Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis

Pramod Bhatotia[†] Marcel Dischinger[†1] Rodrigo Rodrigues[‡] Umut A. Acar[§]

[†]MPI-SWS, [‡]CITI/Universidade Nova de Lisboa, [§]CMU

Technical Report: MPI-SWS-2012-004

September 2012

ABSTRACT

Sliding-window computations are widely used for data analysis in networked systems. Such computations can consume significant computational resources, particularly in live systems, where new data arrives continuously. This is because they typically require a complete re-computation over the full window of data every time the window slides. Therefore, sliding-window computations face important scalability problems. In this paper, we propose techniques for improving the scalability by performing sliding-window computations incrementally. In this paradigm, when some new data is added at the end of the window or old data dropped from its beginning, the output is updated efficiently by reusing previously run sub-computations, avoiding a complete re-computation. To realize this approach, we propose Slider, a novel framework that supports incremental sliding-window computations transparently and efficiently by leveraging self-adjusting computations principles of dynamic dependence graphs and change propagation. We implemented Slider based on the Hadoop MapReduce framework with a declarative SQL-like query interface, and evaluated it with a variety of applications and real world case studies from networked system. Our results show significant performance improvements for large-scale sliding-window computations without any modifications to the existing application code.

1. INTRODUCTION

The complexity of modern networked systems makes data analysis a fundamental component to understand and monitor their behavior, e.g., to measure system performance [21], perform capacity planning [24], detect network anomalies [20], diagnose problems [14], provide accountability [16], or detect security threats [32]. Driven by the criticality and usability of these applications, the networked systems aggressively generate a plethora of detailed raw data, which is collected on-the-fly or in small batches for later analysis.

Since networked systems normally run continuously, they generate massive amounts of data that grows monotonically over time. As a result, one of the most common workflows for data analysis is to continuously compute over incrementally growing data set. In addition to combining newly appended data, many applications also consider the case of discarding older data when it is less relevant compared to the recent data. In other words, data analyses often compute over a *sliding window* of data.

The basic approach to sliding-window data processing is to recompute the application-specific computation whenever the window slides. Consequently, the arrival of new data items cause a full re-computation, requiring re-processing of the old, unchanged data items that remain in the window. This complete re-computation approach has two major drawbacks. First, it wastes resources unnecessarily by processing unchanged data items. Second, it fundamentally limits the timeliness of results by performing this unnecessary computation. Timeliness is important in time-sensitive applications that rely on up-to-date information even under frequent updates, e.g., applications detecting security threats by scanning network streams for virus signatures [29].

One way to overcome the limitations of the basic approach is by using incremental update mechanisms, where the outputs are updated to accommodate the arrival of new data instead of recomputed from scratch. Such an incremental approach can be significantly—often asymptotically—more efficient than the basic approach, because in the common case the size of the window is large relative to increment by which the window slides. One way to support such incremental updates is to rely on the users (programmers, network administrators) to devise an incremental update mechanism. Unfortunately, this approach turns out to be difficult because it requires designing and implementing an *incremental algorithm* (or *dynamic algorithm*) containing the logic for incrementally updating the output. Research in the algorithms community (e.g., for surveys [9, 27, 13]) shows that such dynamic algorithms can be quite difficult to design even for problems that are simple in the static (non-incremental) case where the data does

[1]Currently at Barracuda Networks in Innsbruck, Austria.

not change. Most of these algorithms also assumes a uniprocessor computing model, as a result, they are not easily parallelizable for parallel and distributed systems (such as MapReduce, Spark, Dryad) typically employed to process large-scale data.

In this paper we propose Slider, a novel framework for incremental sliding-window computation for scalable data analysis in networked systems. In Slider, the programmer expresses the computation to be performed using MapReduce programming model by assuming a static, unchanging input window, and we guarantee automatic and efficient updates as the window slides, e.g., due to the arrival of new data. We achieve efficiency by automatically incrementalizing the computation and require no changes to the user code: apart from specifying the computation to be computed on static unchanging data, the programmer need not design application-specific incremental update mechanism.

Our approach to automatic incrementalization is based on the principles of self-adjusting computation (e.g., [1, 2]). The idea behind self-adjusting computation is to represent (reify) a computation with a dynamic data-dependency graph that contains the input data of a computation job, all relevant sub-computations, and the data and control flow between them. When the data changes, a *change-propagation algorithm* pushes the change through the dynamic dependency graph by re-executing all sub-computations that are transitively affected by the change, reusing the sub-computations that remain unaffected, and reconstructing an updated dynamic dependency graph that can be used by further updates. The efficiency of change propagation is determined by the stability of the dynamic dependency graph under changes—the less stable the graphs, the more they change and more time change propagation requires. We design algorithms and techniques that ensure that the dynamic dependency graph remains stable for different variants of sliding-window computations.

We further enhance the self-adjusting-computation approach by designing a *split processing model* that takes advantage of the predictability of incoming input data streams to minimize response times. To achieve this, Slider splits the application processing into two parts: a foreground and background processing. The foreground processing takes place right after the update to the computation window, and minimizes the processing time by combining new data with pre-computed intermediate result. The background processing takes place after the result is produced and returned, paving the way for an efficient foreground processing by pre-computing the intermediate result that will be used in the next incremental update.

We built a prototype of Slider by extending the Hadoop MapReduce framework. Slider allows for analyses using either the conventional MapReduce model or using an interface for declarative query execution (like SQL). We evaluated the effectiveness of Slider by applying it to a variety of microbenchmarks and applications. We also present three real world use cases of using Slider to accelerate sliding-window computations in networked systems: building an information-propagation tree [28] for the Twitter Online Social Network (OSN), monitoring Glasnost [14] measurement servers for detecting traffic differentiation, and providing peer accountability in Akamai NetSession [4], a hybrid Content Distribution Network (CDN). Our experiments show that Slider can deliver significant performance gains, while incurring only modest overheads for the initial run.

The remainder of the paper is organized as follows. We first present related work for incremental sliding-window computation in networked systems (Section 8). Next, we describe the design goals and challenges along with overview of Slider (Section 2). We then cover the design details and implementation (Sections 3 and 4). Thereafter, we evaluate Slider experimentally (Section 5) and apply it to three real-world case studies (Section 6). We cover the declarative query interface for Slider in Section 7. Finally, the analytical performance evaluation is presented as part of the appendix A.

2. OVERVIEW

In this section, we present the motivation, design goals, and an overview of Slider.

2.1 Data Analysis in Networked Systems

Sliding-window computations are used ubiquitously in the analysis of networked systems; we discuss below some representative examples. Traditionally, these sliding-window computations re-compute results when the data window changes, e.g., when new data arrives, recomputing even old results that depend on unchanged data.

One area where sliding-window computations are widely employed is Internet measurements, where packet-, flow-, or graph-level analyses are used to compute low-level network flow characteristics, such as latencies, bandwidth, and loss rates. These network-wide properties are used by many applications, e.g., network coordinate systems [11] or load balancing systems.

Another area that employs sliding-window computations to scan passing network traffic is network security. For example, automated worm fingerprinting analysis [29] examine a packet stream in a sliding window to compute worm signatures. These signatures are continually updated by determining the top-K substrings as the computation window slides over the packet stream. Another example is the detection of stepping stone attacks [32], which involves identifying idle-to-active transitions by leveraging the correlation of multiple flows.

A third area is mapping large-scale systems, e.g., measurements to map the topology of the Internet [30] or crawls of online social networks [28] to extract relationship graphs or to track information flows. When mapping such a system, the input data is collected over a long period of time and potentially using multiple different measurement methods. This can be seen as an append-only, sliding-window computation, where the window grows monotonically as new data arrives.

Finally, traffic monitoring tools for anamoly detection [20] or learning communication rules in the edge networks [19] also use sliding-window computations. These computations involve mechanisms for learning the evolution of underlying link-level network traffic using techniques like association-rule mining, or Principal Components Analysis (PCA).

2.2 Design Goals

We aim to fulfill the following set of objectives.

Transparency. The developer of data-analysis jobs should not have to invest additional effort to use the framework, even when compared to a non-incremental analysis. More concretely, the developer does not have to design and implement an efficient dynamic algorithm for performing incremental updates. Existing data processing jobs should work without any changes.

Efficiency. The amount of work that is performed by the incremental framework should be substantially less than the non-incremental approach, i.e., recomputing when data changes. In particular, the framework should be able to achieve asymptotic performance gains.

Scalability. The framework should be capable enough to support large-scale data analysis in clusters with a large pool of compute nodes.

Expressiveness. The framework should enable expressing a broad range of computations on data and a should not constraint how the window moves. It should be possible to slide the window in either direction, and to grow, and shrink the window at both ends.

Flexibility. The framework should be flexible in making usage of low utilization periods in the compute cluster. It should opportunistically perform pre-processing at low-utilization periods so as to achieve low latency incremental updates.

2.3 Basic Design Overview

To achieve high expressiveness, fault-tolerance, scalability, and ease of use, we develop Slider based on the MapReduce model. Given its success, and in particular the widespread use of the Hadoop framework for it, we hope that choosing MapReduce will help us cover a broad set of existing applications.

We first outline a basic design for Slider, and then detail the limitations of the basic approach. For the presentation, we assume that the reader is familiar with

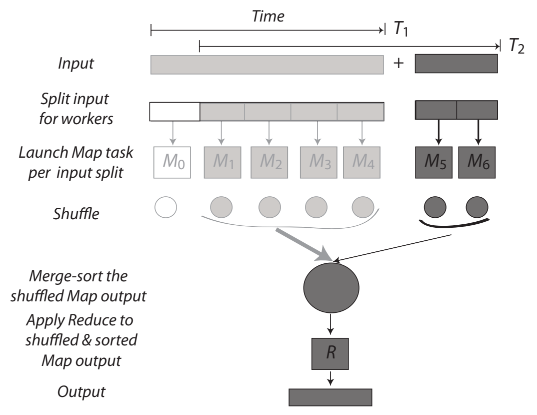

Time
T_1
T_2
Input
+
Split input for workers
Launch Map task per input split
M_0
M_1
M_2
M_3
M_4
M_5
M_6
Shuffle
Merge-sort the shuffled Map output
Apply Reduce to shuffled & sorted Map output
R
Output

Figure 1: Basic architecture of Slider

the MapReduce model [12]. Without loss of generality, we also assume for simplicity that each job includes a single Reduce task (i.e., all mappers output tuples with the same key). This assumption causes no loss of generality, because our techniques apply to multiple reducers (and keys) by symmetry.

Basic design. Figure 1 shows the basic design of Slider. Newly produced data items are always appended at the end of the previous window and some data items are dropped at the beginning. To update the output incrementally, Slider launches a Map task for each new "split" (a partition of the input that is handled by a single Map task) holding new data. Thus, Map tasks are launched only for the new data; for the remaining input, Slider reuses the results from Map tasks from the previous computation. This combination of reused and newly computed results from Map tasks is then fed to the Reduce task for computing the final output.

Limitations of the basic design. This basic design of reusing the results of tasks previously performed suffers from an important limitation: it cannot reuse any work of the Reduce task. This is because the Reduce task takes as input all values for a given key ($< K_i >$, $< V_1, V_2, V_3, ...V_n >$), and therefore a single change triggers a recomputation of the entire Reduce task. Next we provide a brief background on self-adjusting computation, which forms the basis of our approach to overcome the limitations of the basic design.

2.4 Self-Adjusting Computation

Self-adjusting computations is a field that studies ways to efficiently and transparently incrementalize sequential programs [1, 2]. At a very high level, self-adjusting computations maintain a *dynamic data-dependency graph* that contains the input data to a program, all sub-computations, and the data flow between them, i.e., which outputs of sub-computations are used as inputs to other sub-computations, successively until the final output of the program is produced. Input changes are

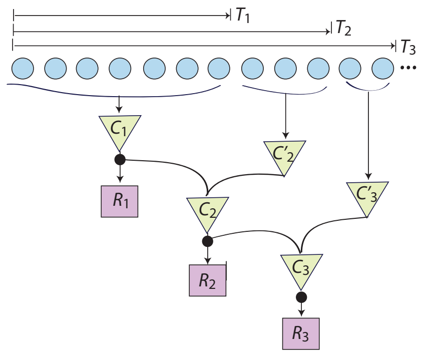


Figure 2: Coalescing contraction tree for append-only monotonically growing windows

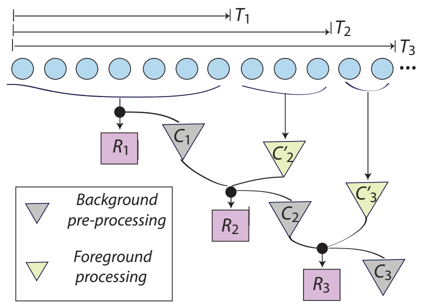


Figure 3: Coalescing contraction tree with background pre-processing

propagated through the dynamic-dependency graph by re-executing all sub-computations that are transitively affected by an input change, and re-using computations via a form of computation memoization, until the final result is updated. This *change propagation* approach can often take full advantage of previously computed results updating computations by performing optimal work. As we describe in the next section, we apply self-adjusting-computation principles to overcome the limitations of the basic design by developing novel data structures that can ensure efficiency in sliding window computation by performing incremental updates.

3. SELF-ADJUSTING TREES

We present the design of Slider. We start by explaining how Slider breaks up the work of the reduce task into sub-computations by using *self-adjusting contraction trees*. We first describe contraction trees (Section 3.1), and then present change-propagation algorithms to make them self-adjusting (Section 3.2).

3.1 Contraction Trees: Breaking up Reduce

Slider leverages *Combiner functions* of MapReduce to break a Reduce task into smaller sub-computations. Combiner functions originally aim at saving bandwidth by offloading parts of the computation performed by the Reduce task to the Map task. To use combiners, the programmer specifies a separate Combiner function, which is executed on the machine that runs the Map task, and performs part of the work done by the Reduce task in order to pre-process various <key,value> pairs, merging them into a smaller number of pairs. The combiner function takes both as an input and output type a sequence of <key,value> pairs.

We use Combiner functions to break up the work done by the (potentially large) Reduce task into many applications of the Combiner. To achieve this, we split the Reduce input into small groups, and apply the Combine to pairs of groups recursively in the form of a balanced tree until we have a single group left. We apply the Reduce function to the output of the last Combiner. We call this hierarchical structure consisting of combiner applications as a *contraction tree*. The contraction tree enables re-using the results of Combiners that remain unaffected by allowing us to propagate only the changed sub-computations down the contraction tree. At each propagation, the final Reduce task will be executed but only with the result of the Combiners that have now condensed the data.

3.2 Change Propagation for Contraction Trees

To support efficient sliding-window computations, we consider specific modes of sliding windows including append-only monotonically growing windows, fixed-width sliding windows, (Sections 3.2.1 and 3.2.2), as well as the fully general variable-width sliding windows (Section 3.2.3). For each case, we present a contract-tree data structure and a change-propagation algorithm for efficiently updating the results.

In each case, we consider two different modes of operation: 1) *initial run* where we consider the whole data set as input to compute the initial contraction tree on which subsequent changes will operate and 2) *incremental run*, where new data is added and old data is removed and the output is updated. A typical sliding-window computation starts with an initial run and follows it with a sequence of incremental runs to incorporate data changes as they come.

To improve further the responsiveness of incremental runs in the monotonically growing, and fixed-width computations, we also consider techniques for background-processing of some of the work of an incremental update. Such background-processing is optional and can be performed during times at which the cluster is underutilized.

3.2.1 Monotonically Growing Windows

Consider the case where new inputs are appended at the end of previously data to be considered as input; older data is never dropped from the beginning of the window. To process this kind of workflow, which is

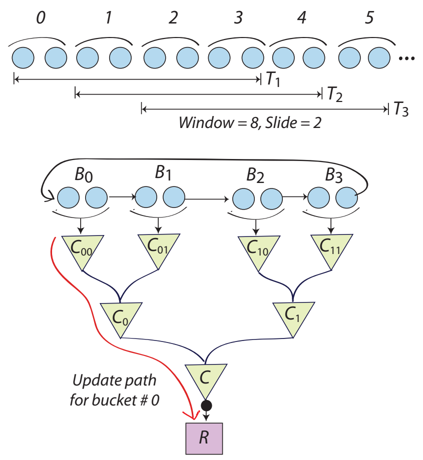


Figure 4: Rotating contraction tree for fixed-width window slides

also known as *bulk-appended data processing*, we propose what we call a *coalescing contraction tree* data structure (see Figure2).

(a) Initial run. The first time the job is invoked, it construct a one-level contraction tree by invoking the Combiner function (C_1 in Figure 2) to all Map outputs. The output of this combiner is then used as input to execute the Reduce task, which produces the final output (R_1 in Figure 2).

(b) Incremental run. To update the output efficiently, Slider constructs a single-level contraction tree on the outputs of the new Map tasks (e.g., C_2' in Figure 2) and coalesces the output with the output of tree from the previous step to form a new contraction tree (e.g., C_2 combines the outputs of C_1 and C_2'). The output of this root of the new tree then provided to a Reduce task (e.g., R_2), which produces the new output.

(c) Background processing. To improve efficiency further, we notice that we can delay the construction of the coalescing contraction tree until after we compute the new output by background processing. This modified design is depicted in Figure 3. We perform the final reduction (R_2) directly on the union of the outputs of the combiner invocation from the previous run (C_1), and the combiner invocation that aggregates the outputs of the newly run Map tasks (C_2'). In the background, we then start the new combiner that will be used in the next incremental run (C_2) by applying, in the background, the combiner function. We thus anticipate the processing that will be necessary for the next time that data is appended and prepare for it in the background.

3.2.2 Fixed-Width Window Slides

In fixed-width sliding-window computation, new data is appended at the end, and old data is dropped from the beginning. Concretely we assume that w (Map) new chunks are appended and w old chunks are removed. To perform such computations efficiently, we cluster w chunks into a bucket and build a balanced binary contraction-tree, whose leaves correspond to the buckets. Since the number of buckets remains the same during updates, we employ an evicting strategy that rotates over the leaves in a first-in-first-out fashion to perform efficient updates. We call this data structure a *rotating contraction tree* (see Figure 4).

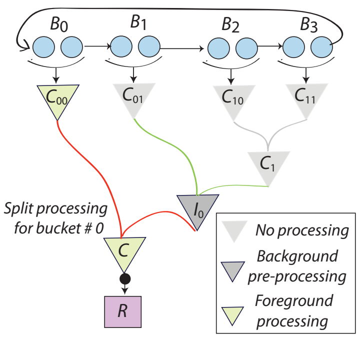


Figure 5: Rotating contraction tree with background pre-processing

(a) Initial run. Given N buckets of data, we construct a balanced contraction-tree by first combining the chunks in each bucket and then combining their results in pairs hierarchically to form a balanced binary tree of height $\lceil log_2(N) \rceil$. Figure 4 shows an example with $w = 2$. At T_1, $N = 4$ and the first level of the tree (C_{00}, C_{01}, C_{10}, C_{11}) is constructed by invoking combiners on the N buckets of size $w = 2$, whose results are then recursively combined to form a balanced binary tree. The output of the combiner at the root of the tree is then used as input to the Reduce function.

(b) Incremental run. We organize the leaf nodes of the contraction tree as a circular list. When w new chunks arrive and w old chunk leave the data set, we replace the oldest bucket with the new bucket and update the output by recomputing the path affected by new bucket. For example, in Figure 4, at T_2 the new bucket (bucket 4) replaces oldest bucket (bucket 0), and we recompute the path to the root affected by the change. Note that each step of this propagation takes as input a combination of an old combiner output, and newly produced combiner output that depend on the new bucket. For instance, in Figure 4, for propagating the change in bucket B_0, we reuse the outputs of combiners C_{01}, and C_1. The update requires updating logarithmic number of combiners.

(c) Background processing. We can further improve the latency for incremental updates by combining the known combiners outputs needed along the update path. For example, in Figure 5, we can compute in the

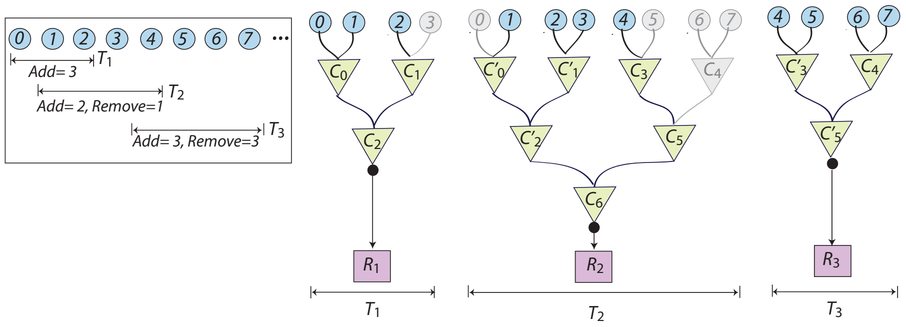


Figure 6: Folding contraction tree for variable-width window slides

background the combiner outputs C_{01}, and C_1, which are re-used, by running them through a combiner. The incremental run only needs to invoke the Reduce task with both the outputs of this background Combiner invocation and the outputs of the newly run Map tasks.

3.2.3 *Variable-Width Windows*

As the most general sliding-window computation, we consider the case where an arbitrary amount of old data can be deleted at one end and an arbitrary amount of new data can be inserted at the other end. In other words, each updates changes the size of the window varies arbitrarily. To perform efficient computations in this general case, we present a data structure that we call *folding contraction tree* (see Figure 6) that dynamically shrinks and expands by folding and unfolding whole subtrees to accommodate the current window size. The key idea behind the folding contraction tree is to maintain a contraction-tree that is nearly complete—i.e., nearly completely balanced—except at the two ends where nodes are deleted and inserted.

(a) Initial run. In the initial run, the folding tree organizes the tree of Map outputs and combiner invocations that process them in a complete binary tree. Unlike the rotating tree, this tree does not organize Map outputs into buckets, but instead builds the smallest complete contraction tree with height $\lceil log_2(maps) \rceil$. Very importantly, the leaf nodes that cannot be filled in the complete binary tree (adding up to $2^{height} - maps$ nodes), are marked as void nodes; these nodes will be occupied by future Map tasks. To complete the output, we apply Reduce to the root of the contraction tree. For example, as shown in Figure 6, at time T_1, for initial Map outputs $\{0, 1, 2\}$, we construct a complete binary tree of height 2 with a void node at location 3. Thereafter, we apply combiners at the granularity of two nodes to form $\{C_0, C_1, C_2\}$.

(b) Incremental run. When the window slides and a new run is started, the folding contraction tree shrinks and expands to accommodate changes in the window size by folding and unfolding complete subtrees. As an example, consider inserting x items and deleting y items from the window. To add items, we first try to fill up the void nodes from the previous run. If those are insufficient, then we create a complete contraction tree whose size is equal to the largest full complete subtree of the current tree. We then merge the two trees, resulting in an increase of the height of the tree by one. To remove items we discard the items from the beginning of the tree by marking their nodes as void and fold all empty complete subtrees by deleting them and their parents that now have a single child.

Figure 6 shows an example where, at time T_2, adding two outputs of Map tasks (nodes 3 & 4) causes the tree to expand to accommodate the overflowing node (node 4) by constructing another tree of height two and merging the new tree with previous tree. This increases the overall height of tree from two to three. Conversely, at time T_3 the removal of three Map outputs causes the tree height to decrease from three to two because all leaf nodes in the left half of the tree are void. Overall, as in the previous two categories, the changes in leaf nodes are propagated to the root, causing the output to be updated.

(c) Background processing. Since we have no a priori knowledge of how much data is being inserted and deleted, it is not possible to take advantage of any background processing.

4. SLIDER ARCHITECTURE

We briefly present an overview of our implementation and describe some interesting components of the architecture of Slider (Figure 7) such as the in-memory, fault-tolerant caching, (Section 4.2) and locality-aware scheduler in more detail (Section 4.3).

4.1 Implementation

We implemented our prototype of Slider based on Hadoop-0.20.2. We implemented the three variants of the self-adjusting contraction trees by inserting an additional stage between the shuffle stage (where the outputs of Map tasks are routed to the right Reduce task) and the sort stage (where the inputs to the Reduce task are sorted). To prevent unnecessary data movement in

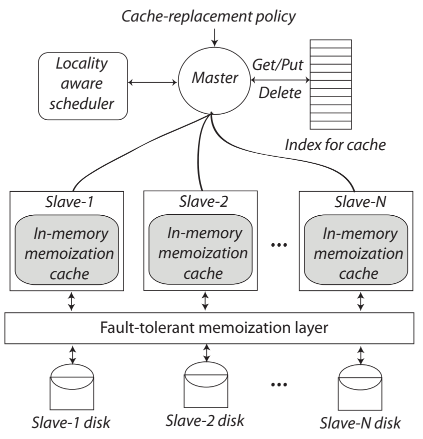


Figure 7: Components of the Slider framework

the cluster, this stage runs on the same machine as the Reduce task that will subsequently process the data.

Instead of building the trees directly as a distributed data structure, we rely on a memoization layer to remember the inputs and outputs of various tasks and the nodes of the self-adjusting contraction trees. A shim I/O layer that we designed provides access the memoization layer. The shim I/O layer transparently caches data for fast access in an in-memory distributed data cache, while also providing fault tolerance (Section 4.2). To achieve this, the shim I/O layer maintains an index at a designated master node, which is implemented using a wrapper around memcached[1]. The index tracks the objects stored in the main memory and the HDFS distributed file system, which replicates data as needed.

We implement the background pre-processing phase with a hook to enable single-phase MapReduce execution for pre-processing the self-adjusting tree. Since the Map and Reduce phase are not required for background pre-processing, their execution is disabled in this mode of operation.

4.2 In-Memory Distributed Data Caching

To provide fast access to memoized results, we designed an in-memory distributed data caching layer. Our use of in-memory caching is motivated by two observations: *(i)* main memory is generally underutilized in data-centric computing [5]; and *(ii)* the number of sub-computations that need to be memoized is limited by the size of the sliding window. The distributed in-memory cache is coordinated by a master node, which maintains an index to locate the data items. The master implements a simple cache replacement policy, which can be changed in accordance with the workload characteristics; the default is the Least Recently Used (LRU).

Storing memoized results in an in-memory cache is beneficial for performance but is prone to machine or memory failures, which can wipe out the benefits, by requiring unnecessary re-computation. We therefore conduct a background replication of memoized results to provide fault tolerance, by creating three replicas of each memoized result. This way fault tolerance is handled transparently: when a new task wants to fetch a memoized result it reads that result from one of the replicas. To ensure that the storage requirements remain bounded, we developed a "garbage collection" algorithm that frees the storage used by results that fall out of the current window.

4.3 Locality-Aware Scheduling

In Slider we modified Hadoop's original scheduler to be aware of the location of memoized results. Hadoop's scheduler chooses the first available node (machine) to run a pending Reduce task, taking locality into account only when scheduling Map tasks by biasing towards the node holding the input. Slider's scheduler adapts previous work in data-locality scheduling [7, 3], to run Reduce tasks on the machine that contains the results of the combiner function that it uses. When the scheduler detects that the favored node (of a Reduce tasks) is overloaded, it migrates tasks from the overloaded node to another node, taking care to migrate also the relevant memoized results. Such migration is critical to prevent significant performance degradation due to straggler tasks [31], which can delay the completion of a job.

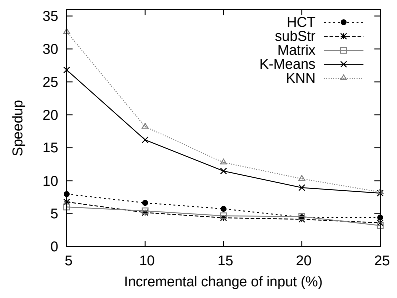
5. EVALUATION

In this section we evaluate a number of microbenchmarks before exploring how Slider performs with real-world use cases of networked system analysis in the following section (§ 6). Our experiments using microbenchmarks aim at answering the following questions:

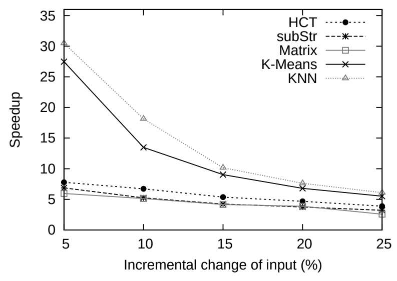
- What performance benefits does Slider provide for incremental sliding-window computations compared to recomputing over the entire window of data?
- What are the overheads imposed by Slider for a fresh run of an application?
- How effective are the optimizations we propose in improving the performance of Slider?

Our microbenchmarks, listed in Table 1, span five data analysis tasks typically performed in networked systems today. Two of them are compute-intensive tasks: Clustering based on the Euclidean distance (K-Means) and object classification using K-nearest neighbors (KNN). As input to these tasks, we use a set of points in a d-dimensional space. We generate this data synthetically by randomly selecting points from a 50-dimensional unit cube. The remaining three tasks are data (I/O) intensive computations: Histogram-based computations (HCT), matrix computations (Matrix), and pattern matching (subStr). As input, we use a publicly available

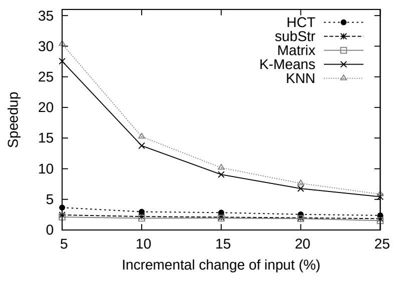
[1]Memcached: `http://memcached.org/`



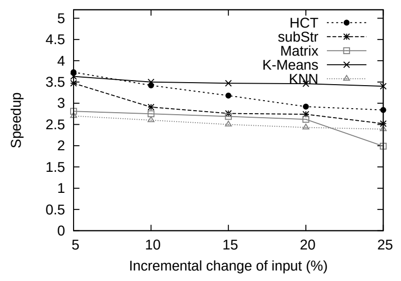
(a) Work speedup for append only



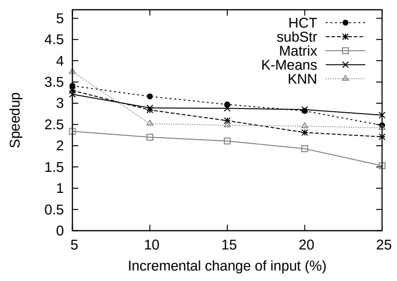
(b) Work speedup for fixed-width



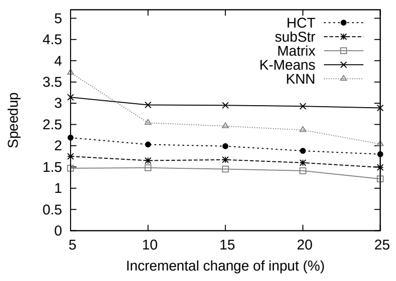
(c) Work speedup for variable-width



(d) Time speedup for append only



(e) Time speedup for fixed-width



(f) Time speedup for variable-width

Figure 8: Performance gains of Slider for incremental runs

Class (`Abbrv.`)	**Application Description**
Clustering (`K-Means`)	K-means clustering for partitioning n data points into k clusters based on Euclidean distance
Histogram, CDFs, Top-K (`HCT`)	Computes histogram over all x and then outputs CDF(x) and top-K items
Machine learning(`KNN`)	K-nearest neighbors classifies objects based on the closest training examples in a feature space
Matrix computations (`Matrix`)	Co-occurrence matrix of size $N \times N$ matrix, with N being the number of unique items in the data-set. A cell m_{ij} contains the number of times item w_i co-occurs with item w_j
Pattern matching (`subStr`)	Extracts frequently occurring sub-strings (bi-gram count) from a given corpus

Table 1: Applications used in Microbenchmarks

dataset with the contents of Wikipedia[2].

To ensure reasonable running times, we adjusted the input sizes such that the running time of each job would be around one hour for all five applications.

Experimental setup. Our experiments run on a cluster of 25 machines, running Linux with kernel 2.6.32 in 64-bit mode, connected with gigabit ethernet. We configured Hadoop so that the name node and the job tracker ran on a master machine which was equipped with a 12-core Intel Xeon processor and 48 GB of RAM. The data nodes and task trackers ran on the remaining 24 machines equipped with AMD Opteron-252 processors, 4 GB of RAM, and 225 GB drives. We configured the task trackers to use two Map and two Reduce slots per worker machine.

[2]Wikipedia data-set: `http://wiki.dbpedia.org/`

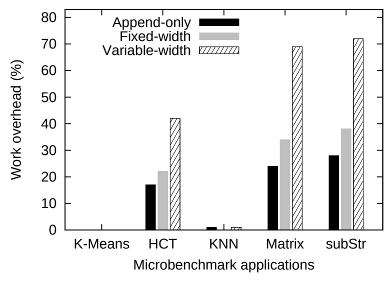
5.1 Methodology

Using the microbenchmarks described above, we compare the execution performance of Slider with the performance of a pure MapReduce implementation.

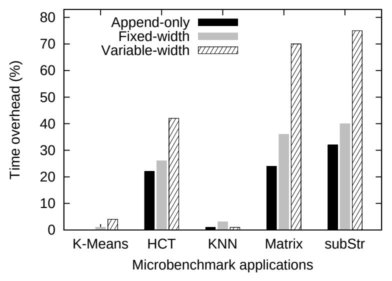
Measurements. We consider two types of measures: *work* and *running time* (or *time*). These are standard measures for comparing efficiency in parallel and distributed computations. Work refers to the total amount of computation performed by all tasks and is measured as the total running time of all tasks. Time refers to the amount of (end-to-end) running time to complete a parallel computation. It is well-known that a computation with W work can be executed on P processors in $\frac{W}{P}$ time plus some scheduling overheads. This is called the work-time principle.

Note that the work measurements include the additional computational work performed by tasks that are speculatively executed by the Hadoop framework (e.g., Hadoop can run the same task on two different machines to improve performance if there is spare capacity on the cluster). Therefore, a difference in the number of speculative tasks that are launched will be reflected in the comparison of work.

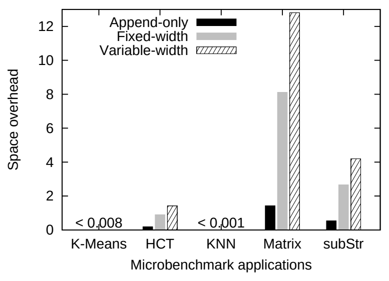
Initial and incremental runs. For Slider, we distinguish two types of runs. First, the *initial run* refers to a run starting with no memoized results. Such a run executes all tasks and populates the memoization layer by storing information about the sub-computations that



(a) Work overhead



(b) Time overhead



(c) Space overhead

Figure 9: Overheads of Slider for initial run

were performed and the location of their results. Second, an *incremental run* refers to the runs of the same job in the steady state, where the input is evolving and Slider can usually avoid a full re-computation by using previous results stored in the memoization layer.

To assess the effectiveness of Slider, we measure the work and running time of each microbenchmark for different dynamic update scenarios, i.e., with different amounts of modified inputs, ranging from 5% up to 25% of input data change. For the append-only case, a p% incremental change of the input data means that p% more data was appended to the existing data. For the fixed-with and variable-width sliding window cases, the window is moved by having p% of the input buckets dropped from its beginning, and replaced with the same number of new buckets in its end, containing newly generated content.

We then compare the work and running time of Slider to an unmodified Hadoop implementation. The ratio between the two is the *speedup* for work and running time, respectively.

5.2 Microbenchmark Results

Our microbenchmark evaluation is structured by answering the sequence of questions raised in the beginning of the section.

5.2.1 Speedup

We begin by measuring the performance gains of Slider in comparison to Hadoop. The results in Figure 8 show that Slider achieves substantial performance gains for all microbenchmarks, independently of the category of sliding window computations.

The gains for computation-intensive microbenchmarks (K-Means and KNN) are the most substantial, with time and work speedups between 1.5 and 35-fold. As expected, the speedup decreases as the overlap between the old and the new window becomes smaller. Nonetheless, for these two benchmarks, even for a 25% input change, the speedup is still between 1.5 and 8-fold.

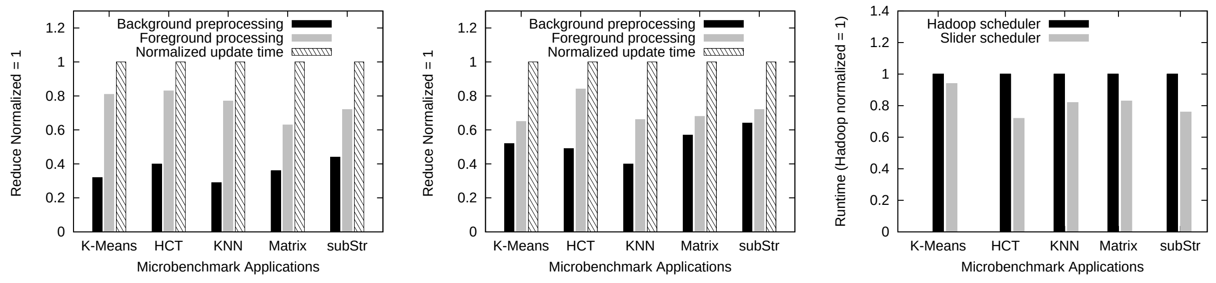
Speedups for data-intensive microbenchmarks (HCT, Matrix, and subStr) are between 8-fold and 1.5-fold. Even though this is a very positive result, the speedup figures are lower than in the case of microbenchmarks with a higher ratio of computation to I/O. This is because the basic approach of memoizing the outputs of previously run sub-computations is effective at avoiding the CPU overheads but still requires some data movement to transfer outputs of sub-computations, even if they were memoized. For this set of micro-benchmarks, the performance gains for variable-width sliding window computations are lower than for the append-only and fixed-width window cases. This is because, updates require rebalancing the self-adjusting tree, thus incurring in a higher overhead.

The results also show that, for very small changes, speedups in work are not fully translated into speedups in running time. This is expected because decreasing the total amount of work dramatically reduces the amount of parallelism, leading to higher scheduling overheads. As the size of the incremental change increases, the gap between the work speedup and time speedup closes quickly.

5.2.2 Overhead

Slider adds two types of overhead. First, building and maintaining the self-adjusting trees causes a *running time and work overhead*. Second, Slider introduces a *space overhead* for memoizing intermediate results that are used to speed up incremental computations. Figure 9 plots the overhead we measured for different microbenchmark applications and categories of sliding-window computations. Note that the time and work overheads are one-time costs for the initial computation. For subsequent incremental runs, Slider achieves the speedups we described.

Performance overheads. Figure 9(a) and Figure 9(b) show the work and running time overheads for the initial runs of each microbenchmark, compared to a computation using a pure MapReduce implementation. This essentially captures the overhead for constructing the initial self-adjusting tree. Computation-intensive microbenchmarks show low overhead as their running time is dominated by the actual processing time. Compared to this, the extra computation Slider requires is small. For data-intensive microbenchmarks running time is dom-



(a) Split processing in append-only case (b) Split processing in fix-width case (c) Scheduling gains for Slider

Figure 10: Effectiveness of optimizations in Slider

inated by storing, retrieving and transferring input data. The running time overhead is higher in this case as memoizing intermediate results competes with the data analysis job for I/O.

The overheads for variable-width are higher than those for fixed-width computations, and significantly higher than the append-only case. This overhead comes from having more levels in the corresponding self-adjusting tree. In particular, for the comparison between fixed and variable-width windows, the leaf nodes correspond to smaller buckets in the case of variable-width computations, and this leads to a higher tree depth, and therefore to running a larger number of Combiner functions, which introduces more overhead.

Space overhead. Figure 9(c) plots the space overhead normalized by the input size. Again, the variable-width sliding-window computation shows the highest overhead, requiring more space than the other computations, for the same reasons that were mentioned in the case of the performance overheads.

Space overhead highly depends on the applications. The Matrix computation has the highest space overhead of 12X, while K-Means and KNN have almost no space overhead. However, the garbage collector limits the storage demands of Slider over time.

5.2.3 Effectiveness of Optimizations

We evaluate the effectiveness of the optimizations in improving the overall performance of Slider by considering: *(a)* latency savings from splitting the execution into foreground and background processing. *(b)* the scheduler modifications to exploit the locality of memoized results; *(c)* the performance gains of using an in-memory distributed data caching layer.

Split processing. Slider is designed to take advantage of the predictability of future updates to split the work between a background pre-processing and foreground processing, as described in Section 4. To evaluate the effectiveness of this optimization, we compared the cost of executing with and without it, for both the append-only and the fixed-width window categories. Figures 10(a) and 10(b) show the time required for background preprocessing and foreground preprocessing, normalized to the time for processing the update without any background pre-processing. Figure 10(a) shows this cost when a new input with 5% the original input size is appended, for different benchmarking applications, whereas Figure 10(b) shows the same cost for a 5% input change in the fixed-width window model. The results show that with the split processing model, on average we are able to perform foreground updates up to 25%-40% faster, while offloading around 36%-60% of the work for background pre-processing.

The results also show that the sum of the cost of background pre-processing and foreground processing exceeds the normal update time, mostly because of the extra merge operation performed due to split processing model. Note that background pre-processing can be performed during low cluster utilization periods, and therefore the results confirm that the latency gains are likely to be worth the extra overall processing time.

Scheduler modification. We now evaluate the effectiveness of our scheduler modifications in improving the performance of Slider. We compare the performance of the Hadoop scheduler with the Slider scheduler in Figure 10(c), where the Y-axis shows the running time of Slider, relative to using the Hadoop scheduler. As evident from the figure, the Slider scheduler saves around 30% of time for data-intensive applications, and almost 15% of time for compute intensive applications, which supports the effectiveness of location aware scheduling to avoid transferring memoized results.

In-memory distributed memoization caching. For evaluating the effectiveness of performing in-memory data caching, we compared our performance gains with and without this caching support. In particular, we disabled the in-memory caching support from the shim I/O layer, and instead used the fault-tolerant memoization layer for storing the memoized results. Therefore, when accessing the fault-tolerant memoization layer, we incur in an additional cost of fetching the data from the disk or network. Table 2 shows the savings in Reduce completion time with in-memory caching for the case of a fixed-width window. This shows that we can achieve

K-Means	HCT	KNN	Matrix	subStr
48.68%	56.87%	53.19%	67.56%	66.2%

Table 2: Reduction in average read time (in %) for Reducers with in-memory caching

50% to 68% savings in Reduce time by using an in-memory memoization layer.

6. CASE STUDIES

In this section, we evaluate Slider with three real-world use-cases of data analysis covering three modes of operation (append only, fixed-with, and variable-width) for sliding-window computations. Our case studies span a variety of networked systems that includes Online Social Networks (OSNs), network monitoring tool, and hybrid Content Distribution Networks (CDNs).

6.1 Information Propagation in Twitter

Analyzing information propagation in online social networks, such as Facebook or Twitter, is an active area of research. We used Slider to analyze how web links are spread in Twitter, repeating an analysis done by Rodrigues et al. [28].

The propagation of URLs in Twitter is tracked by building an information propagation tree for every posted URL based on Krackhardt's hierarchical tree model. This tree tracks URL propagation by maintaining a directed edge between a spreader of an URL and a receiver, i.e., a Twitter user "following" the Twitter account that posted the link. The root node of the tree represents the original publisher of a web link.

We used the Twitter data in [28], which comprises 54 million user profiles, 1.9 billion follow-relations, and all 1.7 billion tweets posted by Twitter users between March 2006 and September 2009. To create a workload where data is gradually appended to the input, we partitioned the dataset in five non-overlapping time intervals as listed in Table 4. The first time interval captures all tweets from the inception of Twitter up to June 2009. We then add one week worth of tweets for each of the four remaining time intervals. For each of these intervals, an average of 5% of new data is appended.

We present the performance gains of incrementally building the information propagation tree using Slider in Table 4. The speedups are almost constant for the four time intervals, at about $8X$ for running time and about $14X$ for work. The running time overhead for computing over the initial interval is 22%. We also repeated other analyses performed in [28] and achieved similar speedups. We omit the results due to space constraints. In summary, these results highglight that the performance gains observed with our microbenchmarks also hold for this particular use case.

Time interval	Mar'06 - Jun'09	Jul'09 1-7th	Jul'09 8-14th	Jul'09 15-21st	Jul'09 22-28th
Number of tweets	1464.3 M	74.2 M	81.5 M	79.4 M	85.6 M
Cummulative change (%)	-	5.1%	5.3%	4.9 %	5.0%
Time speedup	-	8.9	9.2	9.42	9.25
Work speedup	-	14.22	13.67	14.22	14.34

Table 4: Summary of the Twitter data analysis

6.2 Monitoring of a Networked System

Glasnost [14] is a system that enables users to detect whether their broadband traffic is shaped by their ISP. The Glasnost system is deployed on over 70 servers around the globe and has been used by millions of users since 2009. One of the features of the monitoring system is that it tries to direct users to a close by measurement server. Slider enabled us to evaluate the effectiveness of this server selection.

For each Glasnost test run, a packet trace of the measurement traffic between the Glasnost measurement server and the user's host is stored. We used this trace to compute the minimum round-trip time (RTT) between the server and the user's host, which represents the distance between the two. Taking all minimum RTT measurements of a specific measurement server, we computed the median and 95th-percentile across all users that were directed to this server. This gives a good assessment of the quality of server selection in Glasnost.

For this analysis, we used the data collected by a given Glasnost server between January and November 2011 (see Table 3). We started with the data collected from January to March 2011. Then, we added the data of one subsequent month at a time and compute the median and 95th-percentile distance between users and the measurement server for a window of the most recent 3 months. This particular measurement server had between $4,033$ and $6,536$ test runs per 3-month interval, which translate to 7.8 GB to 18 GB of data per interval that needs to be processed.

We measured the performance gains for both work and running time as shown in Table 3. The results show that we get an average speedup in the order of $2.5X$, with small overheads of less than 5%. An anecdotal report that highlights the importance of these performance gains is that our colleagues that maintained Glasnost were, at the time of their work, not able to perform this useful analysis as frequently as they desired, due to its high processing demands.

6.3 Accountability in Hybrid CDNs

Year 2011	Jan-Mar	Feb-Apr	Mar-May	Apr-Jun	May-Jul	Jun-Aug	Jul-Sep	Aug-Oct	Sep-Nov
No. of pcap files	4033	4862	5627	5358	4715	4325	4384	4777	6536
Window change size	4033	1976	1941	1441	1333	1551	1500	1726	3310
% change size	100 %	40.65 %	34.50 %	26.89 %	28.27 %	35.86 %	34.22 %	36.13 %	50.64 %
Time speedup	-	2.07	2.8	3.79	3.32	2.44	2.56	2.43	1.9
Work speedup	-	2.13	2.9	4.12	3.37	3.15	2.93	2.46	1.91

Table 3: Summary of the Glasnost network monitoring data analysis

Content distribution networks (CDNs), recently started working on hybrid CDNs, which employ peer-to-peer (P2P) technology to add end user nodes to the distribution network, this way enhancing their existing server-based infrastructure. Using this architecture, CDN providers can cut costs as less servers need to be deployed. However, this also raises questions about the integrity of the answers that are provided by these untrusted clients [4]: their cooperation is required both to evaluate how much data a client served (for billing purposes) and to ensure that the data served by the client is correct.

Aditya et al. [4] presented a design of a hybrid CDN that employs a tamper-evident logs to provide client accountability. This log is uploaded to a set of servers that need to audit the log periodically using techniques based on the PeerReview [17]. Using Slider, we implemented these audits as a variable-sized sliding-window computation, where the amount of data in a window varies depending on the availability of the clients in the hybrid-CDN during a given time period. The MapReduce data analysis application implements log consistency checks which verify whether all clients report a consist record of the network messages exchanged and whether this record is plausible.

To evaluate the effectiveness of Slider, we used a synthetic dataset generated using trace parameters available from the Akamai's NetSession system, a peer-assisted CDN operated by Akamai (which currently has 24 million clients). From this data set, we selected the data collected in December 2010. However, due to the limited compute capacity of our experimental setup, we scaled down the data logs to $100,000$ clients. In addition to this input, we also also generated logs corresponding to one week of activity with a varying percentage of clients (from 100% to 75%) uploading their logs to the central infrastructure, so that the input size varies across weeks. This allows us to create an analysis with a variable-width sliding window by using a window corresponding to one month of data and sliding it by one week in each run.

Table 5 plots the performance gains for log audits for a different percentage of client log uploads for the 5th week. We observe a speedup of $2X$ to $2.5X$ for log upload probabilities between 75% and 100%. Similarly, the running time speedups are between $1.5X$ and $2X$. This example highlights that Slider achieves impressive speedups in a realistic scenario, even in the more challenging case where we cannot predict in advance how many items will be added when the window moves.

Clients uploading logs in the 5^{th} week (in %)	100%	95%	90%	85%	80%	75%
Time speedup	1.72	1.85	1.89	2.01	2.1	2.24
Work speedup	2.07	2.21	2.29	2.44	2.58	2.74

Table 5: Summary of the Akamai NetSession analysis

7. DECLARATIVE QUERY INTERFACE

As Slider is based on Hadoop, computations can be implemented using the MapReduce programming model. While MapReduce is gaining in popularity, many programmers are more familiar with declarative query interfaces, as provided by SQL or LINQ. To ease the adoption of Slider, we also provide a declarative query interface that is based on Pig[3]. Pig consists of a high-level language, which is similar to SQL, and a compiler that translates Pig programs to sequences of MapReduce jobs. As Slider transparently extends MapReduce without changing its programming model, Pig can use Slider as an execution engine and needs no adjustments; queries automatically use the incremental update mechanism provided by Slider.

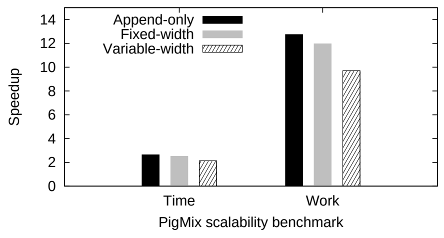


Figure 11: Query processing speedups using Slider

To demonstrate the potential of incremental query-based sliding-window computations, we evaluate Slider using the PigMix[4] benchmark. We ran the benchmark in our three modes of operation with changes in 5% of its input. Figure 11 shows the resulting running time and work speedups. As expected, the results are in line with the previous evaluation, since ultimately the queries are

[3]Apache Pig: `http://pig.apache.org/`

[4]Apache PigMix: `http://wiki.apache.org/pig/PigMix`

compiled to a set of MapReduce analyses. We observe an average speedup of $2.5X$ and $11X$ for time and work, respectively. Finally, we observe an average overhead of 22% for the initial run, and space overhead of $2X$ the input size.

8. RELATED WORK

Sliding-window computations and incremental computations have been studied extensively but often independently. In this section, we describe related work on sliding-window data analyses in networked systems and on incremental computations.

Dynamic algorithms. Algorithms researchers design dynamic and data-streaming algorithms that permit changes to their input dynamically while efficiently updating their output. Several surveys discuss the vast literature on dynamic algorithms (e.g., [9, 27, 13, 6]). This research shows that dynamic algorithms can be asymptotically faster (often by a near-linear factor) than their conventional counterparts. Dynamic algorithms can, however, be difficult to develop and implement even for simple problems; some problems took years of research to solve.

Programming language-based approaches. Programming languages researchers developed incremental computation techniques to achieve automatic incrementalization (e.g. [27, 1]). The goal is to incrementalize programs automatically without significantly sacrificing efficiency. Recent advances on self-adjusting computation made significant progress towards this goal by proposing general-purpose techniques that can achieve optimal update times (e.g., [1, 2]). That work, however, primarily targets sequential computations that are primarily compute-intensive. We develop techniques to perform sliding-window computations on big data by adapting the principles of self-adjusting computation for a large-scale parallel and distributed execution environment. In this specific domain, we also achieve a full transparency, requiring no work from the programmer to annotate their code with special primitives.

Database systems. There is substantial work from the database community on incrementally updating a database view (i.e., a predetermined query on the database) as the database contents change. The techniques they use maintain views under incremental changes, or rely on SQL queries to efficiently compute the modifications to the database view [8]. Building on the experience of database systems, we provide a declarative query interface (similar to SQL) to improve usability. Our work differs significantly from the view maintenance work in databases because we consider large-scale computations on big data, and allow sliding-window computations rather than more structured changes to database tables.

Distributed systems. Since large-scale processing of unstructured data sets is an increasingly common and important task for Internet services, researchers and practitioners have built a wide range of distributed systems for incremental computations [26, 15, 22, 7, 23]. Slider is designed to operate at the same scale and with a similar functionality for distributed execution, including a simple programming model, data-parallelization, fault tolerance and scheduling. In particular, Slider builds on our own previous system called Incoop [7], namely in its use of Combiner functions to break up the work of Reduce tasks. However, none of these approaches, including our own work, consider sliding-window computations. Handling these required several novel technical contributions, for example, in terms of the data structures that are employed for propagating changes.

Batched stream processing. Stream processing engines [18, 10, 25] model the input data as a stream, with data-analysis queries being triggered upon bulk appends to the stream. These stream processing engines exploit temporal and spatial correlations in recurring queries by aligning the execution of multiple queries together when new bulk updates occur, and thereby exploiting redundant I/O or computation across queries. Stream processing systems are especially designed for bulk-appended data processing, which is only a special case of our more general sliding window model. Slider also improves on Comet [18] and Nova [25] by being not requiring the programmer to devise a dynamic algorithm, preserving the transparency relative to a single-pass non-incremental data analysis. Hadoop online [10] is transparent but does not attempt to break up the work of the Reduce task, which is one of the key contributions (and sources of performance gains) of this work.

9. CONCLUSIONS

In this paper we presented Slider, a novel framework for incremental sliding-window data analyses in networked systems. Slider does not require the analysis code to be rewritten, and contains several technical contributions like novel data structures that enable efficient incremental updates. Our evaluation using microbenchmarks and three real world case studies shows that Slider can significantly improve their performance.

10. REFERENCES

[1] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 2005.

[2] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM TOPLAS*, 2009.

[3] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA*, 2000.

[4] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable client accounting for hybrid content-distribution networks. In *NSDI*, 2012.

[5] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI*, 2012.

[6] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Sliding window computations over data streams. Technical Report 2002-25, April 2002.

[7] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *SoCC*, 2011.

[8] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[9] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, pages 1412–1434, 1992.

[10] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[11] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. In *SIGCOMM*, 2004.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[13] C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*, chapter 35: Dynamic Trees. Dinesh Mehta and Sartaj Sahni (eds.), CRC Press Series, 2005.

[14] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling End Users to Detect Traffic Differentiation. In *NSDI*, 2010.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI*, 2010.

[16] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *NSDI*, 2009.

[17] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

[18] B. He and e. a. Yang, Mao. Comet: batched stream processing for data intensive distributed computing. In *SoCC*, 2010.

[19] S. Kandula, R. Chandra, and D. Katabi. What's going on?: learning communication rules in edge networks. In *SIGCOMM*, 2008.

[20] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *SIGCOMM*, 2004.

[21] L. Le, J. Aikat, K. Jeffay, and F. D. Smith. The effects of active queue management on web performance. In *SIGCOMM*, 2003.

[22] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, 2010.

[23] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ mapreduce for log processing. In *USENIX ATC*, 2011.

[24] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: existing techniques and new directions. In *SIGCOMM*, 2002.

[25] C. Olston and et. al. Nova: continuous pig/hadoop workflows. In *SIGMOD*, 2011.

[26] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[27] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[28] T. Rodrigues, F. Benevenuto, M. Cha, K. Gummadi, and V. Almeida. On word-of-mouth based discovery of the web. In *IMC*, 2011.

[29] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI*, 2004.

[30] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. In *SIGCOMM*.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[32] Y. Zhang and V. Paxson. Detecting stepping stones. In *USENIX Security*, 2000.

APPENDIX

A. ANALYSIS OF SLIDER

We analyze the asymptotic efficiency of Slider. We consider two different runs: the *initial run* of an Slider computation, where we perform a computation with some input I, and a second run for *dynamic update* where we change the input from I to I' and perform the same computation with the new input. In the common case, we perform a single initial run followed by many dynamic updates.

For the initial run, we define the *overhead* as the slowdown of Slider compared to a conventional implementation of MapReduce such as with Hadoop. We show that the overhead depends on communication costs and, if these are independent of the input size, which they often are, then it is also constant. Our experiment evaluation confirms that the overhead is relatively small. We show that dynamic updates are dominated by the time it takes to execute fresh tasks that are affected by the changes to the input data, which, for a certain class of computations and small changes, is logarithmic in the size of the input.

In the analysis, we use the following terminology to refer to the three different types of computational tasks that form an Slider computation: Map tasks, Self-adjusting balanced tree (applications of the Combiner function for three different modes of operation for sliding-window computations), and Reduce tasks.

Our bounds depend on the total number of map tasks, written N_M, and the total number of reduce tasks written N_R. In addition, we also take in account the total number of stages in self-adjusting balanced tree, denoted as N_C. We write n_i and n_O to denote the total size of the input and output respectively, n_m to denote the total number of key-value pairs output by the Map phase, and n_{mk} to denote the set of distinct keys emitted by the Map phase. The number of stages in self-adjusting balanced tree is a property of sliding-window computation mode: append-only ($N_{CA} = O(n_{mk})$), fixed-width window slides ($N_{CF} = O(n_{mk} \cdot \lceil log_2(buckets) \rceil)$), and variable-width window slides((N_{CV}) = $\lceil O(n_{mk} \cdot log_2(N_M)) \rceil$)).

For our time bounds, we will additionally assume that each Map, Combine, and Reduce function performs work that is asymptotically linear in the size of their inputs. Furthermore, we will assume that the Combine function is *monotonic*, i.e., it produces an output that is no larger than its input. This assumption is satisfied in most applications, because Combiners often reduce the size of the data (e.g., a Combine function to compute the sum of values takes multiple values and outputs a single value).

THEOREM 1 (INITIAL RUN:TIME AND OVERHEAD). *Assuming that Map, Combine, and Reduce functions take time asymptotically linear in their input size and that Combine functions are monotonic, total time for performing an incremental MapReduce computation in Slider with an input of size n_i, where $n_m$key-value pairs are emitted by the Map phase is $O(N_M + (N_R + N_C)) = O(n_i + n_m)$. This results in an overhead of $O(N_C) = O(N_{CA}||N_{CF}||N_{CV})$ over conventional MapReduce.*

PROOF. The number of Map and Reduce tasks in a particular job can be derived from the input size and the number of distinct keys that are emitted by the Map function: the Map function is applied to *splits* that consist of one or more input chunks, and each application of the Map function is performed by one Map task. Hence, the number of Map tasks N_M is in the order of input size $O(n_i)$. In the Reduce phase, each Reduce task processes all previously emitted key-value pairs for at least one key, which results in at most $N_R = n_{mk}$ reduce tasks. To bound the number of self-adjusting balanced tree, we note that the tree leaves are the output data chunks of the Map phase, whose internal nodes each has at least two children. Since there are at most $n_m$pairs output by the Map phase, the total number of reduce tasks is bounded by n_m. Hence the total number of stages in self-adjusting balanced tree is bounded by $N_C \in O(n_m)$. Since the number of reduce tasks is bounded by $n_{mk} \leq n_m$, the total number of tasks is $O(n_i + n_m)$. □

THEOREM 2 (INITIAL RUN: SPACE). *Total storage space for performing an Slider computation with an input of size n_i, where $n_m$key-value pairs are emitted by the Map phase, and where Combine is monotonic is $O(n_m)$.*

PROOF. Slider requires additional storage space for storing the intermediatery output of the self-adjusting balanced tree. Since Slider only keeps data from the most recent run (initial or dynamic run), we use storage for remembering only the task output from the most recent run. The output size of the map tasks is bounded by n_m. With monotonic Combine functions, the size of the output of Combine tasks is bounded by $O(n_m)$. □

THEOREM 3 (DYNAMIC UPDATE: SPACE AND TIME). *In Slider, a dynamic update requires time, where F where F is the set of changed or new (fresh)* Map*,* Combiner*, and* Reduce *tasks, is*

$$O\left(\sum_{a\in F} t(a)\right).$$

The total storage requirement is the same as an initial run.

PROOF. Consider Slider performing an initial run with input I and changing the input to I' and then performing a subsequent run (dynamic update). During the dynamic update, tasks with the same type and input data will re-use the memoized result of the previous runs, avoiding recomputation. Thus, only the fresh tasks need to be executed, which takes $O\left(\sum_{a\in F} t(a)\right)$, where F is the set of changed or new (fresh) *Map*, *Contract* and *Reduce* tasks, respectively, and $t(\cdot)$ denotes the processing time for a given task. □

In the common case, we expect the execution of fresh tasks to dominate the time for dynamic updates. The time for dynamic update is therefore likely to be determined by the number of fresh tasks that are created as a result of a dynamic change. It is in general difficult to bound the number of fresh tasks, because it depends on the specifics of the application. As a trivial example, consider, inserting a single key-value pair into the input. In principle, the new pair can force the Map function to generate a very large number of new key-value pairs, which can then require performing many new reduce tasks. In many cases, however, small changes to the input lead only to small changes in the output of the Map, Combine, and Reduce functions, e.g., the Map function can use one key-value pair to generate several new pairs, and the Combine function will typically combine these, resulting in a relatively small number of fresh tasks. As a specific case, assume that the Map function generates k key-value pairs from a single input record, and that the Combine function monotonically reduces the number of key-value pairs.

THEOREM 4 (NUMBER OF FRESH TASKS). *If the Map function generates k key-value pairs from a single input record, and the Combine function is monotonic, then the number of fresh tasks is at most $O(k \log n_m + k)$.*

PROOF. At most k combine at each level of the self-adjusting tree will be fresh, and k fresh reduce tasks will be needed. Since the depth of the contraction tree is n_m, the total number of fresh tasks will therefore be $O(k \log n_m + k) = O(k \log n_m)$. □

Taken together the last two theorems suggest that small changes to data will lead to the execution of only a small number of fresh tasks, and based on the tradeoff between the memoization costs and the cost of executing fresh tasks, speedups can be achieved in practice.