Shared Execution of Recurring Workloads in MapReduce*

Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, and Mohamed Eltabakh

Worcester Polytechnic Institute, Worcester, MA USA
(chuanlei,zzhuang,rundenst,meltabakh)@cs.wpi.edu

ABSTRACT

With the increasing complexity of data-intensive MapReduce workloads, Hadoop must often accommodate hundreds or even thousands of recurring analytics queries that periodically execute over frequently updated datasets, 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. The recurring nature of these emerging workloads combined with their SLA constraints make it challenging to share and optimize their execution. While some recent efforts on multi-job optimization in MapReduce have emerged, they focus on only sharing work among ad-hoc jobs on static datasets. Unfortunately, these sharing techniques neither take the recurring nature of the queries into account nor guarantee the satisfaction of the SLA requirements. In this work, we propose the first 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. And then, 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. Our experimental results over real-world datasets confirm that Helix significantly outperforms the state-of-art techniques by an order of magnitude.

1. INTRODUCTION

MapReduce has recently emerged as a new paradigm for large-scale data analytics due to its high scalability, fault tolerance, and flexible programming model. Companies such as Google, Amazon, Facebook, LinkedIn, and many others have embraced MapReduce, and its open-source implementation Hadoop, to perform large-scale analytical applications on massive evolving datasets using recurring queries [5, 13]. These recurring queries are periodically re-executed on data subsets identified by a sliding window on the evolving data, e.g., processing the last n hours, days, weeks, or even months worth of data depending on the granularity of interest. Nevertheless, with the increasing complexity of these recurring workloads, Hadoop may need to execute hundreds or even thousands of these queries at the same time [8]. Typically, many queries within a given workload perform similar tasks, e.g., accessing the same input files, having partial or total computations in common, feeding the results of one task to another task, etc. Therefore, significant gains could be achieved if we can exploit these sharing opportunities. However, the diversity and variation in these recurring queries w.r.t their window, slide, and SLA constraints combined with the evolving nature of the data, i.e., not static input, pose new complications and challenges to the sharing techniques.

Recent work has emerged to address multiple query sharing in MapReduce systems [7, 16, 23]. However, these sharing techniques target traditional ad-hoc map-reduce jobs. They take neither the recurring nature of our target workload nor the embedded service-level agreements (SLAs) in these queries into account. The following examples showcase the unique characteristics of recurring queries, and highlight the new sharing challenges and opportunities the are overlooked in literature.

Motivating Example - Social Network Analytics. Social network companies, such as Facebook, Twitter, and LinkedIn, rely on online news services to extract insights and build product features from massive datasets [4, 19, 21]. For example, a LinkedIn member may receive periodic updates from various recurring applications, including "people you may know", skill endorsement, collaborative filtering, and so on. A common theme among these recurring applications is the large set of feature extraction tasks, e.g., the number of common connections, company and school overlap, geographical distance, age similarity, among many others. These tasks are then followed by model building and verification tasks. These tasks are often specified as recurring queries performing similar work over the latest batches of data over time. Consider the following three recurring queries:

• Query 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 (SLA constraint) such that she can refine her search and quickly identify potential candidates before moving on to other tasks.

• Query 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).

*This project is supported by NSF grants CNS-1305258 and IIS-1018443.

Proceedings of the VLDB Endowment, Vol. 8, No. 7

• Query 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.

Computing the above queries involves deep analytical processing over large-scale data sets across multiple sources. However, these queries differ in their requirements and constraints, e.g., the SLA requirements for the three queries above vary significantly. Without taking these SLAs into consideration, existing multi-query optimization (MQO) techniques [16, 23] in MapReduce would choose to share the above three queries as one single map-reduce job to minimize the I/O overheads and the total execution time. However, this decision may fail to meet q_1's SLA requirements since it has a tight constraint and may be penalized by being forced into sharing with q_3, which involves a data-intensive mining task. In contrast, an optimal solution could be the sharing between q_2 and q_3, while executing q_1 independently. Worse yet, the evolving nature of data sources - in contrast to the static sources explored by the current techniques - adds more complications. For example, without careful intervention, sharing q_1 and q_2 may not be beneficial due to their different job granularities, i.e., q_1 (w = 1 week, s = 1 day), versus q_2 (w = 1 day, s = 1 hour).

Many data analytics applications, such as search query log processing [4], web crawling [3], and more, share similar characteristics as those illustrated in the above mentioned example. These applications involve the execution of many recurring jobs that may differ in their window constraints, SLAs, or both. Without taking these constraints into account, the sharing strategy employed by the state-of-art techniques [7, 16, 23] may cause more harm than good, e.g., missing important deadlines, and missing critical optimization opportunities.

State-of-the-Art Techniques. The closest related work falls into two major categories, namely *MQO* sharing techniques, and *SLA-aware* strategies in MapReduce. Recently, MQO techniques for sharing similar work among a batch of ad-hoc map-reduce jobs have emerged [7, 16, 21, 23]. These techniques fall short in identifying and leveraging optimization opportunities unique to recurring queries. Wang et al. [23] proposed to group multiple jobs into a single job, and to share materialized map outputs among jobs. However, their solution, being oblivious to SLAs and window constraints, does not provide any system-level optimizations for recurring queries.

On the other hand, recent efforts in SLA-aware MapReduce techniques [3, 22, 25] 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. The proposed Helix system is the first to combine the three worlds of *multi-query sharing*, *recurring query execution*, and *SLA-driven computations* together into a single coherent system.

Challenges. Sharing multiple recurring queries in MapReduce is an NP-hard problem. Its sub-problem, MQO in MapReduce, has been already established as an NP-hard problem [23]. 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 of 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 Datasets. Existing shared execution techniques in MapReduce [16, 23] assume that the inputs to their map-reduce jobs are static. In contrast, recurring queries consume evolving datasets 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.

Contributions. To address the above challenges, we propose the *"Helix"* system, *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 [9]. 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 key contributions include:

• 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.

• We propose techniques for solving the *unmatched of Interest* problem over evolving data sources. We introduce the *sliced window alignment* strategy for pre-processing 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.

• 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 optimizer 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.

• 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.

The rest of the paper is organized as follows. Section 2 introduces the recurring query model and the sharing techniques in MapReduce. Section 3 describes the window alignment techniques. Section 4 presents the proposed Helix optimization technique. Sections 5 and 6 discuss experimental results and related work, respectively. Section 7 concludes the paper.

2. PRELIMINARIES

Next, we introduce the recurring query model followed by a review of sharing techniques in MapReduce.

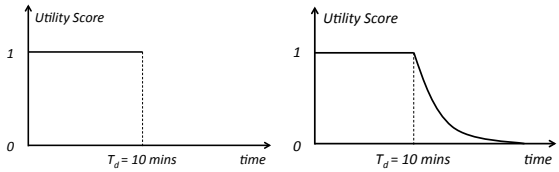
2.1 Recurring Query Model

Query Parameters. Recurring queries in MapReduce are first introduced in [13]. They execute periodically over evolving disk-resident datasets, i.e., datasets stored in HDFS. In each execution, a recurring query *q(w, s)* bounds its computations over the evolving datasets by two configuration parameters *w* and *s*. The window *w* specifies the scope of data to process, while the slide *s* specifies the frequency of execution. For example, *q(w = 12 hours, s = 1 hour)* specifies a recurring query that executes every hour and processes all data within the last 12 hours.

Input Timestamps. Recurring queries process evolving data over time. Thus the timestamps associated with the data are important in our model. A data batch f_i received at time T_{fi} is annotated by the time T_{fi}. We assume that 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 $[T_i, T_{f1})$, $[T_{f1}, T_{f2})$, ... , $[T_{fn-1}, T_{fn})$ with $T_{fi} < T_{fi+1}$. Therefore, there is an order among the files, but there are no order constraints among the tuples within each file. The above model is common in most 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_{f1}, T_{f2}, ..., T_{fn}, where $T_i < T_{f1} < T_{f2} < ... < T_{fn} < T_j$.

Execution Model. The Redoop system [13] is proposed to treat recurring queries as first-class citizen in the MapReduce infrastructure. A recurring query *q(w, s)* is registered in Redoop, where the *w* and *s* properties are defined as configuration parameters. Once registered, Redoop periodically triggers the execution of *q* according to its *w* and *s* parameters. The evolving inputs of *q* are consumed from a specific HDFS directory, while the outputs are also periodically produced to another HDFS directory. Given a recurring query *q(w, s)*, 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 w slides by 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.

The proposed Helix system overcomes the limitations of Redoop by enabling the sharing among multiple recurring queries and specifying the queries' SLAs. We extend Redoop's query model to enable the specifications of the SLAs, i.e., the query is defined as *q(w, s, SLA)*. The SLA parameter is expressed as a time-based function $\vartheta(t)$ that indicates the merit of the query's results (a utility score) if delivered at a certain time t. Figure 1 shows two sample SLAs. The function in Figure 1(a) indicates that the results produced after time T_d are useless to the application, i.e., its utility becomes zero. Figure 1(b) shows an SLA function that decreases the result's utility monotonically as the query execution exceeds T_d = 10 minutes. In general, Helix supports any non-increasing utility function as an SLA for recurring queries. In the remainder of this paper, we will use the SLA utility functions depicted in Figure 1 for demonstration purposes, both used in previous studies [17].



(a) SLA Function 1 (b) SLA Function 2

Figure 1: SLA Function

2.2 MapReduce Sharing Techniques

We now briefly review sharing techniques among ad-hoc queries in MapReduce and discuss the associated key observations [16, 23]. 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 Section 3, 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.

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.

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.

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.

The Redoop system [13] exploits reduce input sharing to eliminate the unnecessarily I/O costs resulting from the overlapping windows. It maintains the reduce inputs on its local file system for subsequent reuse. Therefore these reduce inputs need not to be loaded, processed or shuffled again across windows. Hence the processing time for recurring queries is reduced.

In Helix, all of the above techniques for sharing recurring map-reduce jobs are taken into account. The Helix's optimizer (Section 4) exploits all appropriate sharing techniques in its branch-and-bound optimization algorithm.

2.3 Helix Problem Definition

DEFINITION 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 (ϑ), 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*

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

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.

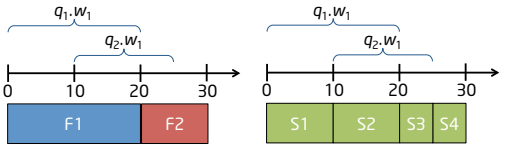
Finding the optimal solution for this problem is prohibitively expensive as discussed in Section 1. Orchestrating a shared execution of recurring queries with different SLAs is a stochastic knapsack problem, where the stochasticity comes from the fact that the utility score of a query is variable and depends on all previous decisions. Any solution to this problem would need to consider the unique characteristics of recurring queries.

3. WINDOW ALIGNMENT FOR RECURRING QUERIES

In this section, we present a new technique for the shared execution among multiple map-reduce recurring queries, call 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 (Section 3.1). And then, we propose a logical window slicing approach that partitions recurring query windows into aligned slices (Section 3.2). 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 Section 3.3, we presenting the mapping procedure from the logical sliced window to physical HDFS files.

3.1 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 recomputation, as illustrated in the following example.



(a) Naive Sharing (b) Sharing with SWA

Figure 2: Relation between Windows

EXAMPLE 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 2(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's resources.*

EXAMPLE 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 2(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.*

3.2 Aligning Queries in Sliced Windows

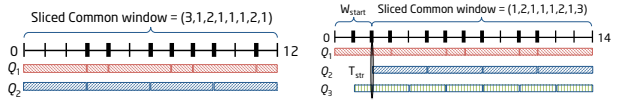
We now explain how to best partition an input data source for evaluating recurring queries that reduces both the required I/O operations and computation costs. The idea of partitioning a data stream into slices was first introduced by Li et al. [14] in the pane-based window approach. In this approach, 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 be 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 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 $r_i = (p_i.end - p_i.start)\ \forall\ 1 \leq i \leq m$. 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 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 [10] 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 Section 3.3.



(a) Common Sliced Window (b) Start time

Figure 3: Sliced Window Example

EXAMPLE 4. *Here we continue using Example 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 3(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 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 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 3(b)*

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 recomputation.

3.3 From Slices to Physical Files

After computing the optimal slicing boundaries for input preprocessing, 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 [13]. 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.

First, one slice corresponds to exactly one physical file. 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. 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 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.

4. SHARED RECURRING QUERY OPTIMIZATION

In this section, 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 Section 3.3. 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 subproblems is NP-hard, but also that they are interdependent, as illustrated in the following example.

EXAMPLE 6. *Continuing with our motivating example in Section 1, assume that the utility score for all three queries q_1, q_2, and q_3 follows the SLA function depicted in Figure 1(a), i.e., 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 w.r.t 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 3 (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.

DEFINITION 4 (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 $G_i \in GS$.*

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 [16, 23]. 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 6, 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 [9] 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 [15], 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 considered. Figure 4 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.

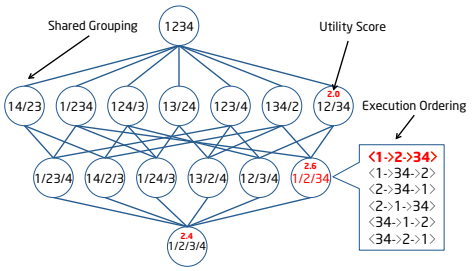


Figure 4: Lattice-Shaped Search Space

4.1 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.

4.1.1 Sharing Strategy

The goal of the proposed sharing strategy is to efficiently produce a good shared grouping solution SG. This solution can SG 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 Section 2.2 versus not exploiting them w.r.t the

utility score gain. The benefit value $bVal$ is obtained as follows:

$$bVal = \frac{\vartheta(t_{shared})}{Cost_{shared}(G_i)} - \frac{\sum_{j=1}^{|G_i|} \vartheta(t_j)}{\sum_{j=1}^{|G_i|} Cost(q_j)} \quad (2)$$

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 2 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 re-inventing the wheel, we exploit the cost model established in [23] 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 3, 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 5 (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 3. 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 [8]. 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 1).

Algorithm 1 *SharedGrouping()* Algorithm

Input: A set of query groups G
Output: A shared grouping solution sol

```
1: for Gi:G do
2:    if Gi.bVal < 0 then
3:       G.remove(Gi) // remove groups with negative benefits
4:    end if
5: end for
6: sort(G) // sort groups by Gi.bVal
7: sol ← φ
8: for Gi:G do
9:    if !isOverlapping(Gi, sol) then
10:      sol.add(Gi) // add query group Gi to the solution
11:   end if
12: end for
13: return sol
```

The time complexity of Algorithm 1 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 1 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 Section 5.3, our experimental results illustrate that this strategy effectively helps the B&B algorithm find the optimized shared execution plan for given recurring workloads.

4.1.2 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 4) produced by Algorithm 1, 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 4). If the global execution ordering EO based on the given shared groups is found to be sub-optimal (described in Section 4.2.1), 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 [9]. 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.

4.2 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.

4.2.1 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 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.*

EXAMPLE 7. *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 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.*

EXAMPLE 8. *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.*

Due to the space limitation, the detailed proof of Lemmas 1 and 2 can be found in our technical report [12].

4.2.2 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 2, 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 4.1. 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 4.2.1.

Algorithm 2 Helix Optimizer Algorithm

Input: Query Set Q, Heap $heap$, Node $root$, Node tmp
Output: Node $solution$

```
1: solution ← SharedGrouping(Q)  // call sharing algorithm
2: solution.score ← order(solution)  // call ordering algorithm
3: Push(heap, root)
4: while !isEmpty(heap) do
5:    tmp ← Pop(heap)
6:    tmp.score ← order(tmp)
7:    if tmp.score > solution.score then
8:       solution ← tmp
9:       Node[] children ← childrenOf(tmp)
10:      for Node c ∈ children do
11:         c.score ← order(c)
12:         Push(heap, c)
13:      end for
14:   else if !groupPrune(tmp.score) then
15:      Node[] children ← childrenOf(tmp)
16:      for Node c ∈ children do
17:         c.score ← order(c)
18:         Push(heap, c)
19:      end for
20:   end if
21: end while
22: return solution
```

Figure 4 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 4, 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.

5. EXPERIMENTAL EVALUATION

In this section, 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.

5.1 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 [1] (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 [2] (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 Section 1. 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 Section 2.1. 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 1 summarizes the utility functions used in this study, which are also commonly used to specify SLA requirements [17]. 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 [13] 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 [13] is the state-or-the-art approach for evaluating a recurring query in MapReduce. Two, GGTMT [23] 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).

#	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	$\theta_{F3}(t) = \begin{cases} 1 & for \quad t \leq t_d \\ 1/(t - t_d) & for \quad t > t_d \end{cases}$

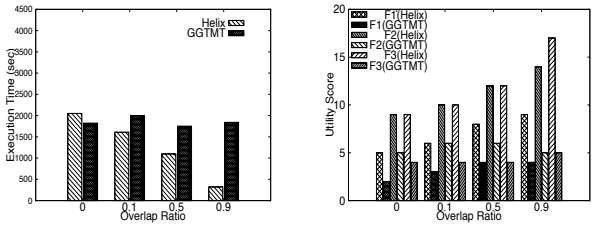
Table 1: Utility Functions Used in the Experimental Study

5.2 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.

5.2.1 Effectiveness of Sliced Window Alignment

Figure 5 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)

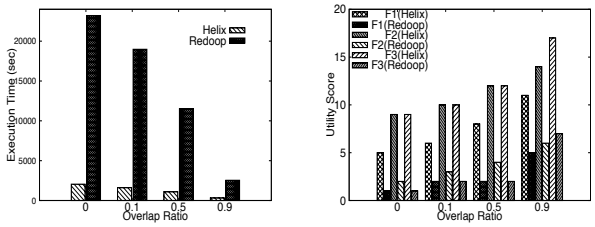
Figure 5: Effectiveness of Sliced Window Alignment

Helix's sliced window alignment technique substantially reduces the execution time while also increasing the SLA satisfaction as illustrated in Figure 5(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 5(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. Due to the space limit, the join operation's results are given in [12].

5.2.2 *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 maganitute 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.



(a) Execution Time (Aggregate) (b) Utility Score (Aggregate)

Figure 6: Significance of Sharing Benefits

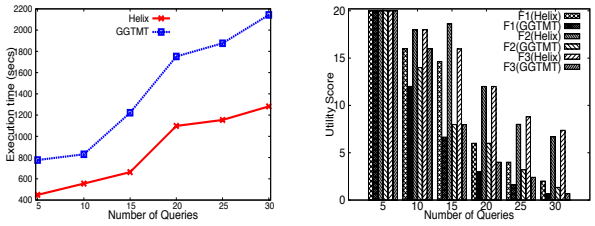
The results in Figure 6(b) illustrate the improvement of Helix over Redoop. In all cases, even when $overlap$ is 0, our Helix system achieves much better SLA satisfaction compared to Redoop. The reason is that Helix's shared execution plan is optimized for maximizing the SLA satisfaction. On the contrary, 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 our Helix solution over Redoop in both metrics. These results can be found in [12].

5.2.3 *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 7, 8, and 9 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). Due to space constraint, the results of the join operation are given in [12].

Effect of number of queries. Figure 7 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.

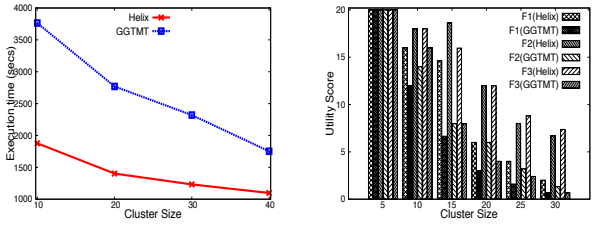
Regarding the utility score, the advantage of Helix is more obvious compared to GGTMT. In Figure 7(b), The total utility score achieved by Helix is up to 1089% more than the one achieved by GGTMT. The winning margin is similar to the observation in Figures 7(a). The reason is that Helix optimization 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.



(a) Execution Time (Aggregate) (b) Utility Score (Aggregate)

Figure 7: Varying Number of Queries

Effect of cluster size. Figure 8 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.



(a) Execution Time (Aggregate) (b) Utility Score (Aggregate)

Figure 8: Varying Number of Nodes

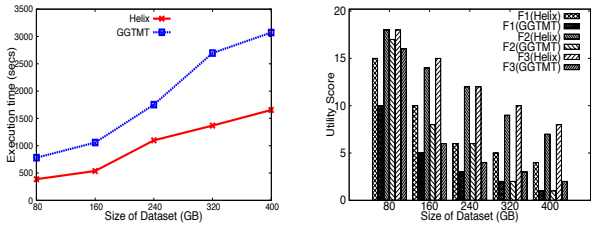
Furthermore, as the cluster size increases, the running time for both approaches decreases. In particular, the running time of Helix decreases much faster than the one of GGTMT solution which therefore enlarges the winning margin as cluster size increases. Thus, the performance improvement from the increased parallelism

using a larger cluster benefits Helix more than the GGTMT technique. The reason is that Helix system's sliced window alignment technique provides appropriate sized inputs. That enables Helix to make full use of the parallelism with more nodes are available.

In Figure 8(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 9(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 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 9(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)

Figure 9: Varying Size of Dataset

5.3 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.

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.

6. RELATED WORK

Recurring Query Processing. Recurring query processing systems [4, 5, 13] have been proposed to support large-scale data analytics applications over evolving data streams. SCOPE [4, 5] 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 [13] 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 [6, 18, 20]. Techniques have been proposed for tackling MQO problems in relational database systems, in particular, materialized views [6, 18] and common sub-expression sharing [20]. In [18], materialized views are effective techniques for implementing common sharable processing across queries. Chaudhuri et al. [6] proposed to pre-compute views over static data to be used by other subsequent queries. In the Cache-on-Demand (CoD) system [20], 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 MapReduce: Several techniques have been proposed for sharing or reusing work across multiple queries on MapReduce. MRShare [16] 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 [7] 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. 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. [23] propose two sharing techniques, the *generalized grouping technique (GGT)* and the *materialization technique*

(MT), to refine multi-query optimization in MapReduce. They also design a cost-based two-phase approach to find shared execution plans. Compared with [23], 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 [11, 24]. Prior work [11] has leveraged specific workload characteristics to meet SLAs without losing efficiency or utilization. To provide real-time responses, [11] 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 [24] 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 [3, 22, 25] 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.

7. CONCLUSION

This paper presented the first targeted 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 sub-optimal 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.

8. REFERENCES

[1] 1998 world cup. `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`.

[2] Wikipedia database. `http://dumps.wikimedia.org/enwiki/latest/`.

[3] G. Ananthanarayanan, C. Douglas, et al. True elasticity in multi-tenant data-intensive compute clusters. In *SoCC*, pages 24:1–24:7, 2012.

[4] N. Bruno, S. Agarwal, et al. Recurring job optimization in scope. In *SIGMOD*, pages 805–806, 2012.

[5] N. Bruno, S. Jain, and J. Zhou. Recurring job optimization for massively distributed query processing. *IEEE Data Eng. Bull.*, 36(1):46–55, 2013.

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

[7] I. Elghandour and A. Aboulnaga. Restore: reusing results of mapreduce jobs. *Proc. VLDB Endow.*, 5(6):586–597, 2012.

[8] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *PVLDB*, 7(6):429–440, 2014.

[9] A. J. Kleywegt and J. D. Papastavrou. The dynamic and stochastic knapsack problem. *Operations Research*, 46:17–35, 1998.

[10] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[11] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterprises. In *BIRTE*, pages 143–156, 2007.

[12] C. Lei, E. Rundensteiner, and M. Eltabakh. Technical report: Shared execution of recurring workloads. `http://users.wpi.edu/~chuanlei/TR1114.pdf`.

[13] C. Lei, E. A. Rundensteiner, and M. Eltabakh. Redoop: Supporting recurring queries in hadoop. In *EDBT*, pages 817–828, 2014.

[14] J. Li, D. Maier, K. Tufte, et al. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, pages 39–44, 2005.

[15] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of mapreduce pipelines. In *ICDE*, pages 681–684, 2010.

[16] T. Nykiel, M. Potamias, et al. Mrshare: sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, pages 494–505, 2010.

[17] V. Raghavan and E. A. Rundensteiner. Caqe: A contract driven approach to processing concurrent decision support queries. In *EDBT*, pages 121–132, 2014.

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

[19] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In *SIGMOD*, pages 1125–1134, 2013.

[20] K.-L. Tan, S.-T. Goh, and B. C. Ooi. Cache-on-demand: Recycling with certainty. In *ICDE*, pages 633–640, 2001.

[21] A. Thusoo, Z. Shao, et al. Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, pages 1013–1020, 2010.

[22] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC*, pages 235–244, 2011.

[23] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. *PVLDB*, 7(3):145–156, 2013.

[24] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010.

[25] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *ICAC*, pages 53–62, 2012.