score assigned to a solution node is computed directly from the SLA functions. The algorithm may change the solution node as it explores the solution space. The solution with the highest score is the output of the algorithm.

Live nodes contain possible solution candidates to our problem and they are connected with other nodes. Once visited without being changed into a solution node, a live node is turned into a dead node, meaning that we do not have to visit it again. In order to calculate the utility score of a live node, we plug in the estimated execution time into the SLA functions associated with the queries in this node, and the above two bounds to estimate the remaining part of the solution. A feature of Branch and Bound is that once we have reached a solution node, we can prune all live nodes that have a score lower than the score of the solution node. Recall that a live node has an estimated utility score (an upper bound score). Therefore, pruning these live nodes does not affect the optimality of the algorithm because the score of a live node means that as we explore this node and fully traverse all children, we will never reach a solution with a higher utility score. In other words, the score of a live node is the theoretical bound of the sub-tree of nodes.

The algorithm, described in Algorithm 7, uses a heap to maintain the set of live nodes sorted by their scores. The first node that enters the heap is the root node, a node that contains the solution produced based on our two strategies described in Section 10.1. The algorithm proceeds by removing the first node of the heap, and testing if it is a better solution or not. In case it is a solution that has a higher utility score than any solution we have seen before, we keep it. In the case that the active node is not a solution, all its child nodes are inserted into the heap. As already explained, we can prune a node and its corresponding sub-tree if it meets the pruning conditions we introduced in Section 10.1.4.

**Algorithm 7** Helix Optimizer Algorithm

```
Input: Query Set $Q$, Heap $heap$, Node $root$, Node $tmp$
Output: Node $solution$
$solution \leftarrow$ SharedGrouping($Q$)  // call sharing algorithm
$solution.score \leftarrow$ order($solution$)  // call ordering algorithm
Push($heap, root$)
while !isEmpty($heap$) do
    $tmp \leftarrow$ Pop($heap$)
    $tmp.score \leftarrow$ order(tmp)
    if $tmp.score > solution.score$ then
        $solution \leftarrow tmp$
        Node[] $children \leftarrow childrenOf(tmp)$
        for Node $c \in children$ do
            $c.score \leftarrow$ order(c)
            Push($heap, c$)
    else if !groupPrune($tmp.score$) then
        Node[] $children \leftarrow childrenOf(tmp)$
        for Node $c \in children$ do
            $c.score \leftarrow$ order(c)
            Push($heap, c$)
return $solution$
```

Figure 10.1 shows an example of how Algorithm 3 explores the search space by traversing nodes. The root node at the bottom of the lattice-shape space holds a set of recurring queries with their optimal execution ordering (i.e., $\langle\{1\} \rightarrow \{2\} \rightarrow \{3\} \rightarrow \{4\}\rangle$). The value of the node is its estimated utility score. In the example of Figure 10.1, the root node has six child nodes. These child nodes have their query grouping fixed, meaning that this part of the solution will remain constant in their child nodes. For instance, a group of queries {3,4} is part of the query groups {{1,2},{3,4}} and {{{1},{2},{3,4}}}. During node traversal all possible sets of groups must be taken into account. In our example the child node of the root with highest utility score is traversed in the decreasing order of their utility scores. Finally, at this point we should notice the importance of the sharing and ordering strategies. Without these two strategies, node {{1},{2},{3,4}} has three child nodes that contains all possible groupings having {3,4} in the same group. The optimizer would have had to consider them all and their child nodes as well, for all possible execution orders of a total of 4 nodes (6+2+2+2+1 = 13 in this case). Given the node {{1},{2},{3,4}} fails to meet our grouping bound, it can be safely pruned so that only its siblings and their child nodes will be traversed.

# 11

# Experimental Evaluation

In this chapter, we describe the experimental study we conducted to evaluate Helix. We will show that: (1) Helix effectively supports shared execution of recurring queries by employing sliced window alignment techniques, (2) Helix maximally satisfies the SLA requirements specified in recurring queries, and (3) the effectiveness of Helix's optimization algorithm of finding the best shared execution plan for a given set of recurring queries.

## 11.1 Experimental Setup & Methodology

*Experiment Infrastructure.* All experiments are conducted on a shared-nothing cluster with one master node and 40 slave nodes. Each node consists of 16 core AMD 3.0GHz processors, 32GB RAM, 250GB disk, and nodes are interconnected with 1Gbps Ethernet. Each server runs CentOS Linux (kernel version 2.6.32), Java 1.6, Hadoop 0.20.1. Each node is configured to run up to 8 map and 8 reduce tasks concurrently. The sort buffer size was set to 512MB, and speculative execution was disabled to boost performance. The replication factor is set to 3 unless stated otherwise.

*Datasets and Queries.* We use two real-life datasets for our experiments. The World

Cup Click dataset [77] (256GB) contains records of more than 1.35 billion web requests made to 1998 World Cup Website. Another dataset is the latest high volume Wikipedia database [78] (400GB) being modified and updated continuously.

We focus on queries that involve join, project, and aggregate operations, which are fundamental operations not only in relational databases, but also in the emerging data analytics tasks described in Chapter 1.2. These queries were generated from the following query template: *select S1.T, sum(value) from S1, S2 where S1.a = S2.b group by S1.T*, where $T$ is a randomly selected list of dimensional attributes. The default number of queries in a query batch was 20 unless otherwise stated. Each query is also assigned a query deadline that ranges from [0.3 - 1] of the query's window size $w$. These 20 queries' deadlines are uniformly distributed within this range. We execute each experiment three times. In the charts, we report their average results.

*Metrics & Measurements.* Given a set of recurring queries, we measure the utility score of each query, and the average execution time for the recurrences of each query. The execution time of recurring queries is a common metric in data management systems, while the utility score is defined in Chapter 2.4. We do not include the data pre-processing time, since it is performed on-the-fly during the loading time. It is hence negligible compared to the disk-based query processing in Hadoop. We verify Helix's effectiveness under different time-based utility functions. Table 11.1 summarizes the utility functions used in this study, which are also commonly used to specify SLA requirements [91]. We also evaluate the optimization overhead incurred by our optimizer with respect to its optimization time.

*Methodology.* We adopted the state-of-the-art method [64] to support single recurring query processing, and implemented the proposed Helix techniques for sharing execution of recurring workloads on top of the extended open-source Apache Hadoop. We compare three algorithms denoted Redoop, GGTMT, and Helix, respectively. Redoop [64] is the

| # | **Utility Functions** |
|---|---|
| F1 | $\theta_{F1}(t) = \begin{cases} 1 & for \quad t \leq t_d \\ 0 & for \quad t > t_d \end{cases}$ |
| F2 | $\theta_{F2}(t)$ = 1/log($t$) |
| F3 | $\theta_{F3}(t) = \begin{cases} 1 & for \quad t \leq t_d \\ 1/(t - t_d) & for \quad t > t_d \end{cases}$ |

**Table 11.1:** Utility Functions Used in the Experimental Study

state-or-the-art approach for evaluating a recurring query in MapReduce. GGTMT [40] only maximizes computational saving by combining both GGT (general grouping technique) and MT (materialization technique), without taking into consideration the SLA satisfactions. Regarding the efficiency of the Helix optimization algorithm, we compare it with the exhaustive (EXH) and random search (i.e., simulated annealing) algorithms (RAND).

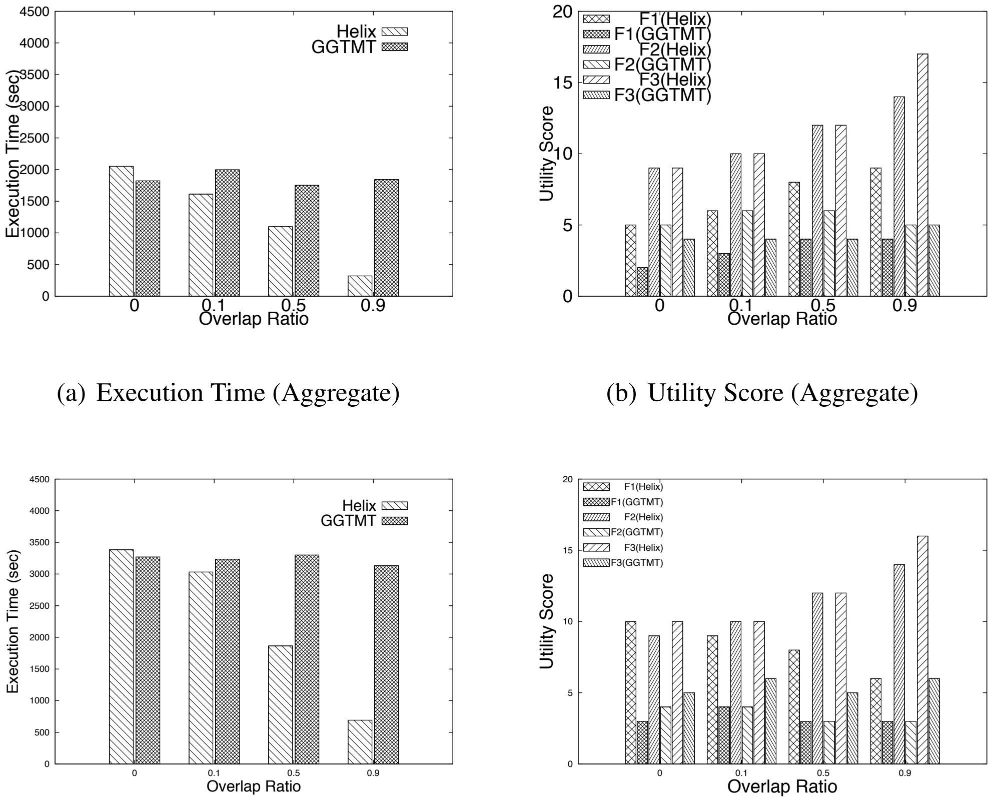
## 11.2 Helix Runtime Performance

We evaluate the effectiveness of our Helix solution from three perspectives. First, we demonstrate the effectiveness of Helix's sliced window alignment technique by comparing with GGTMT as it only targets on sharing techniques for ad-hoc queries over static datasets in MapReduce. Second, we demonstrate the sharing benefits gained by our Helix optimizer by comparing against Redoop as it aims to optimize for a single recurring query. Third, we evaluate the scalability of our Helix solution by varying four parameters, i.e., data size, number of queries, and cluster size.

### 11.2.1 Effectiveness of Sliced Window Alignment

Figure 11.1 illustrates the improvement of Helix over GGTMT. In this experiment, we fix the number of queries to 20 and the number of nodes to 40. The size of the dataset is 240

GB for each slide in the common sliced window. We vary a factor, called $overlap$, which corresponds to the amount of overlapping data between two consecutive windows of each query, to measure the effectiveness of Helix sliced window alignment technique.



(a) Execution Time (Aggregate)

(b) Utility Score (Aggregate)

(c) Execution Time (Join)

(d) Utility Score (Join)

**Figure 11.1:** Effectiveness of Sliced Window Alignment

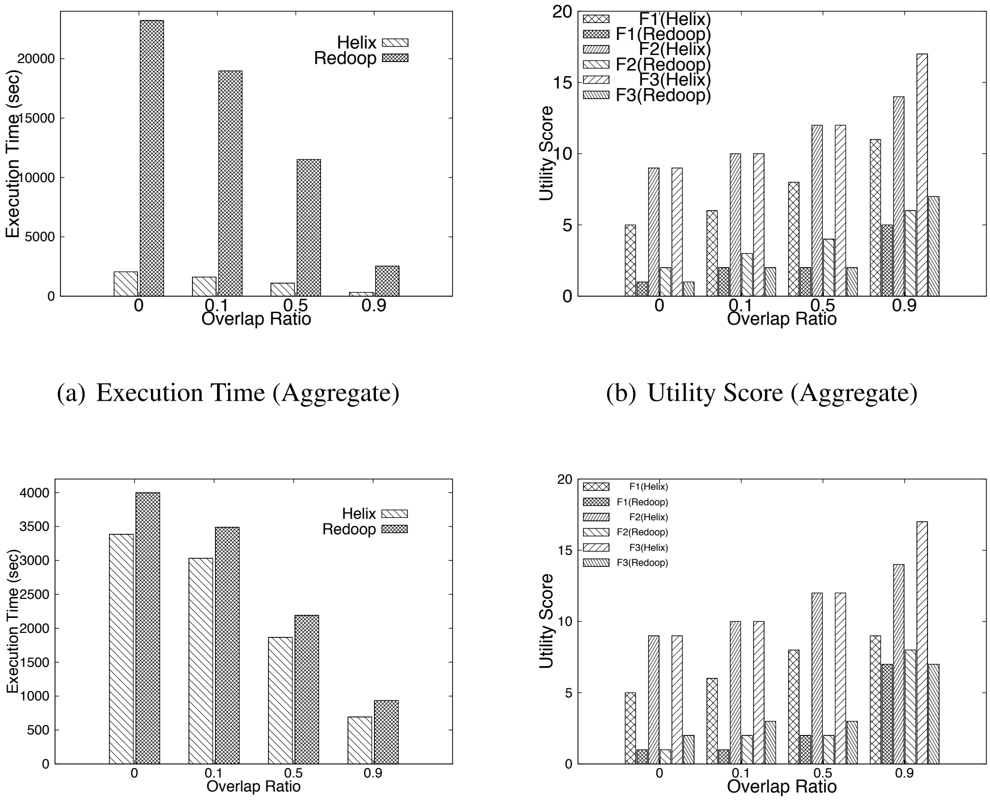
Helix's sliced window alignment technique substantially reduces the execution time while also increasing the SLA satisfaction as illustrated in Figure 11.1(a). Helix benefits from the sliced window alignment to avoid unnecessary data processing, resulting in a significant advantage over GGTMT. Helix processes the newly arriving data in a finer granularity. Moreover, the sliced window opens more sharing opportunities compared to the GGTMT technique. Thus the execution time can be further reduced by up to 83%

when $overlap$ is 0.9. The result in Figure 11.1(b) show that Helix has a clear advantage over GGTMT with respect to the SLA satisfaction. Slicing a window into small chunks breaks the original query into multiple MapReduce jobs, which in turn provides more flexibility in choosing the appropriate queries to meet SLA requirements. The execution time and utility score of the join operation demonstrate a similar trend to the above aggregation operation.

### 11.2.2 Effectiveness of Shared Execution

In this sharing benefit experiment, we also use 20 queries, 40 nodes, and 240 GB data sets for each query recurrence. We again vary the $overlap$ to evaluate the effectiveness of sharing techniques on multiple recurring query processing. Helix's sharing techniques significantly improve the performance in both metrics (i.e., the execution time and the utility score). The execution time of Helix is up to a magnitude faster compared to Redoop when there is no overlap. The trend changes when the $overlap$ ratio is high (0.9). In this case, Redoop system can exploit the reduce input cache which is equivalent to our reduce input sharing in Helix. The savings gained from the reduce input caches contribute to the total savings more than other sharing opportunities such as map input scan or map output sharing. However, when $overlap$ ratio decreases, the benefits of the other sharing techniques become significant. The total savings gained from all sharing techniques make the performance of our Helix system substantially exceeds that of the Redoop system.

The results in Figure 11.2(b) illustrate the improvement of Helix over Redoop. In all cases, even when $overlap$ is 0, our Helix system achieves much better SLA satisfaction compared to Redoop. The reason is that Helix's shared execution plan is optimized for maximizing the SLA satisfaction. On the contrary, Redoop system is neither equipped with as many sharing techniques as Helix nor aware of SLA requirements associated with recurring queries. As expected, the results of the join operation verify the superiority of



(a) Execution Time (Aggregate)

(b) Utility Score (Aggregate)

(c) Execution Time (Join)

(d) Utility Score (Join)

**Figure 11.2:** Significance of Sharing Benefits

our Helix solution over Redoop in both metrics.

### 11.2.3 Putting It All Together

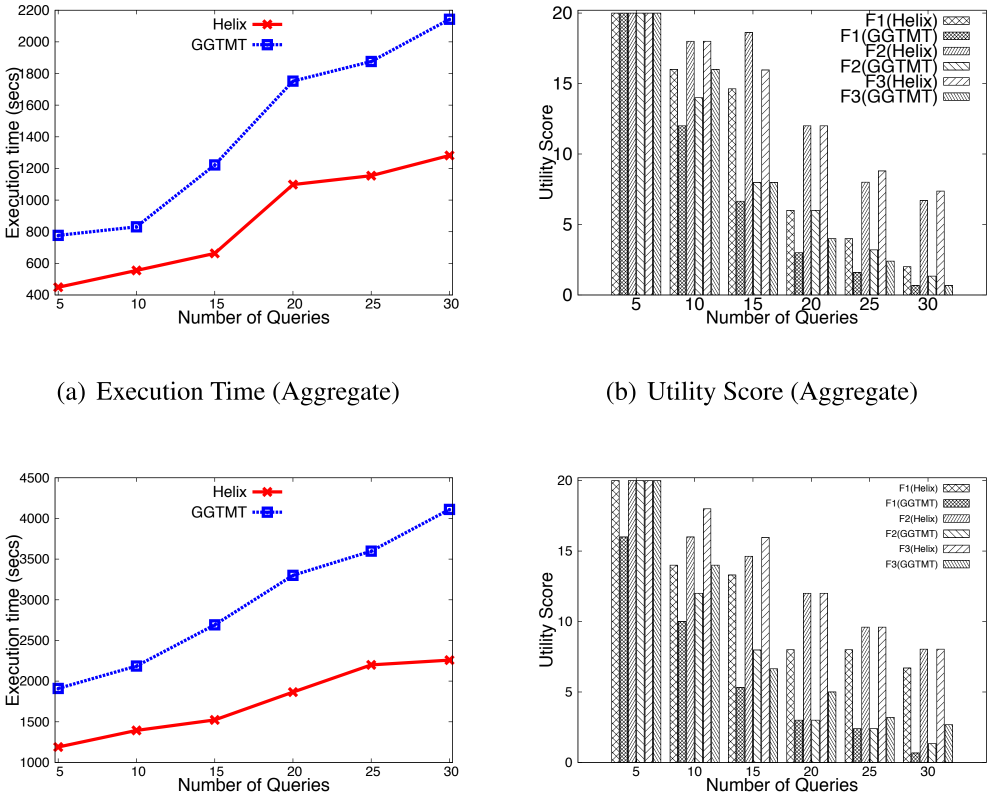
In this set of experiments, we evaluate the effectiveness and scalability of our Helix solution by varying three parameters, i.e., data size, number of queries and cluster size. Figures 11.3, 11.4, and 11.5 show the experimental results of Helix over GGTMT indicated. Each figure shows the aggregation operations with respect to two metrics (the execution time and the utility score).

*Effect of number of queries.* Figure 11.3 compares the performance as the size of a query batch is increased. We observe that our algorithm significantly outperforms GGTMT. For example, Helix outperforms GGTMT by 178% on average and up to 204% with respect to the execution time when the number queries is 30. Further more, as the number of queries increases, the winning margin of our solution over GGTMT also increases. This is expected as the conflicts between sharing opportunities and SLA requirements across queries also increase with the number of queries.

Regarding the utility score, the advantage of Helix is more obvious compared to GGTMT. In Figure 11.3(b), The total utility score achieved by Helix is up to 1089% more than the one achieved by GGTMT. The winning margin is similar to the observation in Figures 11.3(a). The reason is that Helix optimization is SLA-oriented. When the number of queries is small, Helix and GGTMT may happen to produce an identical shared execution plan for given queries due to the relatively small search space. However, when the number of queries increases, the solution produced by Helix is optimal with respect to the SLA requirements whereas GGTMT tries to maximize computational savings only.

*Effect of cluster size.* Figure 11.4 compares the scalability of all methods by varying the number of nodes in the cluster used. Here again our Helix solution significantly outperforms GGTMT in the execution time. For example, Helix outperforms GGTMT by 202% when the number of nodes is 10. Helix outperforms GGTMT by 232% when the number of nodes is 40. Moreover, the improvement factor of Helix over GGTMT does not show significant differences for both aggregation and join operations.

Furthermore, as the cluster size increases, the running time for both approaches decreases. In particular, the running time of Helix decreases much faster than the one of GGTMT solution which therefore enlarges the winning margin as cluster size increases. Thus, the performance improvement from the increased parallelism using a larger cluster benefits Helix more than the GGTMT technique. The reason is that Helix system's sliced



(a) Execution Time (Aggregate)

(b) Utility Score (Aggregate)

(c) Execution Time (Join)

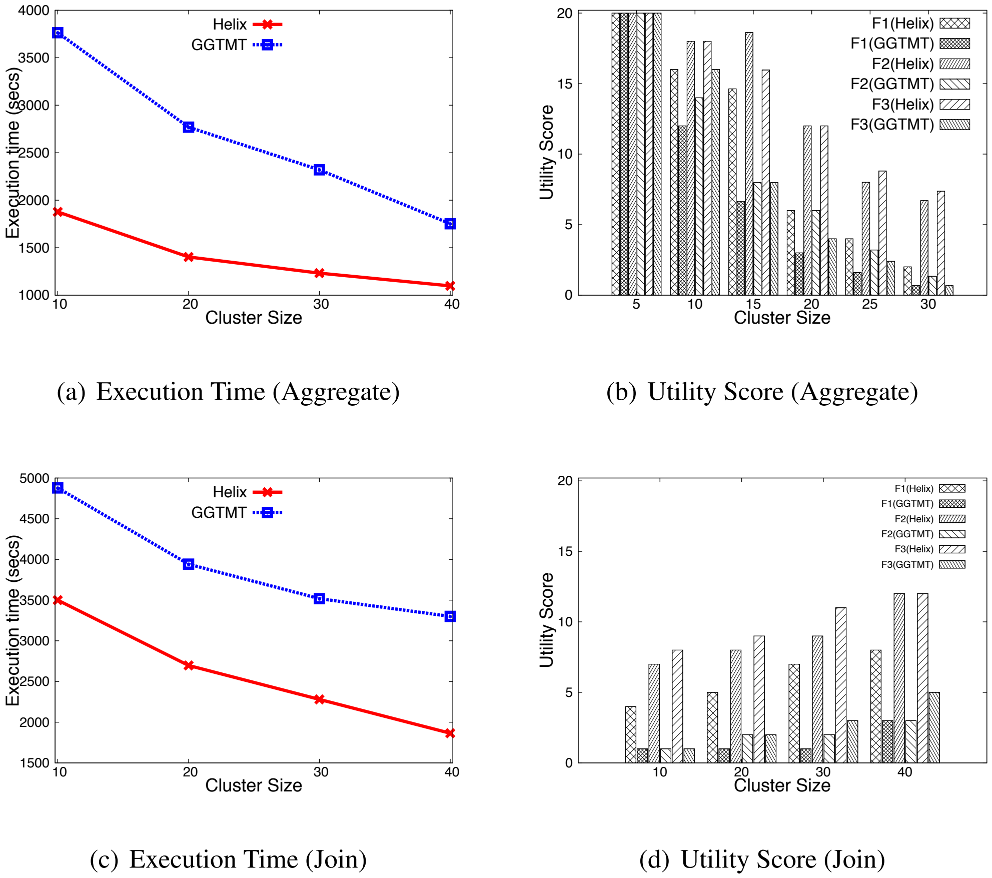
(d) Utility Score (Join)

**Figure 11.3:** Varying Number of Queries

window alignment technique provides appropriate sized inputs. That enables Helix to make full use of the parallelism with more nodes are available.

In Figure 11.4(b), the utility score results confirm the superiority of our Helix approach over GGTMT as well. Helix is able to win in all cases. The winning margin becomes more substantial when the number of nodes increases.

*Effect of data size.* Figure 11.5(a) examine the execution times of Helix with different data sizes per job ranging from 80GB to 400GB. Helix significantly outperforms GGTMT again. For example, Helix outperforms GGTMT by up to 128% when the data size is 400GB for aggregation queries. This is verified by the increase of the execution



(a) Execution Time (Aggregate)

(b) Utility Score (Aggregate)

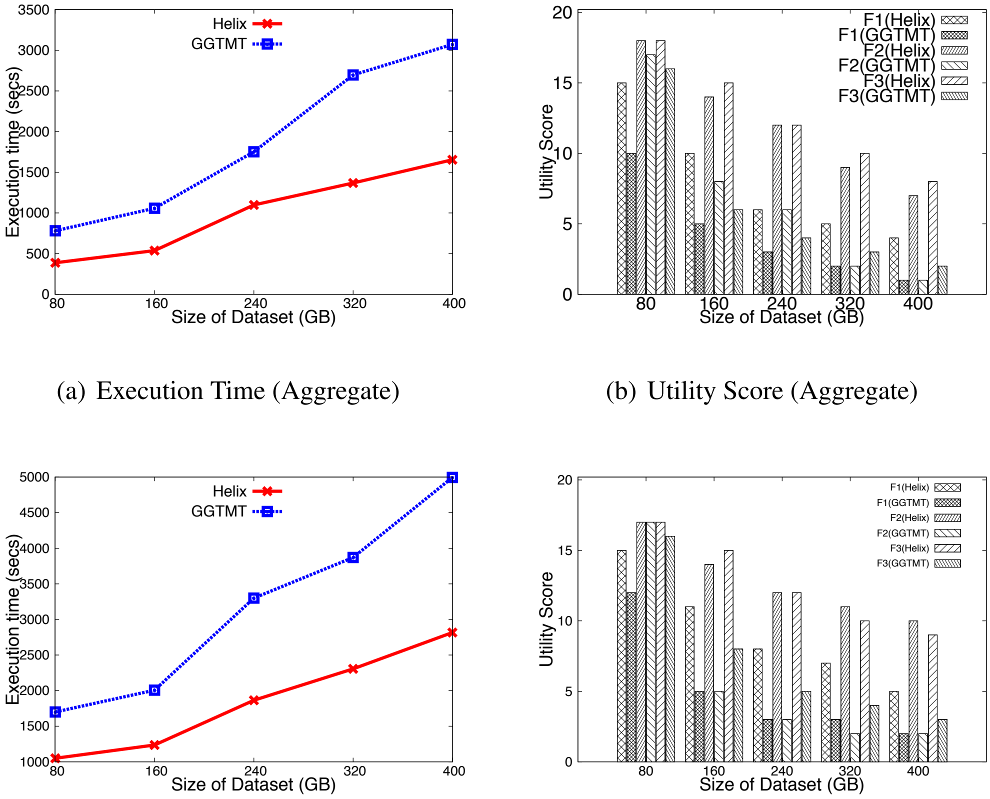
(c) Execution Time (Join)

(d) Utility Score (Join)

**Figure 11.4:** Varying Number of Nodes

time for both approaches as the data size increases. In particular, the running time of GGTMT increases much faster than for the Helix approach. This shows that the robustness of our Helix approach against varying data size. The reason behind this is that by exploiting the sliced window alignment technique and the sequential sharing method, the Helix approach can significantly reduce or even eliminate the unnecessary data recomputations.

The results in utility score metric, as depicted in Figure 11.5(b), prove the success of Helix approach as well. The reason behind this is that the decision of the execution ordering becomes more critical when the data size increases. Making an inappropriate ordering decision may significantly reduce the utility scores of other queries as they are blocked from processing for a long time. The solution produced by Helix can always guarantee the optimality of the execution order of the shared query groups. In other words, most useful and urgent queries are processed ahead of other queries in a shared fashion. This greatly helps the system to maximally satisfy the SLA requirements associated with the queries.



(a) Execution Time (Aggregate)

(b) Utility Score (Aggregate)

(c) Execution Time (Join)

(d) Utility Score (Join)

**Figure 11.5:** Varying Size of Dataset

## 11.3 Efficiency of Helix Optimizer

In this section, we evaluate the efficiency of our Helix's B&B algorithm by comparing against two extreme solutions: a brute-force algorithm that generates the optimal sharing solution (denoted by EXH) and a random approach using the simulated annealing algorithm (denoted by RAND). We measure the optimization times to evaluate query batches of different sizes. We choose not to evaluate the quality of solutions produced by all three methods, because Helix always produces the optimal solution as EXH approach does. While the RAND methods often chooses a sub-optimal solution because it might be stuck in a local optima.

Figure 11.6 shows the optimization times of the three methods with varying numbers of queries. For each query size, our Helix approach outperforms both the exhaustive and random approaches by up to 7 and 2 times, respectively. When the number of queries is small (10 and 15), all three methods feature a similar optimization time. As expected, EXH is not a scalable solution when the number of queries increases. On the contrary, the Helix method achieves the same quality shared execution plan as EXH but does so in a much smaller optimization time. The random method RAND does not suffer from a significant increase of optimization time for large numbers of queries. However, it cannot guarantee any optimality regarding the produced solution.

In summary, the Helix approach guarantees to produce an optimal solution for shared execution of recurring workloads with negligible optimization time overhead.

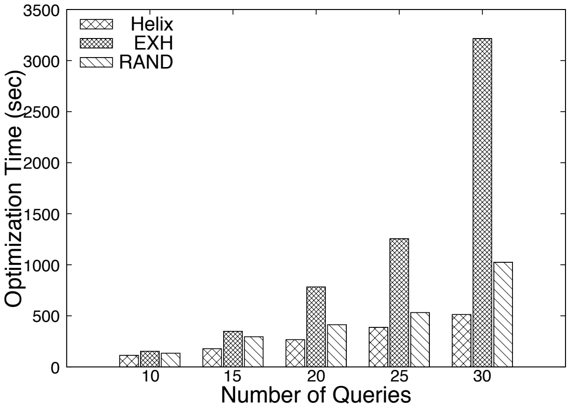


**Figure 11.6:** Comparison of Optimization Methods

# 12

# Related Work

*Recurring Query Processing.* Recurring query processing systems [63, 64, 92] have been proposed to support large-scale data analytics applications over evolving data streams. SCOPE [63, 92] handles recurring queries by instrumenting queries to piggyback statistics collection with its normal execution. Collecting such statistics makes it possible to create a statistical profile that can be fed to the optimizer on a future invocation of the same job. Redoop [64] employs window-aware optimization techniques for recurring query execution including adaptive data partitioning, window-aware task scheduling and inter-window caching. However, none of the above systems support the optimization of multiple recurring queries.

*Multi-Query Optimization (MQO):* MQO is known to be an efficient method for handling large query workloads in traditional database systems [93, 94, 95]. Techniques have been proposed for tackling MQO problems in relational database systems, in particular, materialized views [93, 94] and common sub-expression sharing [95]. In [94], materialized views are effective techniques for implementing common sharable processing across queries. Chaudhuri et al. [93] proposed to pre-compute views over static data to be used by other subsequent queries. In the Cache-on-Demand (CoD) system [95], the intermediate and final results from existing queries are treated as caches that are usable by future queries. Such principles of keeping data generated from one query in views (i.e., caches) more recently have been leveraged in MapReduce as well as explained below.

**Multi-Query Optimization in Data Stream System:** Continuous query processing perspective has also given considerable attention to multiple query optimization [96, 97, 98]. Krishnamurthy el at. [97] have explored run-time aggregation sharing with varying periodic windows and arbitrary selection predicates. However, our Helix system aims to support more operations (e.g., join) other than the aggregation operator. Song et al. [98] proposed to slice the results of a binary join operation into a chain of fine-grained window instances to allow reusing of results across multiple queries. Nevertheless, it does not consider deadline parameters of each query which may prevent the query from sharing execution even if its window and slide match up with other queries.

*Multi-Query Optimization in MapReduce:* Several techniques have been proposed for sharing or reusing work across multiple queries on MapReduce. MRShare [38] aims to partition a batch of jobs into disjoint sharing groups. Specifically, MRShare combines queries that share similar MapReduce jobs into a group and processes such group as a single MapReduce job. However, MRShare does not support general jobs that use multiple inputs (e.g., joins) nor sharing parts of the map functions. Also MRShare does not support window constraints and SLA requirements specified in recurring queries. Thus the sharing groups produced by MRShare would not provide any guarantee of exploiting unique sharing opportunities in recurring queries and thus satisfying the associated SLA requirements.

The ReStore [49] system manages the storage and reuse of intermediate results produced by MapReduce workflows. ReStore materializes map and/or reduce output of MapReduce jobs to identify reuse opportunities by future jobs, therefore avoiding redundant work. The storage space in ReStore maintains the materialized results according to their use frequency. Our work differs from ReStore in both the problem focus and the developed techniques. The sharing decisions made by our technique are SLA-targeted and thus guarantee immediate reuse of materialized results whereas the materialized output produced by ReStore might not be reused at all. Moreover, our Helix system exploits sharing opportunities by executing a group of queries together as one MapReduce job. In contrast, ReStore does not support such shared query executions.

Wang el at. [40] propose two sharing techniques, the *generalized grouping technique (GGT)* and the *materialization technique (MT)*, to refine multi-query optimization in MapReduce. Multiple jobs could share the scan of the input file as well as the map output. They also design a cost-based two-phase approach to find shared execution plans. Compared with [40], our work focuses on the unique challenges of shared execution of recurring queries and not just ad-hoc jobs on static datasets. Moreover, our work is more comprehensive by targeting additional optimization objectives such as maximizing SLA. This leads to a more complex optimization problem. We propose a novel SLA-driven approach with effective pruning heuristics that exploit the characteristics from both MapReduce jobs and recurring queries to find optimal shared execution plans, which better meet SLA requirements in recurring queries.

*Service Level Agreement:* Meeting certain SLA constraints has been addressed in the context of stream processing systems [99, 100]. Prior work [99] has leveraged specific workload characteristics to meet SLAs without losing efficiency or utilization. To provide real-time responses, [99] enable the user to specify a contract in terms of latency, data freshness, CPU and memory usage. Its main method is to shed data from incoming streams to handle load and meet the desired QoS. Our aim is not load shedding but instead sharing of workloads. The second solution in [100] employs scheduling technique to leverage small, uniform task durations to trade short-term SLA violations for efficiency. However, these workload characteristics are not universal. In contrast, our Helix solution is independent of workload characteristics. Moreover, the scheduling technique in Helix captures not only SLA requirements but also computational savings from shared query executions.

Several techniques [50, 51, 52] have been proposed for achieving SLAs of MapReduce jobs. These methods either dynamically adjust resource allocation or they exploit profiling to help jobs provision resources statically at startup. However, none of these efforts consider sharing such as merging similar jobs into one MapReduce job. Moreover, they do not address any unique challenges derived from targeting recurring queries as done in our work.

# Part III

# Fast Approximate Recurring Queries in MapReduce

# 13

# Faro System Overview

Figure 13.1 depicts the Faro architecture as an extension to the Apache Hadoop framework. The window constraints specified in recurring queries can result in data overlaps between consecutive windows. Therefore, we propose a *Sample Repository* to cache the sampled data from previous executions on HDFS. The validity and quality of the cached sample data are maintained by the *Sample Meta-data* component. Such meta-data is utilized to make decisions for subsequent approximate executions. The deadline parameter combined with the run-time data arrival rate require Faro to provide dynamic strategies for approximate executions.

Beyond the map/reduce task structure of Hadoop, Faro adds several new components to Hadoop (illustrated by the white boxes in Figure 13.1), which are:

**1. Pre-Map and Post-Map Adaptive Samplers** are introduced in Faro to support approximate query execution. As described in Chapter 2.3, during the sampling phase, either or both sampling components could be used to select a subset of data blocks on HDFS for the approximate execution. All sample data are stored in a sample repository on HDFS for subsequent executions.

**2. Sample Repository** stores sample data produced to support previous approximate

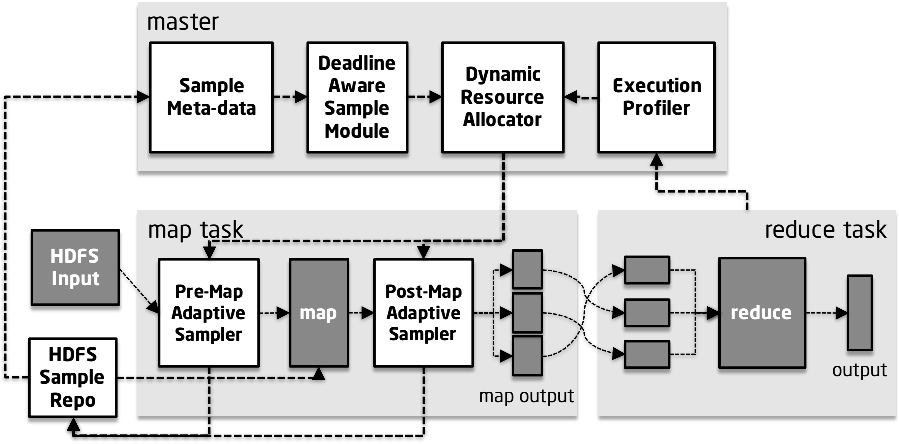


**Figure 13.1:** Faro Architecture

query executions. A subsequent execution at a later time can potentially reuse the sample data in the sample repository, further improving the accuracy of the query results within a shorter time.

**3. Sample Meta-Data** contains: (1) the location of the pre-map and post-map sample data, (2) statistics about the approximate executions that exploit the sample data, including result accuracy and sample rate, and (3) the validity of the sample data with respect to the given query's window constraints.

**4. Deadline-Aware Sample Module** housed on Hadoop master node determines the best sampling strategy for each approximate recurring query execution. Depending on the available resource and the input data distribution, a choice of pre-map or post-map sampling and their respective sampling rate will be made accordingly. Once this choice is made, we rely on pre-map or post-map sampling techniques to select uniform samples for the approximate query execution (Chapter 14).

**5. Dynamic Resource Allocator** fully exploits the sample repository that resides on HDFS for approximate query executions. It employs a variety of mechanisms to dynamically allocate resources to the approximate query executions depending on the query deadline. The resource assignment makes a trade-off between sampling new arrival data and refining existing result progressively, with a goal to maximize the result accuracy within the query's deadline (Chapter 15).

**6. Execution Profiler** collects the statistics after the completion of each query recurrence, i.e., execution times of previous query recurrences. The profiler then transmits the statistics to the Dynamic Resource Allocator such that the allocation decision can be made on-the-fly.

# 14

# Faro Deadline-Bound Sampling

Before introducing the proposed techniques for the approximate recurring query processing, we first formally define our problem.

**Definition 14.1** *Given a cluster configuration $cc$ that represents the available resources, and given a recurring query $\mathcal{Q}$ defined by three constraints: window size $\mathcal{W}$, slide $\mathcal{S}$, and deadline $\mathcal{D}$, the* ***approximate recurring query optimization problem*** *is to find an execution strategy $\mathcal{E}_{approx}$ that maximizes the average accuracy $\mathcal{A}ccu$ over the last $E$ consecutive executions. That is:*

$$Maximize : \frac{\sum_{i=1}^{E} \mathcal{A}ccu(\mathcal{Q}, cc, \mathcal{E}^{i}_{approx})}{E} \tag{14.1}$$

Maximizing the average accuracy over $E$ executions opens more opportunities for the system to optimize the workload from a more global perspective than locally focusing on each single execution, i.e., one on the very best executions. This average accuracy metric is commonly used in the service level agreement from Amazon AWS, Microsoft Azure, etc. The term "accuracy" here refers to the multi-stage sampling theory [68] that computes confidence intervals for each single execution of the recurring query $\mathcal{Q}$. The sampling techniques in [68] are suitable for aggregation queries. For other types of queries, application developers need to plug-in their customized technique into Faro.

Faro creates and maintains a set of samples to accurately and quickly answer queries. In this chapter, we describe the sample creation process in detail. First, we present a cost model to estimate the cost of the execution of an approximate job in the MapReduce framework using the proposed pre-map and post-map sampling techniques (Chapter 14.1). Second, we propose a deadline-aware partition strategy that allows an approximate execution to have maximized result accuracy with reduced sample size (Chapter 14.2). Lastly, we utilize statistical sampling theory to compute error bounds for the Faro approximate executions (Chapter 14.3).

## 14.1 Cost Model

Now we present a cost model to estimate the evaluation cost of a recurring query $\mathcal{Q}$ in the MapReduce framework using the proposed sampling techniques. Similar to other MapReduce work [38, 40], we model only the disk and network I/O costs as these are the dominant cost components. However, our cost model can be extended to include the CPU processing resources as well. Table 14.1 presents the system parameters used in our model, where the disk and network I/O costs are in units of seconds to process an HDFS block.

We assume each recurring query execution $Q_i$ is processed by $m$ map tasks and $r$ reduce tasks on the input file $F$. We use $|F|$ to denote the size of $F$ in terms of number of HDFS blocks. For a map output $M_i$, we use $p^m_{M_i}$ = $\lceil log_D \lceil \frac{|M_i|}{mB_m} \rceil \rceil$ to denote the number of sorting passes of its map tasks where $|M_i|/m$ denotes the average size of a map task. We use $p^r_{M_i}$ = $\lceil log_D \lceil \frac{|M_i|}{rB_m} \rceil \rceil - 1$ to denote the number of sorting passes of its reduce tasks where $|M_i|/r$ denotes the average size of a reduce task. We use $p_{M_i}$ to denote the sum of

| Parameter | Description |
|---|---|
| $C_{lr}$ | cost of reading a data block from local disk |
| $C_{lw}$ | cost of writing a data block to local disk |
| $C_l$ | sum of $C_{lr}$ and $C_{lw}$ |
| $C_{dr}$ | cost of reading a data block from HDFS |
| $C_{dw}$ | cost of writing a data block from HDFS |
| $C_d$ | sum of $C_{dr}$ and $C_{dw}$ |
| $D$ | merge order for external sorting |
| $B_m$ | buffer size for external sorting at mapper nodes |
| $B_r$ | buffer size for external sorting at reducer nodes |

**Table 14.1:** Cost Model Parameters

$p^m_{M_i}$ and $p^r_{M_i}$.

Given a recurring query $Q_i$, its total cost (denoted as $C_{ji}$) consists of its map and reduce costs (denoted as $C_{M_i}$ and $C_{R_i}$ respectively). The map costs are given by:

$$C_{M_i} = C_{dr}|F| + C_{lw}|M_i| + C_l p^m_{M_i}|M_i| \tag{14.2}$$

where $C_{dr}|F|$ denotes the cost to read the input file, $C_{lw}|M_i|$ denotes the cost to write the initial runs of the map output, while the cost to sort the initial runs is denoted by $C_l|M_i|p^m_{M_i}$. The reduce costs are given by:

$$C_{R_i} = C_{lr}|M_i| + C_l p^r_{M_i}|M_i| \tag{14.3}$$

where $C_{lr}|M_i|$ denotes the reading cost for the final merge pass, and $C_l|M_i|p^r_{M_i}$ denotes the sorting cost of the map output.

Therefore, the total costs can be expressed as follows:

$$C_{Q_i} = C_{dr}|F| + (C_l + C_l p_{M_i})|M_i| \tag{14.4}$$

Now we describe the costs of the pre-map and post-map sampling techniques based on the above cost model. The pre-map selects a subset of blocks from the input file $F$ for the mappers to process, and the post-map writes a subset of tuples as the map output to HDFS. Thus, the cost of these two techniques is given by:

$$C_{Q_i} = C_{dr}|F|\lambda_1 + (C_l + C_l p_{M_i})|M_i|\lambda_1\lambda_2 \tag{14.5}$$

where $\lambda_1$ denotes the pre-map sampling rate and $\lambda_2$ denotes the post-map sampling rate. Equation 14.5 is consistent with the observation described in Chapter 2.3. Namely, the pre-map sampling ($\lambda_1$) technique can reduce the cost faster compared to the post-map sampling ($\lambda_2$) technique.

Based on the cost model in Equation 14.5 and the deadline specified by the recurring query, Faro determines the sample size (i.e., the maximum number of tuples) denoted by $N$ that a query can process without exceeding its deadline constraint. The value of $N$ depends on the factors of distributed processing, concurrency, failures, data skew, and other issues. As a simplification, Faro estimates $N$ by assuming that the latency scales linearly with input size, as is commonly observed with a majority of I/O bounded queries in parallel distributed execution environments [89, 101, 102]. To avoid non-linearities that may arise at runtime, Faro's *Execution Profiler* collects latency statistics from query executions and sends them back to the *Deadline-Aware Sample Module* so to proactively correct the estimation.

## 14.2 Equi-depth Partitioning

It has been shown that a static sampling scheme cannot maintain a uniform sample in a bounded space in the context of time-based sliding window [103]. The reason is that the number of sampled tuples may exceed the bounded space with a fixed sample rate when the arrival rate of the data source suddenly increases. Our solution to this problem is based on the insight that we can reduce the sample size as well as minimize the error bound by partitioning the window into disjoint time intervals (buckets) and reducing sampling rates of individual buckets with huge amount of data tuples.

The main challenge in window partitioning is the placement of partition boundaries because they have a significant impact on the quality of the sample. A simple partition scheme is to divide a window into partitions of equal width (time intervals) and then to sample data tuples from each bucket such that the estimated error bound (described in Chapter 14.3) is identical among all the bucktes. As depicted in Figure 14.1(a), we refer to this strategy as *equi-width partitioning*. The data in black is sampled while the white colored are skipped. While this simple partition strategy would reduce the sample size and support incremental sample updates, it ignores the data distribution in each bucket and may not lead to a minimal sample size. Equi-width partitioning means that every bucket is sampled in the same way. Thus, the error bounds are the same across all buckets. However, buckets do not necessarily have to be under the same error bound, as long as the global error bound is minimized. Intuitively, some buckets are more important than others in affecting the global estimation and the error bound. For example, we can partition a skewed data set shown in Figure 14.1(b) into two buckets rather than equally partitioning it into four. More data are sampled from the second bucket with a small range. Hence, the global error bound is minimized with the same sample size of 4.

In general, dense buckets are underrepresented by an equi-width partitioning, while sparse buckets are overrepresented. This insight allows us to relax the error bounds for less critical buckets by sampling less data and in turn to tighten the error bounds for the more critical ones with higher sample rates.

Based on the above observations, we propose an alternative strategy, called *equi-depth partitioning*, where the window is partitioned into buckets of equal size (amount of data),

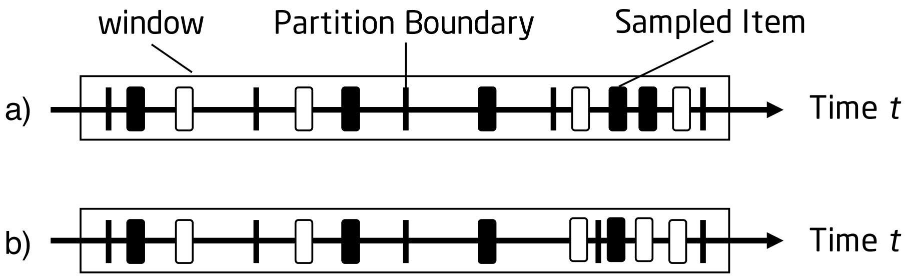


**Figure 14.1:** Equi-width (a) & Equi-depth (b) Partitioning

as shown in Figure 14.1(b). Equi-depth partitioning outperforms equi-width partitioning when the arrival rate of the data stream varies inside a window. Specifically, we (logically) partition the evolving input data file in a window $\mathcal{W}$ into disjoint unequal buckets. For each bucket we maintain a random sample using the uniform sampling. We need an algorithm to find an optimal equi-depth partitioning solution that minimizes the error bound, while satisfying the query's deadline.

Given a window size of $\mathcal{W}$, assume it contains data tuples $D = d_1, \ldots, d_N$, $d_i$, where $D$ is sorted based on the tuples' timestamp. A partition solution partitions $\mathcal{W}$ into $k$ disjoint buckets, $p_i = [a_i, b_i]$, $i = 1, \ldots k$, where $a_i$ and $b_i$ are the time boundaries of a bucket. Each bucket $p_i$ contains $N_i$ values. Let $\widehat{X}_i$ be the estimation using $n_i$ tuples in $p_i$, and let $\epsilon_i$ be the corresponding estimation error. A special case is when $\epsilon_i$ = 0, $n_i$ = $N_i$, i.e. all data in the $i$-th bucket is kept. The detail of error estimation will be discussed in Section 14.3. The global estimation $\widehat{X}$ and the error bound $\epsilon$ as well as those from individual buckets satisfy the following equations:

$$N = N_1 + \ldots + N_k \tag{14.6}$$

$$\widehat{X} = \frac{N_1 \cdot \widehat{X}_1 + \ldots + N_k \cdot \widehat{X}_k}{N} \tag{14.7}$$

$$\epsilon = \frac{N_1 \cdot \epsilon_1 + \ldots + N_k \cdot \epsilon_k}{N} \tag{14.8}$$

Given a window $\mathcal{W}$, our goal is to find a partitioning solution such that $\epsilon$ is minimized and $n_1 + \ldots + n_k < N$, where $N$ denotes the maximum number of tuples can be processed without exceeding the query's deadline constraint. Note that there is no specific number of buckets $k$ that the solution should aim for, instead the objective is to search the space for partitions that minimize the error bound $\epsilon$. This enumeration has an extremely high complexity, as all combinations of partitioning boundaries have to be considered as a potential optimal solution. Thus we present a heuristic algorithm in Algorithm 8 that exploits a local optimum search heuristic to reduce the complexity instead of exhaustively seeking the global optimum. We show that while the algorithm may not lead to the minimal error bound, the derived partitioning solution cannot be worse than conventional uniform sampling and most importantly often lead to significant improvements as proven in our experimental study (see Chapter 16).

Given a window size of $\mathcal{W}$, the algorithm iteratively decreases $\epsilon$ from 100% to 0%. At each iteration, the algorithm scans the window sequentially. For each new value (i.e., discrete time unit) encountered, the algorithm tries to merge it into the current partition (Line 7). Given the updated partition, Faro determines the minimal sample size by gradually increases two sample rates (i.e., $\lambda_1$ and $\lambda_2$) until the desired error bound is satisfied (Line 8). This sub-procedure has a polynomial time complexity. If the merging extends the partition's value range and increases the number of values to be sampled under the error bound $\epsilon$, a new partition is created (Lines 12-14). Otherwise, the value is merged into the current partition and the partition's range is updated (Line 10). The algorithm makes sure that the total sample size of all buckets does not exceed $N$ at the end of each iteration (Line 18). If not, the existing partitioning soution is replaced by the new one with lower $\epsilon$ value (Line 19). The algorithm's complexity in worst case is $O(\mathcal{W})$.

**Algorithm 8** Equi-depth Partitioning Algorithm

```
Input: A window size $\mathcal{W}$, Maximum sample size $N$
Output: A set of partitions $P_m = p_1, \dots p_k$
$\epsilon = 1$, $step = 0.05$, $P_m = \phi$
while $\epsilon \geq 0$ do
    $p_c = [d_1, d_1], p_c.size = 1$
    $P = p_c$
    $i \leftarrow 2$
    while $i \leq W$ do
        $p_t = [p_c.lower, d_i]$
        $p_t.size = computeSize(p_t)$
        if $p_c.size + 1 \leq p_t.size$ then
            $p_c = p_t$
        else
            $p_n = [d_i, d_i], p_n.size = 1$
            $P = p \bigcup \{p_n\}$
            $p_c = p_n$
        i++
    if $P.size <= N$ then
        $P_m = P$
    $\epsilon = \epsilon - step$
return $p$
```

**Theorem 14.1** *Given a bounded sample size $N$, the estimated error bound $\epsilon$ achieved by the equi-depth partitioning strategy given by Algorithm 8 is no more than uniform sampling.*

*Proof:* We prove Theorem 14.1 by showing the equi-depth partitioning strategy samples no more data than uniform sampling given a error bound $\epsilon$. If the algorithm returns one bucket, the equi-depth partitioning sampling is equivalent to uniform sampling. Next, we prove that stratified sampling with two buckets samples no more data than uniform sampling.

Let $[v_1, v_N]$ be the value range, and $[v_1, v_c]$ and $[v_{c+1}, v_N]$ be the two buckets' ranges. From the Hoeffding equation, we have:

$$\begin{aligned} & ln\frac{2}{1-\delta}\cdot\frac{(v_N - v_1)^2}{2\varepsilon^2} \\ \geq\ & ln\frac{2}{1-\delta}\left[\frac{(v_N - v_{c+1})^2}{2\varepsilon^2} + \frac{(v_{c+1} - v_1)^2}{2\varepsilon^2}\right] \end{aligned}$$

By case analysis, we can also prove the following inequality: $min\ \{x + y, N\} \geq min\ \{x, N_1\} + min\ \{y, N_2\}$ where $N = N_1 + N_2$. Then, we have:

$$\begin{aligned} & min\left\{ln\frac{2}{1-\delta}\cdot\frac{(v_N - v_1)^2}{2\varepsilon^2}, N\right\} \\ \geq\ & min\left\{ln\frac{2}{1-\delta}\cdot\frac{(v_N - v_{c+1})^2}{2\varepsilon^2}, N_1\right\} + \\ & min\left\{ln\frac{2}{1-\delta}\cdot\frac{(v_{c+1} - v_1)^2}{2\varepsilon^2}, N_2\right\} \end{aligned}$$

The left hand of the inequality is the sample size when applying uniform sampling. The right hand is the total sample size when using uniform sampling for two buckets. For equi-depth samplings with more than two buckets, we can prove by induction: let $[v_j, v_N]$ be the rightmost bucket and $[v_1, v_{j-1}]$ be the remaining range. We first show that using two or more buckets for $[v_1, v_{j-1}]$ is no worse than using one bucket. Then we show that combining $[v_1, v_{j1}]$ and $[v_j, v_N]$ is no worse than uniform sampling over $[v_1, v_N]$. ■

## 14.3 Error Estimation

To execute an approximate recurring query based on the sample, we show how statistical sampling theory can be used to compute error bounds (i.e., confidence intervals). Specifically, we apply the multi-stage sampling theory [68] to develop a unified approach for computing error bounds. As a proof of concept, we augment Faro with an algorithm for error-bound estimation for the most common operations in MapReduce, namely, aggregation operations. In general, Faro can execute arbitrary recurring queries with approximation, e.g., mining task, or even black-box analytical operations, as long as an appropriate algorithm for error-bound estimation is provided to the system.

We now introduce statistical estimation for recurring queries with aggregations. While providing statistical estimations for a wider range of operations provides an exciting research direction for future work, it is beyond the scope of this thesis. Note that we describe the approximation of sum aggregation here; approximations for the other operations are similar. Suppose we have a file on HDFS consisting of $BLK$ blocks, and each block contains $TPL_i$ tuples so that the total sample size $N = \sum_{i=1}^{BLK} TPL_i$. Suppose further that each tuple $j$ in block $i$ has a value $v_{ij}$. Then we want to compute the sum of these values across the file, i.e., $\sum_{i=1}^{BLK} \sum_{j=1}^{TPL_i} v_{ij}$.

To compute an approximate sum, Faro creates a sample by randomly choosing $blk$ blocks (i.e., pre-map sample rate $\lambda_1 = \frac{blk}{BLK}$), and then randomly choosing $tpl_i$ tuples from each chosen block $i$ (i.e., post-map sample rate $\lambda_2 = \frac{tpl_i}{TPL_i}$). Thus the sample size is $|\lambda_1 \cdot \lambda_2 \cdot N|$. Two-stage sampling then allows us to estimate the results for aggregation functions including sum, count, average, and ratio. For example, the estimated sum from the sample would be:

$$\widehat{\tau} = \lambda_1 \sum_{i=1}^{blk} (\lambda_2 \sum_{j=1}^{tpl_i} v_{ij}) \pm \epsilon \tag{14.9}$$

where the error bound $\epsilon$ is defined as:

$$\epsilon = t_{n-1,1-\alpha/2} \sqrt{\widehat{Var}(\widehat{\tau})} \tag{14.10}$$

$$\widehat{Var}(\widehat{\tau}) = (BLK - blk)\frac{s_u^2}{\lambda_1} + \frac{1}{\lambda_1} \sum_{i=1}^{blk} (TPL_i - tpl_i)\frac{s_i^2}{\lambda_2} \tag{14.11}$$

where $s_u^2$ is the inter-unit variance (computed using the sum and average of the values associated with tuples from each unit in the sample), $s_i^2$ is the intra-unit variance for unit $i$, and $t_{n-1,1-\alpha/2}$ is the value of the $t$-distribution with $n$ - 1 degrees of freedom at the desired confidence $1 - \alpha$. Thus, to compute the error bound with 95% confidence, we use the value $t_{n-1,0.975}$.

# 15

# Faro Adaptive Resource Allocation

In this section, we present Faro's resource allocation strategies which aim to maximize the average accuracy[1] of a recurring query. As discussed in Chapter 14, the accuracy refers to the distance to the true answer (e.g., the approximate results are within 5% of the true answer). Maximizing the average accuracy is challenging because of the fluctuating nature of evolving data sources. Variance of the input data sources can at times result in temporary load spikes, consequently resulting in failures to meet the query deadline.

Since a recurring query imposes deadlines to deliver the results, its approximate execution will require judicious resource allocation strategy to achieve high result accuracy. As described in Section 14.1, Faro exploits its Execution Profiler to determine the maximum number of tuples $N$ that can be processed given a query's deadline and the available resource. Hence, the resource is referred to as the bounded sample size in the rest of this section. The deadline-bound sampling strategies for approximate executions described in Chapter 14.2 is at the *file granularity*. However, the scope of data to process in a window $\mathcal{W}$ can contain multiple files, consequently multiple approximate executions. This requires Faro to determine where the resources should be used in approximate executions

[1] The average accuracy is computed over the last $E$ consecutive executions.

of a recurring query at the *window granularity*. Namely, the resource allocation strategy in Faro decides the bounded sample size for each approximate execution in the current window so to maximize the result accuracy of this window.

Given that multiple files belong to a window $\mathcal{W}$, including both new arrival and existing files, a unique opportunity of re-sampling to improve the result accuracy can be exploited on these existing files that have been previously executed. This extra option means that a prioritization must carefully weigh the gains from sampling on existing data versus new arrival data while still meeting the deadline constraint. However, decisions in favor of one purpose may affect another purpose instead, which might require to recompute the gains of the rest of the executions. The resource allocation problem is NP-complete [104] and requires efficient heuristics to make appropriate decisions adaptively at run-time as illustrated in the following example.

**Example 15.1** *Given a recurring query $Q$ with $\mathcal{W} = 6$ and $\mathcal{S} = 2$. As depicted in Figure 15.1, the file $F_3$ is included in three windows, and thus it will participate in three consecutive executions. However, due to $Q$'s deadline, the available resources, and the size of new arrival file $F_4$, the second execution $E_2$ over $w_2$ may not be able to sample all three files. In fact, $E_2$ has to decide among the following options: (1) sampling data from the new file $F_4$, (2) sampling data from one or more of the existing files $F_2$ or $F_3$, where data blocks that have not been sampled in previous executions can now be sampled, and (3) sampling data from either a combination of $F_2$ and $F_4$, or $F_3$ and $F_4$. If $E_2$ chooses not to sample $F_3$ for its own sake, then the potential benefit of sampling $F_3$ decreases as it can only contributed to the third execution $E_3$. Consequently, it will be less likely that we select $F_3$ for $E_3$. On the contrary, if $E_2$ chooses $F_3$ rather than $F_4$, then the decision of $E_3$ could be different, selecting $F_3$ for re-sampling to further increase the result accuracy.*

In brief, we may need to make the resource allocation decision dynamically based on not only the current situation but also the decisions made previously. This problem

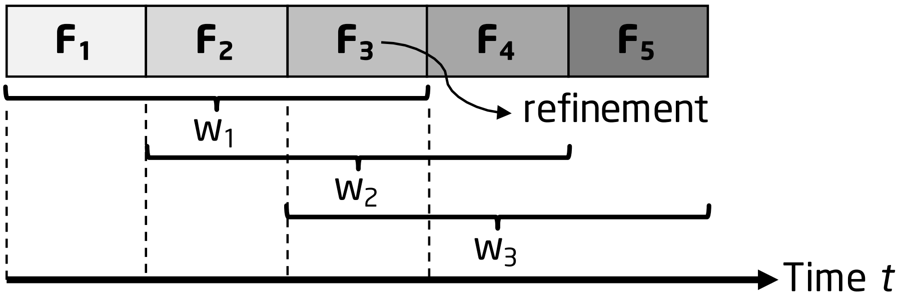


**Figure 15.1:** Sampling on New and Existing Data

is, unfortunately, NP-complete in general [105] and devising good heuristics is critical to maximizing the average accuracy of approximate query executions. In the following, we first define the concept of lifespan of each file (informally introduced in Chapter 5) and then introduce our Greedy Resource Allocation (GRA) mechanism and Accuracy-Aware Resource Allocation (AARA) mechanism in Chapters 15.1 and 15.2, respectively.

**Definition 15.1 (Lifespan of File)** *A file $F_i$ in the current window may serve also executions over future windows. The number of windows in which a file can survive is termed the lifespan of file.*

In the recurring query scenario, the lifespan of a file $F_i$ can be determined as follows.

**Lemma 15.1** *Given the slide size $\mathcal{S}$ of a query $Q$ and the starting time of the current window $W_c.T_{start}$, the lifespan of $F_i.life$ of a data block $F_i$ in $W_c$ with timestamp $F_i.ts$ is calculated by $F_i.life = \lceil \frac{F_i.ts - W_c.T_{start}}{\mathcal{S}} \rceil$, indicating that $F_i$ will participate in window $W_c$ to $W_{c+F_i.life-1}$.*

Therefore, the longer a file's lifespan is, the more future approximate executions the file can participate in. Naturally, the file with the largest lifespan could be a good candidate to get substantial resources for its approximate execution. Henceforth, quickly updating the lifespan of each file is critical for deriving resource allocation decision and in turn maximizing result accuracy.

## 15.1 Greedy Resource Allocation

Greedy Resource Allocation (GRA) is an algorithm that greedily picks the files with the longest lifespan to execute next. Specifically, only the new arrival files would be considered for the resource allocation since they have the longest lifespan. Each file will be assigned resources in proportion to the file's respective size. Given the available resources associated to each file, the sampling technique described in Chapter 14.2 can be applied. Such greedy allocation strategy may often waste resources when the size of new arrival file is relatively small compared to existing ones. In an extreme case, if the available resource is sufficient for a full execution on the new file, then the error estimation of the result for such execution would be $0\%$. However, it would be more beneficial to divide the resource among both new arrival and existing files, such that the result accuracy for the existing files can be much increased with a slight decrease in the result accuray for the new file. Hence the global result accuracy can be improved.

If one extends this idea to the case where re-sampling on existing files is considered, then a natural approach is to first group all files according to their lifespans. Each group will be assigned resources in proportion to their respective lifespan. Within each group, the assigned resources will be further allocated to each file with respect to their respective file size. We term this the Greedy Resource Allocation (GRA) policy (see Algorithm 9). The following example illustrates GRA policy in detail.

**Example 15.2** *Given a query $Q$ with $\mathcal{W} = 5$ and $\mathcal{S} = 2$ as depicted in Figure 15.2, to decide the resource allocation for the second execution over the window $w_2$, we first group the input files contained in $w_2$ into three groups (i.e., $\{F_3, F_4\}$, $\{F_5, F_6\}$, and $\{F_7\}$) according to their lifespans. $F_3$ and $F_4$ will only be valid for $w_2$ and thus $F_3.life = F_4.life = 1$. According to Lemma 15.1, $F_6.life = 2$, even though it is just newly arrived. Assume that the total number of sample data that can be processed by the available re-*

**Algorithm 9** Greedy Resource Allocation

```
Input: A set of files $F$ in $W_c$, Available resource $R$
Output: A resource allocation plan $p$
$LS = \phi$
for each $F_i$ in $F$ do
    $F_i.lifespan$ = computeLS($F_i$)
    if !$LS$.containsKey($F_i.lifespan$) then
        $LS$.put($F_i.lifespan$, new List($F_i$))
    else
        $LS$.get($F_i.lifespan$).add($F_i$)
$totalLS$ = sum($LS.keySet$)
for each element $ls$ in $LS$ do
    $weight = ls.key/totalLS$
    $totalSize$ = sum($ls.values$)
    for each $F_i$ in $ls.values$ do
        $F_i.resource = weight \times F_i/totalSize$
        $p$.add($F_i$)
return $p$
```

*sources $R$ is 12 million, three groups will get 2 million, 4 million, and 6 million as their bounded sample size respectively. We further assume that all files have identical size, then the resource assigned to these files (from $F_3$ to $F_7$) would be 1, 1, 2, 2, and 6 (million).*

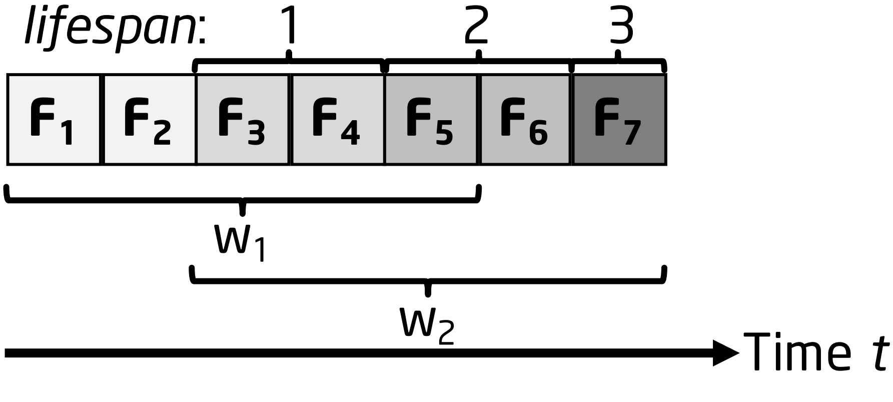


**Figure 15.2:** Greedy Resource Allocation

The greedy mechanism works well when an evolving data source has relatively stable arrival rates (i.e., relatively identical file size). In this case, the accumulative resources allocated to each file are the same. Hence, the average result accuracy for the executions on each file is the same as well. Since the result accuracy for each execution is maximized using the deadline-aware sampling strategy proposed in Chapter 14.2, the overall accuracy is thus maximized. However, when data sources exhibit load variances, the GRA mechanism may fail to maximize the average result accuracy by wasting resources on files showing load spikes, as explained in the following example.

**Example 15.3** *Continuing with Example 15.2, assume that $w_1$ has a load spike in which the size of the file $F_5$ is three times larger than the other files. As before, the policy generated by GRA would allocate a bounded sample size of 4 million to the group $\{F_5, F_6\}$. Given that $|F_5| = 3|F_6|$, the resource assigned to the file $F_5$ would be 3 million (25% of the total resource). By re-sampling $F_5$ to increase the sample size, certainly the more accurately the sample will reflect the population in $F_5$ it was drawn from. However, the accuracy gain from re-sampling is less significant compared to re-sampling on the other files with identical resources due to the file size of $F_5$. Hence the resource allocation mechanism should take the accuracy gain into consideration as well. In this case, a better resource allocation plan would assign 25% of the total resource to the other files in proportion to their lifespans.*

## 15.2 Accuracy-Aware Resource Allocation

The opportunity cost of re-sampling on existing files is an important factor to consider, and leads to a refined policy called the Accuracy-Aware Resource Allocation (AARA) mechanism.

**Definition 15.2 (Accuracy Gain)** *If a file $F_i$ is valid for both windows $w_j$ and $w_{j+1}$, the result accuracy of two consecutive query executions over $F_i$ are $\epsilon_j$ and $\epsilon_{j+1}$, respectively.*

*Then the accuracy difference is defined as $\epsilon_\Delta = \epsilon_j - \epsilon_{j+1}$. The resource assigned to $F_i$ in execution $E_{j+1}$ is $R_{i,j+1}$. The accuracy gain on $F_i$ is defined as a ratio between the accuracy difference and the assigned resource, namely $\frac{\epsilon_\Delta}{R_{i,j+1}}$.*

Therefore, the larger a file's accuracy gain is, the more cost effective the file is. If the accuracy gain between two consecutive executions over a file is low, it would be more beneficial to dedicate the resource to other files with higher potential of increasing the global result accuracy. Naturally the file with largest accuracy gain could be a good candidate for the next approximate execution. Henceforth the accuracy gain of each file has to be updated after each execution in order to derive resource allocation decision and in turn maximizing result accuracy.

We now design a resource allocation mechanism that favors files based on both their lifespan and accuracy gain. While the lifespan of a file is determined by the window constraints from the query, while the accuracy gain is determined by a ratio between error estimation and cost. Our Accuracy-Aware Resource Allocation (AARA) assigns a weight ($weight(F_i)$) to each file $F_i$ that incorporates the above two concepts:

$$weight(F_i) = F_i.life \times \frac{\epsilon_\Delta}{R_{i,j+1}} \tag{15.1}$$

The special case of Equation 15.1 is when a file is brand new. Since the file has not been processed yet, the error bound from previous execution is not available. Consequently, the accuracy gain cannot be computed. Therefore, we set the default weight of a new arrival file to be its lifespan. This makes sure that the file does not suffer from starvation. After assigning weights to the files, the files are sorted by their weights in descending order. The available resources are allocated to each file in proportion to the file's respective weight, rather than being allocated to different group of files solely based on their respective life spans.

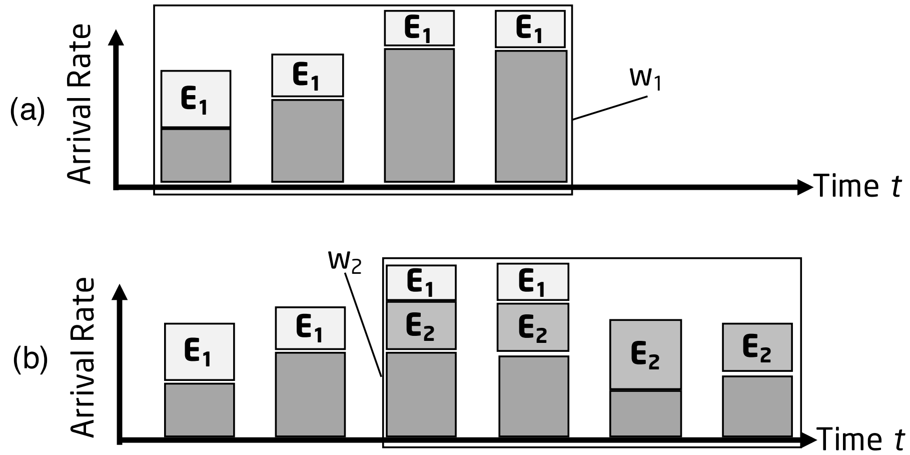


**Figure 15.3:** Accuracy-aware Resource Allocation

Our AARA mechanism can dynamically allocate the available resources to the files with maximal potential accuracy gain. However it still suffer from two issues. First, the weight of a file continuously changes, as we can see from Equation 15.1. As a consequence, the file with the highest weight varies over time. On the one hand, this dynamic ordering is a key to the good result accuracy of Faro. It imposes a time complexity of $\Omega(|F|)$ for updating the $|F|$ files waiting to be processed in the current window. Second, the files with relatively lower scores may get limited to no resources. In this case, the file cannot be sampled effectively due to the MapReduce job starting and ending overhead. To handle these two issues, a natural variant is to select top $k$ files as an input for the AARA mechanism. To determine the initial value of $k$, we again use the accuracy gain as a threshold. Namely, only the ones with their accuracy gains above the average accuracy gain of all files are considered for the accuracy-aware resource allocation. Furthermore, $k$ is periodically adjusted based on the latest average accuracy gain of all files from previous executions at runtime. Monitoring top-$k$ input files in sliding windows is a well-studied problem [106], which only takes logarithmic time in the size of $O(log(k))$ in the average case. Algorithm 10 describes the details of revised AARA mechanism.

**Algorithm 10** Accuracy-Aware Resource Allocation

```
Input: A set of files $F$ in $W_c$, Available resource $R$
Output: A resource allocation plan $p$
for each $F_i$ in $F$ do
    $F_i.weight$ = computeWeight($F_i$)
$F$ = topK($F$)
$sumWeight$ = 0
for each $F_i$ in $F$ do
    $sumWeight$ += $F_i.weight$
for each $F_i$ in $F$ do
    $F_i.resource$ = $R \times F_i.weight/sumWeight$
    $p$.add($F_i$)
return $p$
```

# 16

# Experimental Evaluation

In this chapter, we describe the experimental study we conducted to evaluate Faro. First, we compare Faro to recurring query execution on full-sized data sets to demonstrate how even a small trade-off in the accuracy of final answers can result in orders of magnitude improvements in query execution times. We also demonstrate Faro's ability to scale gracefully with increasing cluster size and input data size. Second, we evaluate the accuracy of our deadline-aware sampling approach against random sampling approach given identical resources. Third, we evaluate the effectiveness of our dynamic resource allocation mechanisms in meeting the query's deadline requirements while maximizing the result accuracy. Finally, we demonstrate Faro's efficiency and robustness with a commonly used data mining algorithm.

## 16.1 Experimental Setup & Methodology

*Experiment Infrastructure.* All experiments are conducted on a shared-nothing cluster with one master node and 40 slave nodes. Each node consists of 16 core AMD 3.0GHz processors, 32GB RAM, 250GB disk, and nodes are interconnected with 1Gbps Ethernet.

Each server runs CentOS Linux (kernel version 2.6.32), Java 1.6, Hadoop 0.20.1. Each node is configured to run up to 8 map and 8 reduce tasks concurrently. The sort buffer size was set to 512MB, and speculative execution was disabled to boost performance. The replication factor is set to 3 unless stated otherwise.

*Datasets and Queries.* We use two real-life datasets for our experiments. The World Cup Click dataset [77] (256GB) contains records of more than 1.35 billion web requests made to 1998 World Cup Website. Another dataset is the latest high volume Wikipedia database [78] (400GB) being modified and updated continuously.

We first focus on aggregation queries which are fundamental operations not only in relational databases, but also in the emerging data analytics tasks described in Chapter 1.2. These aggregation queries were generated from the following query template:

```
SELECT S1.T, AggFunc(A1) FROM S1
WHERE Predicates GROUP BY S1.T
[WINDOW, SLIDE, WITHIN]
```

where $T$ is a randomly selected list of dimensional attributes. The aggregation operators are SUM, COUNT and AVG. Each query is also assigned a query deadline that ranges from [0.05 - 0.2] of the query's window size $\mathcal{W}$. Faro can also be used to provide approximation results for advanced mining algorithms such as parallel k-means clustering on MapReduce [54]. Faro can seamlessly integrate their techniques to speed up k-means without changing the underlying algorithm. We execute each experiment three times. In the charts we report their average results.

*Metrics & Measurements.* Given a recurring query, we measure the average accuracy of each query, and the average execution time for the recurrences of each query. The execution time of recurring queries is a common metric in data management systems, while the average accuracy is defined in Chapter 2.4. We do not include the data pre-processing time since it is performed on-the-fly during the loading time. It is hence negligible compared to the disk-based query processing in Hadoop.

*Methodology.* We implemented the proposed Faro techniques for approximate execution of recurring workloads on top of the extended open-source Apache Hadoop. We evaluate Faro's effectiveness under different parameter settings, including varying the deadline of the query, input data volume, cluster size, and the overlap (i.e., the ratio between slide size and window size). We compare three systems denoted Faro, Redoop, and Hadoop, respectively. Redoop [64] is the state-or-the-art approach for evaluating a recurring query in MapReduce. It minimizes recurring query execution time by caching reduce inputs, without taking into consideration approximation. Regarding the efficiency of the Faro deadline-aware sampling strategy, we compare our equi-depth partitioning strategy to the equi-width partitioning strategy. We also compare Faro's greedy and accuracy-aware resource allocation mechanisms against a baseline approach that always dedicates all resources to the new arrival files.

## 16.2 Faro vs. Full Execution

We first compare the performance of Faro versus Redoop and stock Hadoop that executes queries on complete data in each window. In this experiment, we ran on both data sets. To demonstrate the significance of sampling even for the simplest analytical queries, we ran a simple aggregation query that computed sum count. We compared the processing time of the full (accurate) execution of this query on Hadoop against its (approximate) execution on Faro with deadline constraints. Figure 16.1 shows the results of processing time. As depicted in Figure 16.1, Faro significantly reduces the processing time by a factor of 8 - 14 folds compared to Hadoop with both data sets. This is because Faro is able to read far less data to compute a fairly accurate answer. It also achieves 4 - 6 times performance gain compared to Redoop system. Regarding the statistical error, Faro returns the results with 3% and 5% error bound at 95% confidence with both data sets, respectively.

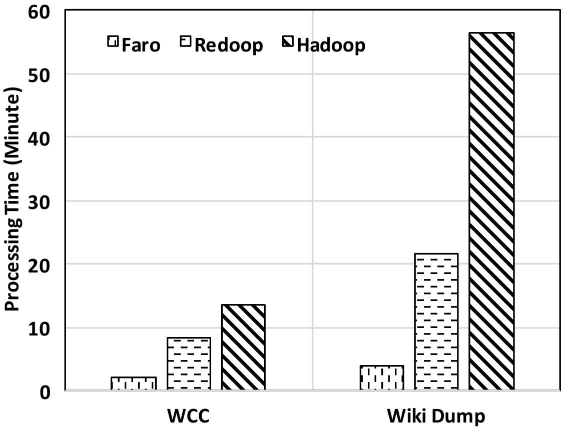
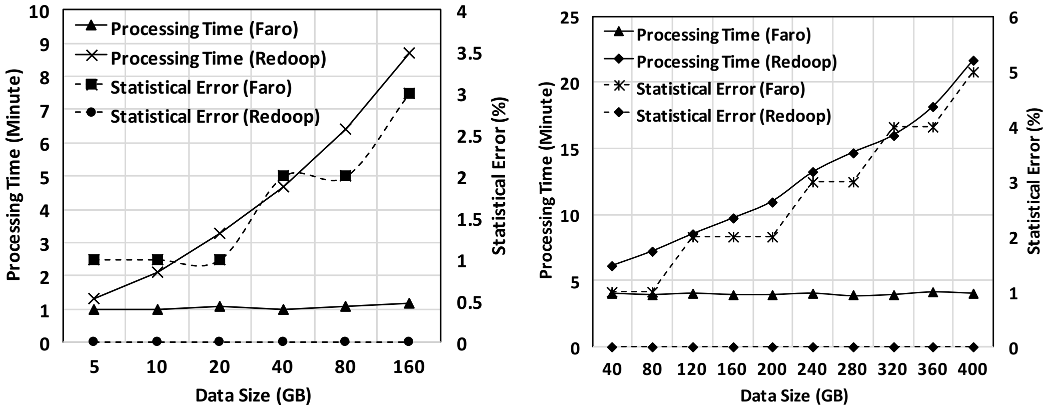


**Figure 16.1:** Faro vs. Full Execution

Now we closely study the scalability of our Faro solution by varying two parameters, i.e., data size and and cluster size. We show the experimental results of Faro over Redoop indicated in Figures 16.2 and 16.3. We omit the comparison against Hadoop as Redoop system consistently outperforms it in most cases [64].

*Effect of data size.* Figure 16.2 examine the execution times of Faro using two data sets. We vary the data sizes per job ranging from 5GB to 160 GB for WCC data set and 40GB to 400GB for Wiki Dump data set. The setting for the other parameters is: $overlap = 0.9$, $\mathcal{N} = 10$, and $\mathcal{D} = 1$ and 4 minutes. Faro substantially reduced the processing time compared to Redoop again. For example, the average processing time of Redoop for each execution is 5 times longer than Faro for aggregation query when the data size is 400GB. In particular, the running time of Redoop increases substantially when the size of data size increases. On the contrary, Faro is able to meet the deadline constraints without sacrificing much result accuracy, with 5% error bound with 95% confidence in worst case (see Figure 16.2(b)). This shows that the robustness of our Faro approach against varying data size. The reason behind this is that by exploiting the deadline-aware sampling technique and the accuracy-aware resource allocation mechanism, the Faro approach can effectively sample data in a progressive manner while meeting the specified deadline of query executions.

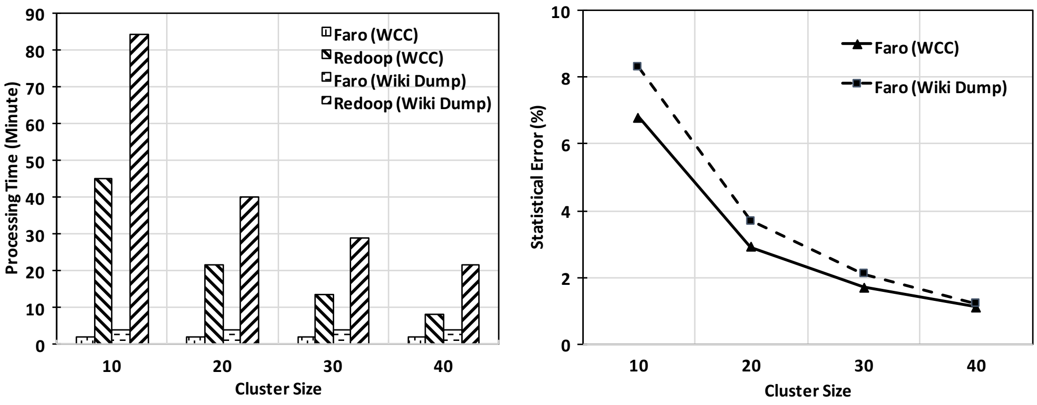


(a) WCC

(b) Wiki Dump

**Figure 16.2:** Varying Size of Data Set

*Effect of cluster size.* Figure 16.3 compares the scalability of both systems by varying the number of nodes in the cluster used. Each query execute on $10n$ GB of data (where $n$ is the cluster size). So for a 10 node cluster, each execution consumes 100 GB of data and for a 40 node cluster each query operates on around 400 GB of data. As expected, our Faro solution consistently outperforms Redoop in the execution time as shown in Figure 16.3(a). Regarding the statistical error, the Faro system shows its robustness as plotted in Figure 16.3(b). The corresponding results for Hadoop system are omitted since it operates on the full data sets. Thus, the error is always 0%. In the worst case scenario (the number of nodes is 10), the statistical error of Faro system is still below 8% with 95% confidence. This verifies the design goal of our Faro system, trading off between query accuracy and execution time.



(a) Processing Time

(b) Statistical Error

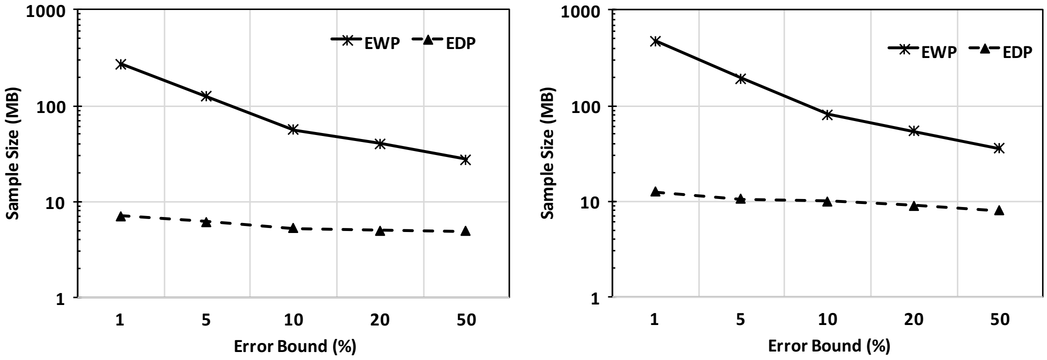
**Figure 16.3:** Varying Number of Nodes

## 16.3 Deadline-Aware Sampling

In this section, we evaluate the effectiveness of Faro's deadline-aware sampling technique. We compare our equi-depth partitioning strategy (denoted by $EDP$) to the equi-width partitioning strategy (denoted by $EWP$) with different error bounds, using both data sets. Results here are averaged among all executions. The confidence is set to $95\%$. The sizes of the samples produced by the two approaches are shown in Figure 16.4. As we can see, when the error bound increases, the sample size of the $EDP$ decreases dramatically. The $EWP$ approach is able to produce a small number of samples with different error bound constraints. It is much more robust against the change of error bounds.

## 16.4 Adaptive Resource Allocation

We next evaluate Faro's runtime resource allocation mechanisms by varying three parameters, the overlap (i.e., the ratio between slide size and window size) and the query's deadline ($\mathcal{D}$).We compare Faro's greedy and accuracy-aware resource allocation mechanisms (denoted by $GRA$ and $AARA$ respectively) against the baseline approach $INC$



(a) WCC

(b) Wiki Dump

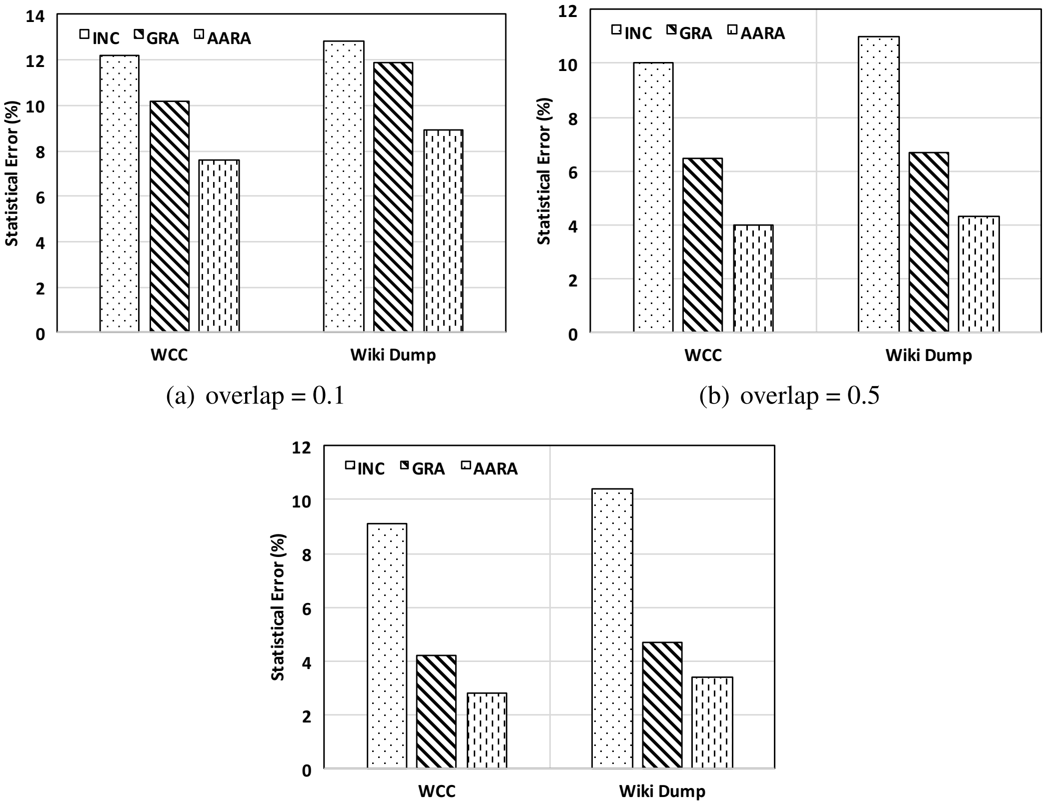
**Figure 16.4:** Deadline-Aware Sampling Technique

that always dedicates all resources to the new arrival files. We omit the comparison to the approach that samples the data over the current window from scratch as it is been proven to be inefficient in Chapter 14.

*Effect of overlap.* The factor $overlap = (\mathcal{W} - \mathcal{S})/\mathcal{W}$ represents a ratio between the slide and window size. This ratio represents the portion of the newly arriving data tuples in the window after the window slides. Thus, the higher the ratio is, the greater the amount of data shared between consecutive windows. We vary the $overlap$ from 0.1 to 0.9 which shows how these mechanisms behave under different scenarios. Figure 16.5 shows the results where $AARA$ clearly outperforms the other two mechanisms. We now describe these results in detail.

$AARA$ on average improves the accuracy of deadline-bound jobs by 216% and 324% compared to $GRA$ and $INC$ respectively. Gains in both data sets are similar. The gains compared to $INC$ as baseline are consistently higher than $GRA$. Also, the gains with large $overlap$ are pronounced compared to small and medium $overlap$ because the relatively longer lifespan of each file provides plenty of potential for $AARA$ to improve the result accuracy over several consecutive executions.

*Effect of deadline.* For experiments on deadline, we first calibrate and obtain the



(a) overlap = 0.1

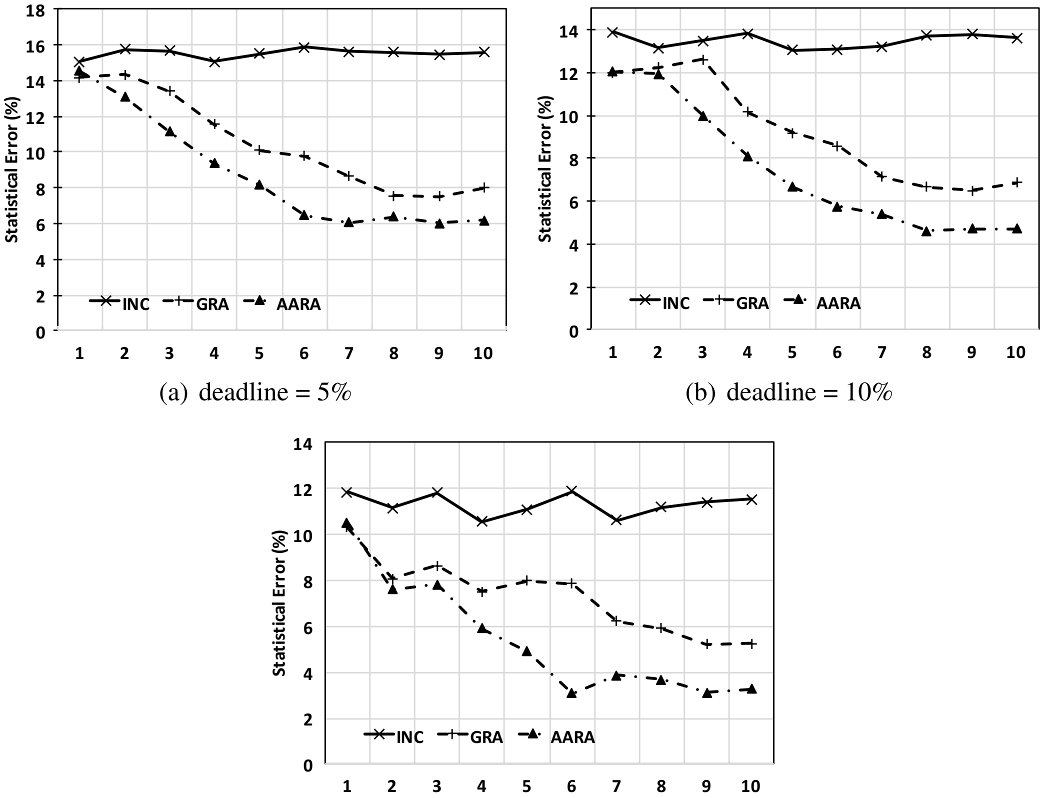
(b) overlap = 0.5

(c) overlap = 0.9

**Figure 16.5:** Varying Overlap

processing time of a full execution by submitting the query to stock Hadoop. We then set deadlines to be an factor (between 5% to 20%) of this full duration. Figure 16.6 shows the statistical error over 10 consecutive windows over Wiki Dump data set. Figure 16.6 shows the results where $AARA$ consistently outperforms the other two mechanisms for 10 windows. We now describe these results in detail.

As plotted in Figure 16.6, for the initial window, all three mechanisms need to sample the whole window full of tuples and thus $AARA$ and $GRA$ achieve slightly more accurate results (i.e., statistical error). For the subsequent sliding steps (windows 2-10), $AARA$ and $GRA$ benefits from the sample refinement over existing files, resulting in a substantial improvement of estimated error compared to $INC$. Specifically, $AARA$ gains 6 fold



(a) deadline = 5%

(b) deadline = 10%

(c) deadline = 20%

**Figure 16.6:** Varying Deadline

improvements over $INC$ when the deadline is 20% of the duration of a full execution (shown in Figure 16.6(c)).

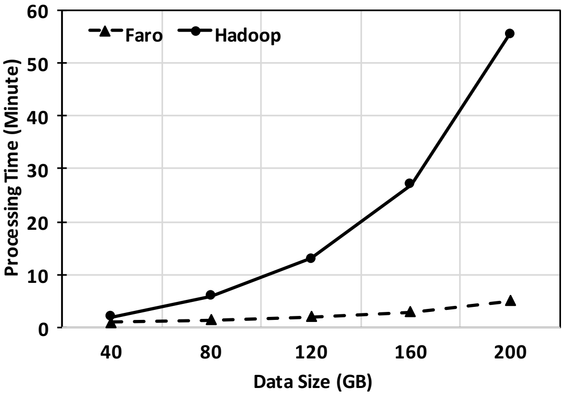


**Figure 16.7:** Faro vs. Hadoop with $k$-means

## 16.5 Data Mining Task

In this section, we conduct a performance study when using Faro to approximate the execution of a commonly used data mining algorithm, $k$-means. There are various techniques used to speed up $k$-means, including parallel processing based on MapReduce [54]. Faro plugged in their techniques as a black box to speed up $k$-means. In this experiment, we use a synthetic data set so that we can validate that Faro finds the actual centroids. Figure 16.7 shows the results of running $k$-means with Faro and stock Hadoop. The query deadline is configured from 1 to 5 minutes corresponding to different data size from 40 to 200GB. As expected, our approach significantly outperforms Hadoop by approximately 10x with only 7% of the optimal when the data size is 200GB. The reasons are twofold: (1) $k$-means is executed over a small sample of the original data and (2) $k$-means converges more quickly for smaller data-sets.

# 17

# Related Work

*Approximate Query Processing (AQP).* Sampling has been applied for providing approximate answers to relational database systems [65, 67, 107, 108] and data stream systems [109, 110, 111]. Most works use statistical inequalities and the central limit theorem to model the confidence interval or variance of the approximate answers. For example, online aggregation by Hellerstein et al. [67] focuses on grouped aggregation with statistically robust confidence intervals based on random sampling. This was extended to handle join queries using the ripple join family of operators in [112]. BlinkDB [65] constructs a large number of multi-dimensional samples from a static data set using stratified sampling. During query processing, it then chooses samples based on the accuracy and response time requirements of an aggregation query. Other works focused on specific types of queries. For example, Acharya et al. [113] study approximation for join queries using samples from the base relations. Joshi and Jermaine [108] propose an EM algorithm to execute aggregate queries with subset testing.

Instead of the system taking responsibility for result accuracy which may not be possible in general, our Faro follows a different approach. Faro provides an interface for users to plug-in their own one-pass sampling technique. Faro supports a variety of user-defined sampling techniques; we view prior work described above as part of a layer between the user and our generic approximate processing framework, that helps with the resource allocation of given recurring queries in a deadline-aware manner.

*AQP in MapReduce.* Recent extensions to Hadoop, e.g., MapReduce Online (MRO) [80], EARL [54] and ApproxHadoop [55], focus on approximate results for analytics on massive data sets. MapReduce Online (MRO) [80] supports progressive output by adding pipelining to MapReduce. Early result snapshots are produced by reducers, each annotated with a rough progress estimate based on averaging progress scores from different map tasks. Early termination of map tasks is addressed in [114] by producing a fixed-size sample of a massive data set using MapReduce. However, unlike Faro, neither system attempts to provide any error estimation. Neither guarantees uniform samples. EARL [54] utilizes uniform sampling and works iteratively to compute larger samples on a static data set, until a given accuracy level is reached. ApproxHadoop [55] leverages statistical theories to compute error bounds for MapReduce jobs when approximating with input data sampling and/or task dropping. However, none of these efforts identified and leveraged optimization opportunities unique to recurring queries, e.g., understanding window semantics, incremental sample updates, and progressive result refinement across the consecutive execution of a recurring query. This leads to inefficient sampling, wasting precious computational resources, and possibly poor result accuracy. The proposed Faro system seamlessly integrates *approximate query processing* with *recurring query execution*. For that, it provides progressive result refinement by leveraging the recurring nature of the queries.

*Recurring Query Processing.* Recurring query processing systems [63, 64, 92, 115] have been proposed to support large-scale data analytics applications over evolving data streams. SCOPE [63, 92] handles recurring queries by instrumenting queries to piggyback statistics collection with their normal execution. Collecting such statistics makes it possible to create a statistical profile that can be fed to the optimizer on a future invocation of the same job. Redoop [64] employs window-aware optimization techniques for recurring query executions, including adaptive data partitioning, window-aware task scheduling, and inter-window caching. Helix [115] supports the optimization of multiple recurring queries with service level agreements (SLAs). It explores the sharing opportunities among the queries to maximize the satisfaction of SLA requirements. However, none of these works considers approximate query processing as an option. If data arrival rate is high and SLA requirements (i.e., deadlines) are tight, these full execution approaches would fail. In contrast, our Faro solution aims to provide the best quality results within the given time constraints.

*Service Level Agreement:* Meeting SLA constraints has been addressed in the context of stream processing systems [57, 84, 99, 100]. Prior work [99] leveraged specific workload characteristics to meet SLAs without losing efficiency or utilization. To provide real-time responses, [99] enables the user to specify a contract in terms of latency, data freshness, CPU and memory usage. Its main method is to shed data from incoming streams to handle load and meet the desired QoS. Our aim is not load shedding but instead to provide appropriate sampling strategies with error bounds. The second solution in [100] employs scheduling techniques that rely on small, uniform task durations to trade short-term SLA violations for efficiency. However, these workload characteristics are not universal. In contrast, our Faro solution is independent of workload characteristics. Moreover, the resource allocation mechanisms in Faro not only meet SLA requirements (i.e., deadlines) but also maximize the accuracy of approximated results with best efforts.

Several techniques [50, 51, 52] have been proposed for achieving SLAs of MapReduce jobs. These methods either dynamically adjust resource allocation or they exploit profiling to help jobs provision resources statically at startup. However, none of these efforts consider approximate query processing as an approach to tackling the problem.

Moreover, they do not address any unique challenges derived from targeting recurring queries as done in our work.

# Part IV

# Conclusion and Future Work

# 18

# Conclusions of This Dissertation

The goal of this dissertation is to break fundamentally new ground in big data processing by supporting a wide spectrum of data-intensive recurring queriesoverlooked by current systems. This dissertation proposes a new comprehensive data and query model for big data analytical workloads, and offers end-to-end optimizations ranging from query sharing, approximate evaluation, and SLA guarantees, to adaptive processing. This will impact applications in domains from business, scientific to Internet-scale disciplines that require recurring big data processing. The three highlights of this dissertation can be summarized as follows.

First, we focus on the problem of the efficient execution of the recurring query. We achieve this by presenting the design, implementation, and evaluation of the Redoop technology, a novel distributed system that optimizes the recurring query processing as MapReduce jobs on big data. Redoop offers 3 key innovations: (1) adaptive incremental data processing to reduce resource utilization and to reduce query processing time; (2) window-aware caching to avoid repeated work and disk access; and (3) window-aware cache-oriented scheduling to improve the cached data utilization and to improve the query processing performance. Our experiments show that the Redoop system achieves an up to 9 times performance gain compared to the standard Hadoop.

Second, we target the optimization for shared execution of recurring workloads on MapReduce. Our Helix system offers 3 key innovations. (1) The recurring query model established for Helix integrates the multiple recurring query optimization problem in MapReduce with the SLA satisfaction. (2) The sliced window alignment technique opens new sharing opportunities by partitioning the data sources into sharing-appropriate granularities. (3) Two novel strategies, sharing and ordering methods, effectively prune suboptimal solutions from the search space. They guide the Helix optimizer to explore the more promising part of the search space first, thus succeeding to efficiently produce the optimal global shared execution plan. Our experimental results on a rich variety of workloads show that our proposed techniques outperform the state-of-the-art approaches consistently by up to an order of magnitude.

Lastly, we address the problem of approximate recurring query processing on MapReduce. Our Faro system offers two key innovations. (1) a deadline-aware sampling strategy that builds samples from original data with reduced sample size compared to uniform sampling, and (2) adaptive resource allocation strategies that maximally improve the approximate results while assuring to still meet the queries' response time requirements. Our experimental results on a rich variety of workloads show that our proposed techniques achieve 14x faster performance than the state-of-the-art approaches with an estimated error of 2-5%. A direction for the future is to investigate other sampling methods that although are not as general as uniform sampling can still provide better performance in specific analytics applications.

# 19

# Future Work

## 19.1 Integration of Proposed Techniques

The obvious directions for future research based on the proposed techniques in this dissertation are to consider the inter-relationships among them. Namely, the potential benefits to the recurring query processing can be achieved by exploiting the proposed techniques in an integrated manner.

We first analyze the interrelationship between Redoop and Helix, and discuss the integration of these two techniques. Recall that Redoop is optimized for single recurring query execution and Helix is designed for sharing similar recurring workloads. Essentially, both techniques aim to avoid redundant I/Os and computations at different stages in MapReduce. To reduce the unnecessary I/O costs resulting from the overlapping windows, Redoop's task nodes cache the input data partitions on their local file systems for subsequent reuse. Redoop system maintains caches at two stages of a MapReduce job, reduce input and output. Both cached data need not to be loaded, processed or shuffled again with the same mapper across windows. On the other hand, Helix system avoid redundant I/Os and computations by sharing map input scan, map function, and map outs.

Hence, the proposed techniques in Redoop and Helix are naturally fit together without any change at all to process recurring workloads. In summary, our Redoop and Helix techniques can co-exist in our recurring query processing architecture without further extension.

Next we analyze the scenario of integrating Redoop with Faro. In this scenario, Faro's pre-map and post-map sampling techniques can substantially reduce the amount of data to be processed by the reducers. Therefore, the amount of data can be cached and then be reused by Redoop system can be much less or even negligible. Consequently, the I/O and computational savings gained from Faro's sampling technique are the dominant factor compared to the one from reduce input cache and reduce output cache in Redoop system. In the worst case scenario, the overheads of caching and maintaining the reduce input and output data could offset the benefits of I/O and computational savings due to the relatively small file sizes. Apparently, future research is needed to tackle this issue. Specifically, an adaptive optimizer should be plugged in to make decision on whether enabling or disabling the Redoop caching mechanisms depending on the statistics collected from the actual executions.

Now we analyze the scenario of integrating three systems as a whole infrastructure. As mentioned in Chapter 3, Redoop serves as a bedrock for recurring query processing. Built upon Redoop, Helix and Faro are designed for sharing similar workloads and providing approximate results for tight time constraints, respectively. The new challenge is basically to make a resource allocation decision depending on different purposes, including cache usability, sharing benefits, and result accuracy gain. As described in both Chapters 10 and 15, the resource allocation/scheduling is in general a hard problem. This is clearly one of the directions to explore in future work.

Further, the approximate techniques in Faro are optimized for the single recurring query processing. Extending it to support approximate processing for multiple queries is obviously a future direction. Similar to BlinkDB, we need to optimize a set of samples for all queries sharing common MapReduce tasks. Recall that the number of samples we read to satisfy a query will vary according to user-specified time bounds. So in general we want access to a family of samples, one for each possible value of number of samples. We can organize queries into a dissemination graph to exploit the dependencies across queries. In this way, recurring queries closer to the root (source of data flow) can potentially be used to provide sample data of descendant/dependent queries. We can exploit some statistical approach to combine answers from ancestor nodes to generate the results for a node.

To further seamlessly integrate three proposed techniques, a high-level interface and query optimization is needed on top of these techniques. For example, Hadoop has several high-level interfaces, e.g., Pig [5], Hive [60], and Jaql [116] that enable end-users to express their queries and workflows abstractly using high-level constructs. Ideally, we should adopt one existing interface and then extend its language to accommodate our proposed extensions in the recurring query model. These include binding the inputs and outputs to data locations, defining execution specification parameters, defining properties that can be utilized in optimizations. We also need to extend the compiler/optimizer components for the language to: (1) generate the low-level map-reduce-finalize functions, (2) based on the parameters specific to recurring queries, the optimizer may decide to trigger appropriate optimization techniques.

## 19.2 Velocity and Variety in Recurring Query Processing

A major part of the challenge in data analytics today comes from the sheer volume of data available for processing. Data volumes that many companies want to process in timely and cost-efficient ways have grown steadily from the multi-gigabyte range to terabytes and now to many petabytes. A major part of the recurring query processing techniques that we presented in this dissertation were aimed at handling such large data sets. This challenge of dealing with very large data sets has been termed the volume challenge. There are two other related challenges, namely, those of *velocity* and *variety*.

The *velocity* challenge refers to the short response-time requirements for collecting, storing, and processing data. Though the fundamental system that we proposed in this dissertation is a batch system, we are aware of its insufficiency for latency sensitive applications, such as identifying potential fraud and recommending personalized content, batch data processing is insufficient. We proposed approximate processing techniques to tackle the velocity challenge from one aspect. However, there is an increasing appetite towards getting query results faster when the data streams into the system. A mixed- or memory-based system is clearly future directions in the area of velocity. In the following section, we highlight the details of these directions.

The *variety* challenge refers to the growing list of data types—relational, time series, text, graphs, audio, video, images, genetic codes—as well as the growing list of analysis techniques on such data. New insights are found while analyzing more than one of these data types together. The recurring query processing techniques that we have proposed in this dissertation are predominantly aimed at handling data that can be represented using a relational model (rows and columns) and processed by query plan operators like filters, joins, and aggregation. However, the new and emerging data types cannot be captured easily in a relational data model, or analyzed easily by software that depends on running operators like filters, joins, and aggregation. Instead, the new and emerging data types need a variety of analytical techniques such as linear algebra, statistical machine learning, text search, signal processing, natural language processing, and iterative graph processing.

### 19.2.1 Handling Velocity

**Memory-based Hadoop.** Given the steadily increasing memory sizes in commodity servers, a memory-based solution should be considered as an alternative to disk-based Hadoop system. All data should be kept in main memory as long as there is enough space available. All data structures should be optimized for cache-efficiency instead of being optimized for organization in traditional disck blocks. Furthermore, the memory-based system should compress the data using a variety of compression schemes. When the limit of available main memory is reached, entire data objects, e.g., tables or partitions, are unloaded from main memory under the control of application semantics and reloaded into main memory when they are required again. While virtually all data is kept in main memory by the processing engines for performance reasons, data is stored by the persistence layer for backup and recovery in case of a system restart after a shutdown or a failure.

Memory-based extensions and improvements on Hadoop system have also been proposed. M3R (Main Memory MapReduce) [46] is a framework that extends Hadoop for running MapReduce jobs in memory. M3R caches input and output data in memory, performs in-memory shuffling, and always maps the same partition to the same location across all jobs in a sequence in order to allow for the re-use of already built memory structures. PowerDrill [117] is a column-oriented datastore similar to Dremel, but it relies on having as much data in memory as possible. PowerDrill uses two dictionaries as basic data structures for representing a data column and employs several optimizations for keeping the memory footprint of these structures small.

**Stream Processing Systems.** As motivated in Chapter 1.2, timely analysis of activity and operational data is critical for companies to stay competitive. Activity data from a company's Web-site contains page and content views, searches, as well as advertisements shown and clicked. This data is analyzed for purposes like behavioral targeting, where personalized content is shown based on a user's past activity, and showing advertisements or recommendations based on the activity of her social friends. Operational data includes monitoring data collected from Web applications (e.g., request latency) and cluster resources (e.g., CPU usage). Proactive analysis of operational data is used to ensure that Web applications continue to meet all service-level requirements.

The vast majority of analysis over activity and operational data involves continuous queries over a data source that is constantly updated. Continuous queries arise naturally over activity and operational data because of two reasons: (1) the data is generated continuously in the form of append-only streams; (2) the data has a time component such that recent data is usually more relevant than older data. The growing interest in continuous queries is reflected by the engineering resources that companies have recently been investing in building continuous query execution platforms. Yahoo! released S4 [118] in 2010, Twitter released Storm [119] in 2011, and Walmart Labs released Muppet [120] in 2012. Also prominent are recent efforts to add continuous querying capabilities to the popular Hadoop platform for batch analytics. Examples include the Oozie workflow manager [45], MapReduce Online [53], and Facebooks real-time analytics system [43].

**Hybrid Systems.** All the above techniques, however, are imperfect solutions for a few reasons: it still requires the developer to build and maintain code that binds to different execution frameworks. In many cases, two sets of aggregation logic must be created due to the inherent differences between batch and online processing. Moreover, when writing code that is supposedly agnostic to the processing model, it is easy to forget the constraints of the execution environment. For example, scaling out in Hadoop is often as simple as increasing the number of reducers, but the ability to scale out in an online environment by splitting streams is more restrictive. Thus, it is not uncommon to prototype a particular feature in Hadoop and then discover that the implementation is too slow to run in an online production setting. As another example: in batch processing, it is possible to take advantage of disk storage if in-memory data structures grow too large, but in online processing this is usually not possible due to latency requirements. Managing memory limitations is particularly important when trying to track large event spaces.

In reality, the biggest challenge is developer productivity, not runtime performance, since ultimately, what runs is either a "vanilla" Hadoop job or Storm topology. Thus, the contributions of the language lie in the abstractions it introduces and its balance between simplicity and expressiveness with respect to a broad range of analytical queries. The ability to integrate batch and offline analytics should be supported by a hybrid processing model [121] where we are able to efficiently and seamlessly provide access to aggregations across long time spans while maintaining up-to-date values with minimal latency.

The key insight is that certain algebraic structures provide the theoretical foundation for seamlessly integrating batch and online processing. From this starting point, we need a data processing framework that supports both batch and online computations formulated in terms of these algebraic structures. Such framework should provide a domain-specific language for expressing analytical queries that transparently generates either Hadoop jobs (batch computations) or Storm topologies (online computations) without requiring any changes to the program logic. Furthermore, such system should be able to operate in a hybrid processing mode that transparently integrates batch and online results to efficiently generate up-to-date views over long time spans. The desired language should be sufficiently expressive to capture large classes of analytical queries in a production environment.

The basic idea behind hybrid processing is to periodically "roll up" aggregates using Hadoop and to "fill in" results from real-time data using Storm. The desired architecture should integrate batch and online results while transparently preserving correctness with two key features.

1. Hybrid processing should not require changing the logic of programs the same

exact pro- gram runs in either batch or online mode. The only additional requirements from the developers perspective are a few metadata extractors to define how inputs are grouped in batches and a modest number of hooks into other parts of Twitters infrastructure. Any additional bookkeeping is performed behind the scenes without the developers knowledge.

2. Downstream clients are completely shielded from the details of the hybrid processing. Integration of results from Hadoop and Storm are transparently handled by the client library, which presents a simple key/value interface.

The complete architecture of such system running in hybrid mode is shown in Figure 19.1.

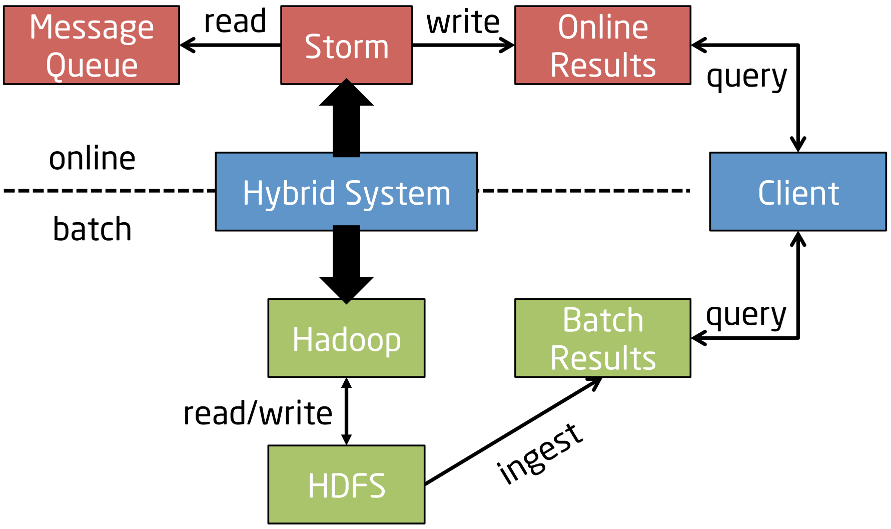


**Figure 19.1:** System Architecture for Hybrid Processing

The hybrid system requires integration with other infrastructure. On the source end, we assume the existence of message queues that deliver event data in real-time and that the same data are also deposited onto HDFS. On the store end, we assume the existence of two separate key/value stores: one for the batch results, and the other for the online results (although the client library transparently handles results merging).

Periodically, a users job on the Hadoop platform is triggered to compute aggregates on the next incremental source batch that has been deposited in HDFS. The process of physically launching these jobs is accomplished through resource allocator. The mapping from batch ids to physical HDFS paths can be deterministically computed since data are structured according to a certain physical layout. The data import pipeline is engineered so that a directory does not appear until all the files contained in that directory have arrived, so it is not possible to process partially-imported results.

The batch results key/value store polls HDFS periodically for the appearance of newly-created stores, and when one appears, the contents are ingested. Since the key/value pairs on HDFS capture results for only that source batch, the ingestion process requires applying the semi-group associative operator to aggregate those key/value pairs with the current contents of the batch results store. However, instead of storing (K,V) pairs directly, the contents are transformed into (K, (batchId, V)) pairs and this data structure captures the value of a particular key up to and including the specified batch id. This transformation should be performed "behind the scenes" without the developers knowledge.

In parallel with the batch jobs, the same user's program is continuously executed in a Storm topology, and the results are deposited in an online results key/value store. Instead of aggregating by key K, however, the system automatically builds a compound key (K, batchId) for performing the grouping. These represent the online partial results for each batch.

The client side maintains connections to both the online and batch results store. All queries first go to the batch results store: by comparing the wall clock time and the batch id from the result, the system knows how "far behind" the value is. Based on this, the client can figure out how many values need to be "filled in" from the online store, which is keyed by (K, batchId). It can then issue appropriate requests to the online results store.

The final, up-to-date value is arrived at by aggregating all the partial values once again, the validity of these operations is licensed by the fact that the values are (at least) semigroups.

Typically, the batch results key/value store is much larger and backed by durable storage, whereas the online results are kept in memory-resident key/value stores (e.g., memcached). To prevent memory overflow, keys are pruned based on a time-to-live (TTL) setting. The TTL is tuned such that, under normal operating circumstances (and within a "margin of safety"), there will be no "gap" between the batch and online results. That is, largest batch id in the batch results store will be greater than the smallest batch id in the online results store. However, during times of excessive load on the Hadoop cluster or outages, gaps may appear when the online results coverage is not sufficient to fill in where the batch results end. In this case, a client lookup will fail.

### 19.2.2 Handling Variety (Graph Processing)

For a growing number of applications, the data takes the form of graphs that connect many millions of nodes. The growing need for managing graph-shaped data comes from applications such as: (1) identifying influential people and trends propagating through a social-networking community, (2) tracking patterns of how diseases spread, and (3) finding and fixing bottlenecks in computer networks.

The analysis needs of such applications not only include processing the attribute values of the nodes in the graph, but also analyzing the way in which these nodes are connected. The relational data model can be a hindrance in representing graph data as well as expressing analysis tasks over this data especially when the data is distributed and has some complex structure. Graph databases—which use graph structures with nodes, edges, and their properties to represent and store data—are being developed to support such applications.

There are many techniques for how to store and process graphs. The effectiveness of these techniques depend on the amount of data—the number of nodes, edges, along with the size of data associated with them—and the types of analysis tasks, e.g., search and pattern matching versus more complex analytics tasks such as finding strongly connected components, maximal independent sets, and shortest paths.

Many graph databases such as Pregel [34] use the Bulk Synchronous Parallel (BSP) computing model. Like the map and reduce functions in MapReduce, Pregel has primitives that let neighboring nodes send and receive messages to one another, or change the state of a node (based on the state of neighboring nodes). Graph algorithms are specified as a sequence of iterations built from such primitives. GraphLab uses similar primitives (called PowerGraph) but allows for asynchronous iterative computations [122]. GraphX runs on Spark and introduces a new abstraction called Resilient Distributed Graph (RDG). Graph algorithms are specified as a sequence of transformations on RDGs, where a transformation can affect nodes, edges, or both, and yields a new RDG [123].

Techniques have also been proposed to support the iterative and recursive computational needs of graph analysis in the categories of systems that we have considered in this monograph. For example, HaLoop and Twister are designed to support iterative algorithms in MapReduce systems [81, 82]. HaLoop employs specialized scheduling techniques and the use of caching between each iteration, whereas Twister relies on a publish/subscribe mechanism to handle all communication and data transfers. Efficient techniques to run recursive algorithms needed in machine-learning tasks are supported by the Hyracks dataflow system [124].

A natural idea is to reconsider the nature of graph analytics and resort to recurring query processing for performance boosting. Namely, we wish to improve the efficiency of the graph processing with the power of recurring query processing model and optimization techniques. Specifically, we expect to peruse the following directions.

1. Incremental algorithms compute changes to the matches in response to updates, to minimize unnecessary re-computation. We plan to investigate incremental algorithms for graph patter matching, a routine process in emerging applications such as social networks. For certain graph patterns, the algorithms should be in linear time in the size of the changes in the input and output, which characterized the cost that is inherent to the problem itself. For general patterns, the incremental matching problem is unbounded, i.e., its cost is not determined by the size of the changes alone. Hence we may consider exploiting weighted landmark vectors, an extension of landmarks [125], to help us find shortest paths between node pairs in a graph. Further, we can provide a lazy incremental algorithm that updates the landmarks only when necessary.

2. Approximate algorithms give users some answers with reasonable accuracy and high efficiency for a wide spectrum of graph analytics tasks. The approach should be orthogonal to the approaches that design an approximation algorithm for a specific graph problem. The proposed algorithms should be seamlessly integrated with the proposed optimization techniques for recurring query processing. Hence it can be benefited from all performance improvements from the underlying system.

3. Asynchronous parallel execution can be explored to alleviate the overheads of the bulk synchronous parallel model used by synchronous graph processing systems, including stale messages and frequent global synchronization barriers. Existing asynchronous systems have limited scalability or retain frequent global barriers, and do not always support graph mutations or algorithms with multiple computation phases. Our goal is to design barrierless asynchronous processing model that reduces both message staleness and global synchronization. This enables our system to overcome the limitations of existing asynchronous models while retaining support for graph mutations and algorithms with multiple computation phases.

In summary, it would be interesting and promising to explore these three directions.

We hope the experimental evaluation can confirm the superiority of the designs/models above. Eventually, future work in this direction will expand the applicability of the recurring query processing system to a wider spectrum.

# References

[1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004. 1, 18, 72

[2] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org`. 1, 10, 20, 72

[3] John Russell. Couldera-Impala. *O'Reilly Media*, 2013. 1

[4] M. Traverso. Presto: Interacting with petabytes of data at Facebook. 2013. 1

[5] *Pig*. `http://hadoop.apache.org/pig`. 1, 22, 72, 157

[6] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, pages 1626–1629, 2009. 1

[7] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 267–273, Berkeley, CA, USA, 2008. USENIX Association. 1

[8] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava.

Automatic optimization of parallel dataflow programs. In *USENIX*, pages 267–273, 2008. 1

[9] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 international conference on Management of data*, pages 975–986, 2010. 1

[10] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: operator scheduling for memory minimization in data stream systems. In *ACM SIGMOD*, pages 253–264, 2003. 1

[11] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003. 1

[12] B. Babcock, S. Babu, R. Motwani, and J. Widom. Models and issues in data streams. In *PODS*, pages 1–16, June 2002. 1

[13] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, Sep 2003. 1

[14] Tim Sutherland, Brad Pielech, Yali Zhu, Luping Ding, and Elke A . Rundensteiner. An adaptive multi-objective scheduling selection framework for continuous query processing. In *IDEAS*, pages 445–454, July 2005. 1

[15] Aparna S. Varde, Elke A. Rundensteiner, Carolina Ruiz, Mohammed Maniruzzaman, and Richard Sisson. Learning semantics-preserving distance metrics for clustering graphical data. In *MDM '05: Proceedings of the 6th international workshop on Multimedia data mining*, pages 107–112, New York, NY, USA, 2005. ACM. 1

[16] Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. Nugget discovery in visual exploration environments by query consolidation. In *CIKM*, pages 603–612, 2007. 1

[17] Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. Neighbor-based pattern detection for windows over streaming data. *EDBT*, 2009. To Appear. 1

[18] Sayan Ranu and Ambuj K Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 844–855. IEEE, 2009. 1

[19] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7), 2014. 1

[20] Hao Huang, Yunjun Gao, Kevin Chiew, Lei Chen, and Qinming He. Towards effective and efficient mining of arbitrary shaped clusters. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 28–39. IEEE, 2014. 1

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003. 1

[22] The Apache Software Foundation. HDFS architecture guide. `http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html`. 1

[23] Flavio Chierichetti, Nilesh Dalvi, and Ravi Kumar. Correlation clustering in mapreduce. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 641–650. ACM, 2014. 1

[24] Charu C Aggarwal. *Managing and mining sensor data.* Springer, 2013. 1

[25] Dawei Jiang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Sai Wu. epic: an extensible and scalable system for processing big data. *Proceedings of the VLDB Endowment*, 7(7), 2014. 1

[26] ShirishTatikonda MatthiasBoehm, PrithvirajSen BertholdReinwald, DouglasR YuanyuanTian, and ShivakumarVaithyanathan Burdick. Hybrid parallelization strategies for large-scale machine learning in systemml. *Proceedings of the VLDB Endowment*, 7(7), 2014. 1

[27] Sergej Fries, Stephan Wels, and Thomas Seidl. Projected clustering for huge data sets in mapreduce. In *EDBT*, pages 49–60, 2014. 1

[28] Kevin Loney. *Oracle Database 10g : The Complete Reference*. McGraw-Hill, 2004. 1

[29] Microsoft Inc. Microsoft SQL Server. http://www.microsoft.com/sql/default.asp. 1

[30] Huan Liu and Dan Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In *CCGRID*, pages 295–305, 2008. 2

[31] Christopher Olston, Edward Bortnikov, Khaled Elmeleegy, Flavio Junqueira, and Benjamin Reed. Interactive analysis of web-scale data. In *CIDR*, 2009. 2

[32] Andrew Pavlo and et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009. 2

[33] Daniel J. Abadi. Tradeoffs between parallel database systems, hadoop, and hadoopdb as platforms for petabyte-scale analysis. In *SSDBM*, pages 1–3, 2010. 2

[34] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale

graph processing. In *Proceedings of the 2010 international conference on Management of data*, pages 135–146, 2010. 2, 165

[35] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, pages 1–2, 2009. 2

[36] David J. DeWitt, Erik Paulson, Eric Robinson, Jeffrey Naughton, Joshua Royalty, Srinath Shankar, and Andrew Krioukov. Clustera: an integrated computation and data management system. *Proc. VLDB Endow.*, pages 28–41, 2008. 2

[37] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130, 2010. 2

[38] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, pages 494–505, 2010. 2, 8, 12, 22, 87, 111, 119

[39] Iman Elghandour and Ashraf Aboulnaga. Restore: reusing results of mapreduce jobs in pig. In *SIGMOD Conference*, pages 701–704, 2012. 2

[40] Guoping Wang and Chee-Yong Chan. Multi-query optimization in mapreduce framework. *PVLDB*, 7(3):145–156, 2013. 2, 8, 11, 12, 22, 87, 89, 100, 112, 119

[41] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, et al. In-situ mapreduce for log processing. In *USENIXATC*, pages 9–9, 2011. 2, 5, 7, 73, 74

[42] Roshan Sumbaly, Jay Kreps, and Sam Shah. The big data ecosystem at linkedin. In *SIGMOD*, pages 1125–1134, 2013. 2

[43] Facebook. *http://www.facebook.com*. 6, 160

[44] Christopher Olston, Greg Chiou, Laukik Chitnis, et al. Nova: continuous pig/hadoop workflows. In *SIGMOD*, pages 1081–1090, 2011. 7, 73

[45] *Apache. Oozie: Hadoop workflow system.* http://yahoo.github.com/oozie/. 7, 160

[46] Avraham Shinnar, David Cunningham, Benjamin Herta, et al. M3r: Increased performance for in-memory hadoop jobs. *PVLDB*, pages 1736–1747, 2012. 7, 73, 159

[47] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, pages 985–996, 2011. 7, 43, 58, 73

[48] Nathan Backman, Karthik Pattabiraman, Rodrigo Fonseca, et al. C-mr: continuously mapreduce workflows on multi-core processors. In *Proceedings of 3rd international workshop on MapReduce and its Applications Date*, pages 1–8, 2012. 7, 73, 74

[49] Iman Elghandour and Ashraf Aboulnaga. Restore: reusing results of mapreduce jobs. *Proc. VLDB Endow.*, 5(6):586–597, 2012. 8, 73, 111

[50] Ganesh Ananthanarayanan, Christopher Douglas, et al. True elasticity in multi-tenant data-intensive compute clusters. In *SoCC*, pages 24:1–24:7, 2012. 8, 113, 150

[51] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic

resource inference and allocation for mapreduce environments. In *ICAC*, pages 235–244, 2011. 8, 113, 150

[52] Zhuoyao Zhang, Ludmila Cherkasova, Abhishek Verma, and Boon Thau Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *ICAC*, pages 53–62, 2012. 8, 113, 150

[53] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD*, pages 1115–1118, 2010. 9, 160

[54] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on mapreduce. *PVLDB*, 5(10):1028–1039, 2012. 9, 28, 139, 147, 149

[55] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS*, pages 383–397, 2015. 9, 149

[56] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *SIGMOD Conference*, page 668, 2003. 10

[57] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: A data stream management system. In *SIGMOD Conference*, page 666, 2003. 10, 74, 150

[58] The Apache Software Foundation. http://hbase.apache.org/. 22

[59] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06*. USENIX Association, 2006. 22

[60] *Hive*. `http://hadoop.apache.org/hive`. 22, 72, 157

[61] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009. 22

[62] Yingjie Shi, Xiaofeng Meng, Fusheng Wang, and Yantao Gan. You can stop early with cola: Online processing of aggregate queries in the cloud. In *CIKM*, pages 1223–1232, 2012. 29

[63] Nicolas Bruno, Sameer Agarwal, et al. Recurring job optimization in scope. In *SIGMOD*, pages 805–806, 2012. 30, 110, 149

[64] Chuan Lei, Elke A. Rundensteiner, and Mohamed Eltabakh. Redoop: Supporting recurring queries in hadoop. In *EDBT*, pages 817–828, 2014. 30, 32, 82, 99, 110, 140, 141, 149, 150

[65] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013. 31, 148

[66] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. In *NSDI*, pages 289–302, 2014. 31

[67] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997. 31, 148

[68] S. Lohr. *Sampling: Design and Analysis.* Brooks/Cole, 1999. 31, 118, 119, 127

[69] Minos N. Garofalakis and Phillip B. Gibbon. Approximate query processing: Taming the terabytes. In *VLDB*, page 725, 2001. 31

[70] Jin Li, David Maier, Kristin Tufte, et al. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, pages 39–44, 2005. 35, 39, 72, 79

[71] Song Wang, Elke Rundensteiner, Samrat Ganguly, et al. State-slice: new paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006. 35, 72

[72] Lukasz Golab. *Sliding window query processing over data streams.* PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 2006. AAINR23520. 35, 72

[73] Erik Zeitler and Tore Risch. Massive scale-out of expensive continuous queries. *PVLDB*, pages 1181–1188, 2011. 42

[74] Bin Liu, Yali Zhu, Mariana Jbantova, et al. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005. 42

[75] Mehul S. Joseph, Joseph M. Hellerstein, et al. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2002. 42

[76] Chris Chatfield. The holt-winters forecasting procedure. *Applied Statistics*, pages 264–279, 1978. 44

[77] 1998 world cup. http://ita.ee.lbl.gov/html/contrib/WorldCup.html. 64, 99, 139

[78] Soccer - real time tracking system. http://www.iis.fraunhofer.de/en/bf/ln/referenzprojekte/redfir.html. 64, 99, 139

[79] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, et al. Parallel data processing with mapreduce: a survey. *SIGMOD Rec.*, pages 11–20, 2012. 72

[80] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, pages 313–328, 2010. 72, 149

[81] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, 2010. 72, 165

[82] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, 2010. 72, 165

[83] Ahmed M. Aly, Asmaa Sallam, Bala M. Gnanasekaran, et al. M3: Stream processing on main-memory mapreduce. In *ICDE*, pages 1253–1256, 2012. 73

[84] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005. 74, 150

[85] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and

Jennifer Widom. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003. 74

[86] Chuan Lei, Elke A. Rundensteiner, and Joshua D. Guttman. Robust distributed stream processing. In *ICDE*, pages 817–828, 2013. 74

[87] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006. 80

[88] Anton J. Kleywegt and Jason D. Papastavrou. The dynamic and stochastic knapsack problem. *Operations Research*, 46:17–35, 1998. 87, 92

[89] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of mapreduce pipelines. In *ICDE*, pages 681–684, 2010. 87, 121

[90] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *PVLDB*, 7(6):429–440, 2014. 90

[91] Venkatesh Raghavan and Elke A. Rundensteiner. Caqe: A contract driven approach to processing concurrent decision support queries. In *EDBT*, pages 121–132, 2014. 99

[92] Nicolas Bruno, Sapna Jain, and Jingren Zhou. Recurring job optimization for massively distributed query processing. *IEEE Data Eng. Bull.*, 36(1):46–55, 2013. 110, 149

[93] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190 – 200, 1995. 110

[94] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000. 110

[95] K-L Tan, Shen-Tat Goh, and Beng Chin Ooi. Cache-on-demand: Recycling with certainty. In *ICDE*, pages 633–640, 2001. 110

[96] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of VLDB Conference*, pages 297–308, 2003. 111

[97] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 623–634. ACM, 2006. 111

[98] Song Wang, Elke Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd international conference on Very large data bases*, pages 619–630. VLDB Endowment, 2006. 111

[99] Alexandros Labrinidis, Huiming Qu, and Jie Xu. Quality contracts for real-time enterprises. In *BIRTE*, pages 143–156, 2007. 112, 150

[100] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010. 112, 150

[101] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceed-*

*ings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42, 2008. 121

[102] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G. Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, pages 265–278, 2010. 121

[103] Rainer Gemulla and Wolfgang Lehner. Sampling time-based sliding windows in bounded space. In *SIGMOD*, pages 379–392, 2008. 121

[104] A. Federgruen and H. Groenevelt. The greedy procedure for resource allocation problems: Necessary and sufficient conditions for optimality. *Oper. Res.*, 34(6):909–918, December 1986. 130

[105] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *SoCC*, pages 15:1–15:12, 2012. 131

[106] Di Yang, Avani Shastri, Elke A. Rundensteiner, and Matthew O. Ward. An optimal strategy for monitoring top-k queries in streaming windows. In *EDBT*, pages 57–68, 2011. 136

[107] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2), June 2007. 148

[108] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4):23:1–23:54, December 2008. 148

[109] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004. 148

[110] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006. 148

[111] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*, pages 651–662, 2010. 148

[112] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, 1999. 148

[113] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD*, pages 275–286, 1999. 148

[114] Raman Grover and Michael J. Carey. Extending map-reduce for efficient predicate-based sampling. In *ICDE*, pages 486–497, 2012. 149

[115] Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, and Mohamed Y. Eltabakh. Shared execution of recurring workloads in mapreduce. *PVLDB*, 8(7):714–725, 2015. 149, 150

[116] *Jaql*. `http://code.google.com/p/jaql`. 157

[117] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, and Marc Nunkesser. Processing a trillion cells per mouse click. *Proc. VLDB Endow.*, 5(11):1436–1446, 2012. 159

[118] The Apache Software Foundation. S4. http://incubator.apache.org/s4/. 160

[119] The Apache Software Foundation. Storm. https://storm.apache.org/. 160

[120] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. *Proc. VLDB Endow.*, 5(12):1814–1825, August 2012. 160

[121] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, December 2014. 161

[122] 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. *Proc. VLDB Endow.*, 5(8):716–727, April 2012. 165

[123] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, pages 2:1–2:6, 2013. 165

[124] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011. 165

[125] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast shortest path distance estimation in large networks. In *ACM CIKM*, pages 867–876, 2009. 166