Redoop: Supporting Recurring Queries in Hadoop[*]

Chuan Lei, Elke A. Rundensteiner, and Mohamed Y. Eltabakh
Worcester Polytechnic Institute, Worcester, MA 01609, USA
{chuanlei|rundenst|meltabakh}@cs.wpi.edu

ABSTRACT

The growing demand for large-scale data analytics ranging from online advertisement placement, log processing, to fraud detection, has led to the design of highly scalable data-intensive computing infrastructures such as the Hadoop platform. 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. Despite their importance, the plain Hadoop along with its state-of-art extensions lack built-in support for recurring queries. In particular, they lack efficient and scalable analytics over evolving datasets. In this work, we present the *Redoop* system, an extension of the Hadoop framework, designed to fill in this void. Redoop supports recurring queries as first-class citizen in Hadoop without sacrificing any of its core features. More importantly, Redoop deploys 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. Our extensive experimental study with real datasets demonstrates that Redoop achieves significant runtime performance gains of up to 9x speedup compared to the plain Hadoop.

1. INTRODUCTION

Motivation. The proliferation of data and the availability of large-scale data processing systems, e.g., the MapReduce [14] and Hadoop platforms [28], have enabled most applications to explore their data using data-intensive analytical tasks that were not possible before. Hadoop is a widely-used platform for such data-intensive applications because of its scalability, flexibility, and fault tolerance. However, as proven by a flurry of research work on Hadoop [20, 23, 16], the scalability and distributed processing are not enough to achieve high performance. Instead, the system needs to be highly optimized for various query types. In this paper, we identify a type of queries, called *recurring queries*, very common in most Hadoop-based large-scale applications, yet not supported effectively by the state-of-the-art systems.

Recurring queries appear in numerous applications that periodically generate and collect huge volumes of fresh data that must be continuously integrated into the complex analysis, such as log processing [22], news feed updates [27], and social network services [27]. Thus, the same analytical queries are periodically 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. As we will highlight in the following motivating examples, recurring queries are challenging to support because the combined characteristics from both *real-time continuous queries* and *offline ad-hoc queries* create new optimization opportunities to explore.

Example 1. Log Processing. Scalable log processing is critical for running large-scale web sites and services. With the current log generation rates in the order of 1-10 MB/s per machine, a single data center may collect 10s of TBs of log data per day [22]. For example, Facebook analyzes 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 variety of data-intensive jobs on a large Hadoop cluster. An example recurring query that would 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. Such query involves expensive join and grouping operations over high-volume evolving data.

Example 2. News Feed Updates. On modern consumer websites, news feed generation is nowadays driven by online systems. Online news services may be generated on a per-member basis based on the member's interactions with other components in the system. For example, a LinkedIn member may receive periodic updates on their profile changes. Computing these updates involves deep analytical processing of large-scale data sets across multiple sources. For example, to generate an update highlighting the company in which most of a member's connections have worked in the past month requires joining the company's 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.

Example 3. Clickstream Analysis. 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

[*] This project is supported by NSF grants CNS-305258, IIS-1018443 and IIS-0917017

10.5441/002/edbt.2014.04

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.

Many other data analytics applications share similar characteristics with the above mentioned examples: they need to periodically run recurring queries over large volumes of data. Even with a relatively small processing window, the amount of overlapping data between consecutive executions can be huge. Therefore, without proper system-level support, e.g., understanding the recurring nature of queries, critical optimization opportunities will be missed.

Spectrum of Recurring Queries. Recurring queries present unique challenges to existing systems because they inherit properties from both continuous queries (in stream processing systems) and ad-hoc queries (in batch processing systems).

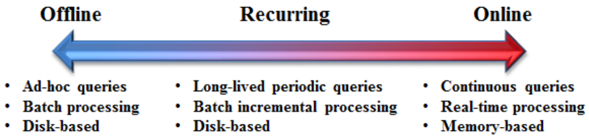


Figure 1: Large-Scale Data Processing Spectrum

As shown in Figure 1, 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. However, they fundamentally differ in that recurring queries do not mandate real-time millisecond processing. Instead, they tend to have a large granularity of execution, e.g., they may execute once every hour or every day, and may return the results within a certain period of time, e.g., a few minutes to a couple of hours. Hence the systems may remain idle for long periods of time. Lastly, recurring queries are inherently data-intensive disk-based queries that may process TBs of disk-resident data in each execution. In contrast, stream processing systems [4, 3, 8, 19] target main-memory real-time processing in contrast to disk-based processing. Hence, they will not scale to the huge volume of data being processed by recurring queries. They will waste significant system resources by maintaining the intermediate data in memory at all times even when the queries are intermittently inactive.

On the other side of the spectrum, batch-processing systems, e.g., Hadoop [28], are well-designed for scalability and disk-based processing, which are common properties for both recurring and ad-hoc batch queries. However, these systems [28, 5] lack the notion of sliding windows. Hence they fall short in providing efficient support for incremental processing over overlapping data sets. That is, most applications using Hadoop tend to run their recurring queries by issuing a separate query (job) for each re-execution. Thus the system cannot leverage possible optimization opportunities. This is clearly an inefficient solution especially under the evolving infrastructure-as-a-service (IaaS) models [9], where end-users are eager to optimize their jobs to process their data while consuming less resources, yet finish faster.

Insufficiency of State-of-the-Art. Recently, few extensions to Hadoop, e.g., Nova [24] and Apache Oozie [7], have been proposed to address the requirements of the above applications. However, they remain far from providing an end-to-end optimized system for supporting recurring workloads. Apache Oozie [7] is a workflow scheduler that provides partial support by enabling developers to write scripts for automatic scheduling of jobs. Hence, end-users would not need to re-issue the recurring query over and over. However, Apache Oozie does not provide any system optimizations since it lacks the notions of sliding windows and incremental processing. The Nova [24] 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, e.g., offering neither caching of intermediate data for reuse, cache-aware task scheduling, nor adaptive processing based on input data rates.

Contributions. We propose a new system called *Redoop* that is designed to support recurring queries. Redoop extends the Hadoop platform to be amenable to optimization opportunities from recurring queries. This paper makes the following contributions:

1. The Recurring Query Model: A recurring query model is established 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. Adaptive Window-Aware Input Data Packing: 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. Window-Aware Caching and Maintenance: 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. Window-Aware Task Scheduling: 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 evaluated 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.

The paper is organized as follows: Section 2 introduces the Redoop architecture. Sections 3 and 4 describe our proposed strategies for adaptive input data handling and window-aware caching, respectively. Section 5 provides system design details for the Redoop system. Sections 6 and 7 discuss experimental results and related work. Section 8 concludes the paper.

2. PRELIMINARIES AND SYSTEM OVERVIEW

2.1 Recurring Query Model

Parameters. Recurring queries execute periodically over evolving disk-resident datasets, i.e., datasets stored in HDFS. In each execution, a query bounds its computations to a time-based window over the datasets. Therefore, a recurring query is specified by two configuration parameters, win and $slide$. The window win specifies the scope of data to process while the slide $slide$ specifies the frequency of execution. For example, a recurring query with *win = 12 hours* and *slide = 1 hour* specifies a query that executes each hour and processes the available data within the last 12 hours.

Timestamps. Between two consecutive executions E_i and E_j at times T_i and 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 $T_i < T_{f1} < T_{f2} < ... < T_{fn} < T_j$. The data records within each file will have their own timestamps. We assume that time ranges covered by the batch files do not overlap and are in order. That is, the time ranges covered by the tuples in files f_1, f_2, ..., f_n, are in the range of $[T_i, T_{f1})$, $[T_{f1}, T_{f2})$, ... , $[T_{fn-1}, T_{fn})$. Thus, there is an order among the files, but there is no order constraint among the tuples within each file. 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.

Execution. Redoop optimizes the consecutive executions of recurring queries through disk-based caching and incremental processing. Even ad-hoc queries can benefit from the caching of the intermediate data to avoid re-processing and re-shuffling of overlapping data segments across adjacent windows. Redoop aims to finish the execution of a recurring query before the next execution. Therefore, if Redoop observes a peak in data arrival rates, then it automatically applies a *proactive best effort* mechanism for early processing the batch files received so far. Our Redoop system leverages the statistics collected from the previous executions to decide when to switch to this proactive mode. The partial results generated from the proactive execution are merged together before reporting to end-users. Redoop is designed as a general-purpose system for recurring queries. It will provide the system-level optimizations based on the window semantics independent of the logic of the queries, which are defined by the end-users in the map and reduce functions. In Section 5, we introduce several extensions to the map-reduce model to enable users to provide their own window specifications and finalization function that merge multiple partial outputs and generates a final output from each execution.

2.2 Background on MapReduce Paradigm

MapReduce is a distributed programming paradigm proposed by Google for large-scale data processing in distributed environments [14]. Given a list of <key, value> pairs as the input for a map-reduce job, each map function produces zero or more intermediate <key, value> pairs by consuming one input tuple at a time. Then, the run-time system groups the intermediate <key, value> pairs based on their keys into buckets to be processed by the reduce tasks. Each reduce task consumes one group of <key, list-of-values> at a time and produces zero or more output tuples. Next, we briefly highlight several important components of Hadoop as context for the architectural design changes propose in Redoop.

Input and Output Data. Hadoop utilizes the distributed fault tolerant file system HDFS spread across the local disks of the computation nodes. Hadoop reads input data from and stores output results to HDFS. HDFS has built-in data replication strategies to recover from failures and balance its workload.

Handling Intermediate Data. The intermediate data is shuffled and sorted after the map stage. The intermediate data is then stored in the local disks of the mapper nodes where they are produced. All output tuples sharing the same key are assigned to the same reducer. Reducers will then retrieve their inputs from multiple mappers and start executing on their assigned groups.

Task Scheduling. A centralized component in Hadoop called *Job Tracker* is responsible for dividing a job into small tasks and assigning each task to a compute node. The Job Tracker is communicating with a local *Task Tracker* on each compute node to monitor the overall execution of a job. Hadoop comes with different scheduling policies, with the *FIFO* scheduler as the default.

Fault Tolerance. Handling failures is one of the key considerations in the MapReduce architecture. In Hadoop, the distributed file system handles the failure of disks or nodes using data replication. A failure of a map task requires the failed task re-execution while a failure of a reduce task entails retrieving the corresponding map outputs again and re-executing the reduce task.

2.3 The Proposed Redoop System

Figure 2 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. 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 2.

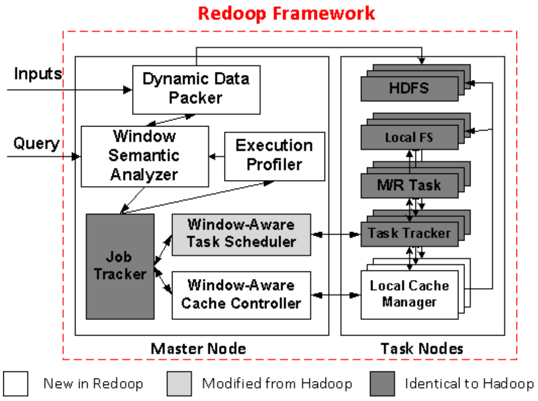


Figure 2: Redoop System Architecture

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 (Section 3.1).

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 (Section 3.2). 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 the Redoop's execution modes to tackle data fluctuations (Section 3.3).

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 (Section 4.1).

5. Window-Aware Cache Controller is a new module housed on the Redoop master node that maintains window-aware metadata 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 (Section 4.2).

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 (Section 4.3).

3. 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.

3.1 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 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 Section 4 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 1 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.

Algorithm 1 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: end if
9: return $PP$
```

Having such optimization on mapping logical data units to physical files, the Dynamic Data Packer can avoid creating many small files in Hadoop. The logical pane size is 2 minutes as a result of GCD (6, 2), namely win = 6 minutes and $slide$ = 2 minutes. 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 3, with each file denoted by the same color.

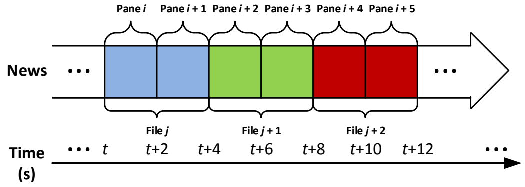


Figure 3: A Partition Example

3.2 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 1:

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.

3.3 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 [20]. 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:

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

$$T_i = \gamma \cdot (L_i - L_{i-1}) + (1 - \gamma) \cdot T_{i-1} \quad (2)$$

The smoothing parameters α and γ can be selected by fitting historical data (for details please refer to [12]).

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

$$\hat{X}_{t+k} = L_t + k \cdot T_t \quad (3)$$

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.

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.

4. WINDOW-AWARE CACHING

To reduce the unnecessarily 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.

4.1 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 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.

pid	type	expiration
S1P3	1	1
S2P4	2	0

Table 1: Examples of a Local Cache Registry

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.

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.

4.2 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 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.

pid	nid	type	ready	doneQueryMask
S1P3	1	1	2	10011
S1P3	1	2	2	10011
...	...	...	...	...
S1P7	1	1	2	10011

Table 2: Example of Window-aware Cache Controller

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.

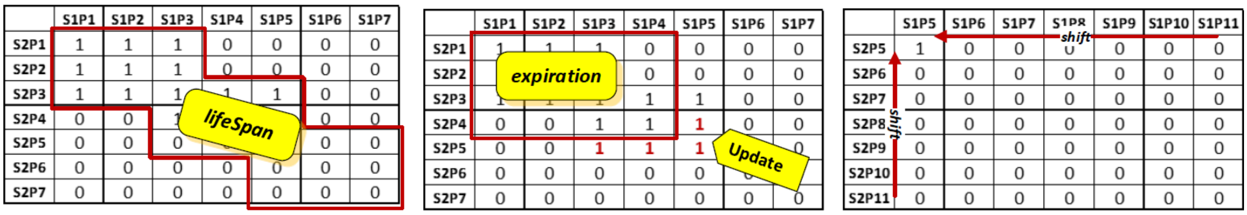
	S1P0	S1P1	S1P2	S1P3
S2P0	1	1	0	0
S2P1	1	1	0	0
S2P2	0	0	0	0
S2P3	0	0	0	0

Table 3: Example of Cache Status Matrix

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



	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 4: Operations on Cache Status Matrix

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 4.1, 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 4, 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.

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 4(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 4(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 4(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 4(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 4(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.

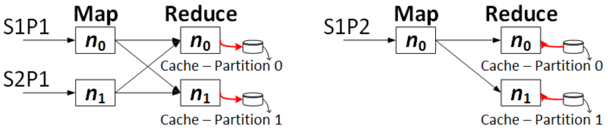
4.3 Cache-Aware Task Scheduling

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 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.

The schedule in Figure 5 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).



(a) Window 1 (b) Window 2

Figure 5: Cache-Aware Task Scheduling

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:

$$node = arg\ \min_{i \in N} [Load_i + C_{task,i}] \quad (4)$$

where $Load_i$ indicates the current load on the ith 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 [20] 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 4 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 2 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 4 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 2 Cache-Aware Task Scheduling Algorithm

```
Input: Node $node$, Map$\langle Node, List\ \langle Partition \rangle\rangle$ $cache$
1: boolean $start$ = true
2: if $start$ = true then
3:     Partition $p$ = defaultSchedule($node$);
4:     $cache$.get($node$).add($p$);
5:     $start$ = false;
6: else if !$mapTaskList$.isEmpty() then
7:     Partition $p$ = $mapTaskList$.get(0);
8:     $node$ = selectNode(M, p);   // select a node for a map task
9:     schedule($p$, $node$);
10:    $cache$.get($node$).add($p$);
11:    $mapTaskList$.remove(0);
12:    updateCache($p$);     // update cache associated information
13: else if !$reduceTaskList$.isEmpty() then
14:    Partition $p$ = $reduceTaskList$.get(0);
15:    $node$ = selectNode(R, p);   // select a node for a reduce task
16:    schedule($p$, $node$);
17:    $reduceTaskList$.remove($index$);
18:    updateCache($p$);     // update cache associated information
19: end if
```

5. REDOOP IMPLEMENTATION

This section presents the Redoop implementation details.

Window Controller and API. As Figure 2 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 Section 2.1) 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 3.2). 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.

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 reconstruct this cache as in Hadoop failure recovery is completely transparent to user programs.

6. 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.

6.1 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) [1] 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 [2]. 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.

6.2 Effect of Pane-based Caching

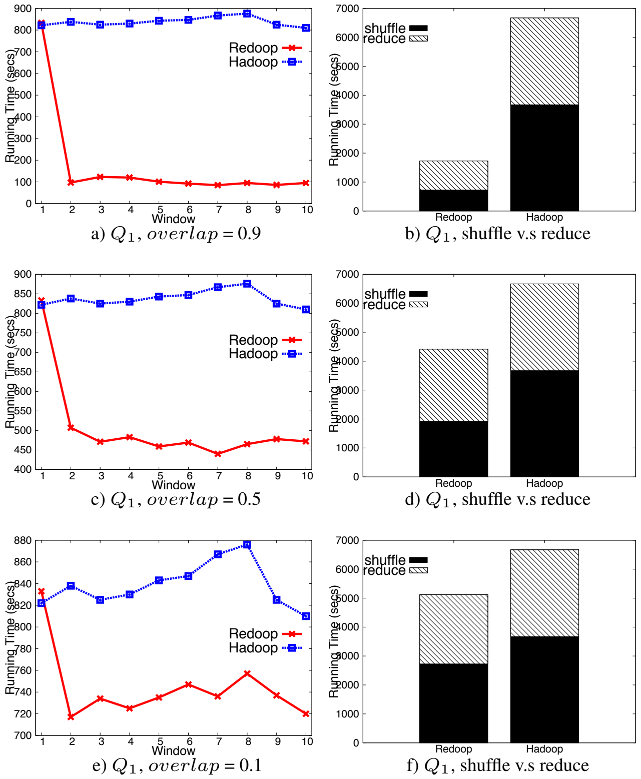
Our incremental processing experiments evaluate the Redoop system's performance by varying the factor $overlap = \frac{win-slide}{win}$, a ratio between the slide and window size. This ratio represents the portion of the newly arriving data tuples in the window after the window slides. Thus, the higher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

6.2.1 Aggregation Query Evaluation

We run an aggregation query over the WCC dataset that ranks the movements of players. Figure 6 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 6), Redoop significantly outperforms Hadoop. Redoop's task scheduling and pane-based caching substantially reduce aggregation time. The running time depicted in Figures 6(a), (c), and (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 6(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 improve-

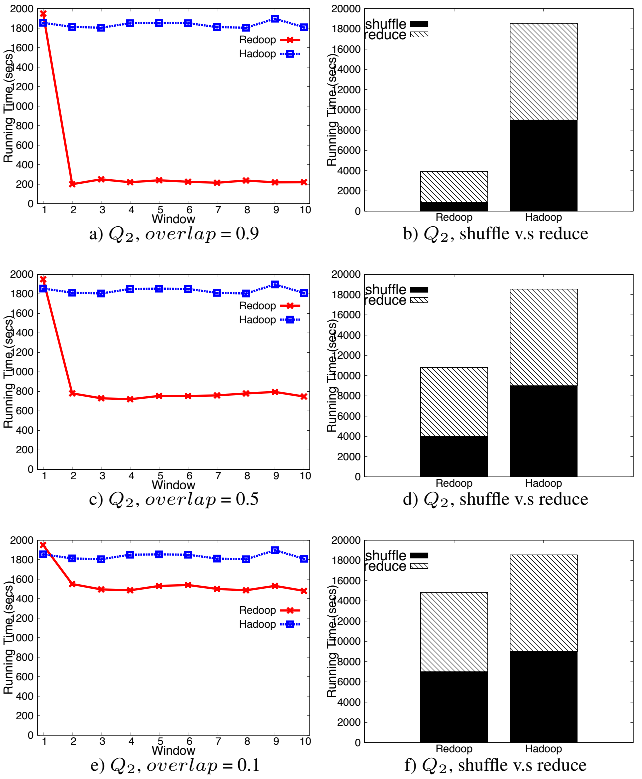


a) Q_1, $overlap = 0.9$ b) Q_1, shuffle v.s reduce

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 6: Aggregation Query Performance



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: Join Query Performance

ments to each processing phase, we also compared the cost distribution of the aggregation across the Shuffle and Reduce phases. Figures 6(b), (d), and (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 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 6(b). By contrast, for Figure 6(f), the cache does not help much, because the *overlap* (10%) is low. The results in Figure 6 clearly demonstrate the effectiveness of our incremental design gained by using pane-based caching.

6.2.2 Join Query Evaluation

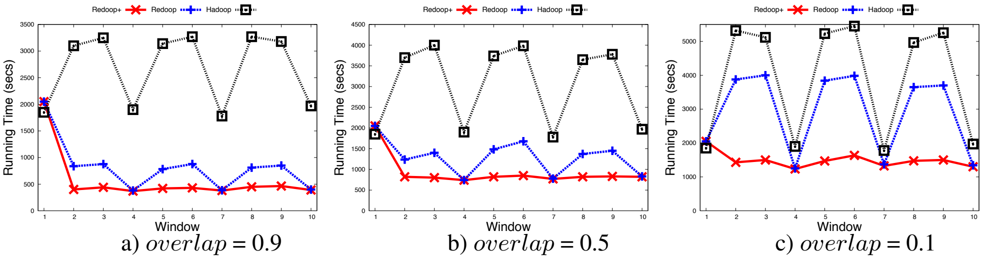
We run a join query on the FFG dataset with the same system settings as above. Overall, as Figure 7 shows, for a join job, Redoop achieves better performance by winning almost an order of magnitude in the best case scenario with the cache enabled, as described below.

Overall response time. Redoop's performance shows a similar pattern with the join query on the FFG dataset. The running times in Figures 7(a), (c), and (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 6a achieves 9 fold performance improvement by taking advantage 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(b), (d), and (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(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.

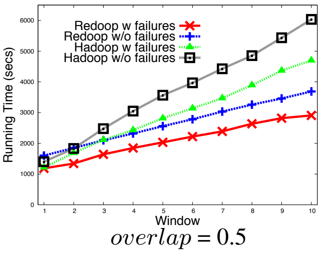
6.3 Effect of Adaptive Input Partitioning

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 pe-



a) $overlap = 0.9$ b) $overlap = 0.5$ c) $overlap = 0.1$

Figure 8: Adaptive Input Partitioning



$overlap = 0.5$

Figure 9: Fault Tolerance

riodically. Specifically, windows 1, 4, 7, and 10 have the normal workloads. The workloads of the rest of the windows are doubled. Figure 8 shows the processing time for 10 windows with three different settings ($overlap$ = 90%, 50%, and 10%).

For very small overlaps as shown in Figure 8(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 starting 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 3.3.

However, as the overlaps grow, we observe an interesting behavior by Redoop without adaption. In Figure 8(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.

6.4 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 9, 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 9, 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 6(a) 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.

7. RELATED WORK

Hadoop Extension. MapReduce [14] and its open-source implementation Hadoop [28] have emerged as a popular model for large-scale distributed data processing in shared-nothing clusters. Hence, several extensions have been proposed to improve Hadoop's performance w.r.t different query types, e.g., SQL-like queries [18, 25], online processing [13], and iterative queries [11]. However, none of these systems support or optimize the data-intensive recurring queries addressed in this paper.

While some studies such as HaLoop [11] and Twister [15] 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 [11] 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 [21, 29, 17] to provide an end-to-end optimization for supporting recurring queries. Similar to HaLoop, Twister [15] 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.

Nova [24] is the most 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 $M3R$ [26], $SOPA$ [20], C-MR [10], and In-situ (iMR) [22] 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 [20] 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 [6] builds a main-memory implementation of MapReduce and places constraints on the type of supported jobs imposed by available memory size. C-MR [10] 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) [22] 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 [4, 3, 8, 19]. However, as we highlighted in Section 1, 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.

8. CONCLUSION

In summary, this work presents 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.

9. ACKNOWLEDGEMENTS

The authors thank Lei Cao, Qingyang Wang, Medhabi Ray, and other DSRG members for helpful discussions.

10. REFERENCES

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

[2] Soccer - real time tracking system. `http://www.iis.fraunhofer.de/en/bf/ln/referenzprojekte/redfir.html`.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, et al. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[4] D. J. Abadi, D. Carney, U. Çetintemel, et al. Aurora: A data stream management system. In *SIGMOD*, page 666, 2003.

[5] A. Abouzied, K. Bajda-Pawlikowski, et al. Hadoopdb in action: building real world applications. In *SIGMOD*, pages 1111–1114, 2010.

[6] A. M. Aly, A. Sallam, B. M. Gnanasekaran, et al. M3: Stream processing on main-memory mapreduce. In *ICDE*, pages 1253–1256, 2012.

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

[8] A. Arasu, B. Babcock, S. Babu, M. Datar, et al. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[9] M. Armbrust, A. Fox, et al. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[10] N. Backman, K. Pattabiraman, R. 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.

[11] Y. Bu, B. Howe, M. Balazinska, and others. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[12] C. Chatfield. The holt-winters forecasting procedure. *Applied Statistics*, pages 264–279, 1978.

[13] T. Condie, N. Conway, P. Alvaro, et al. Mapreduce online. In *NSDI*, pages 313–328, 2010.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[15] J. Ekanayake, H. Li, B. Zhang, et al. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, 2010.

[16] M. Y. Eltabakh, Y. Tian, F. Özcan, et al. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, pages 575–585, 2011.

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

[18] *Hive*. `http://hadoop.apache.org/hive`.

[19] C. Lei, E. A. Rundensteiner, and J. D. Guttman. Robust distributed stream processing. In *ICDE*, pages 817–828, 2013.

[20] B. Li, E. Mazur, Y. Diao, et al. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, pages 985–996, 2011.

[21] 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.

[22] D. Logothetis, C. Trezzo, K. C. Webb, et al. In-situ mapreduce for log processing. In *USENIXATC*, pages 9–9, 2011.

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

[24] C. Olston, G. Chiou, L. Chitnis, et al. Nova: continuous pig/hadoop workflows. In *SIGMOD*, pages 1081–1090, 2011.

[25] *Pig*. `http://hadoop.apache.org/pig`.

[26] A. Shinnar, D. Cunningham, B. Herta, et al. M3r: Increased performance for in-memory hadoop jobs. *PVLDB*, pages 1736–1747, 2012.

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

[28] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org`.

[29] S. Wang, E. Rundensteiner, S. Ganguly, et al. State-slice: new paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006.