Slider: Incremental Sliding Window Analytics

Pramod Bhatotia
MPI-SWS
bhatotia@mpi-sws.org

Umut A. Acar
CMU and INRIA
umut@cs.cmu.edu

Flavio P. Junqueira
Microsoft Research
fpj@microsoft.com

Rodrigo Rodrigues
NOVA Univ. Lisbon/CITI/NOVA-LINCS
rodrigo.rodrigues@fct.unl.pt

ABSTRACT

Sliding window analytics is often used in distributed data-parallel computing for analyzing large streams of continuously arriving data. When pairs of consecutive windows overlap, there is a potential to update the output incrementally, more efficiently than recomputing from scratch. However, in most systems, realizing this potential requires programmers to explicitly manage the intermediate state for overlapping windows, and devise an application-specific algorithm to incrementally update the output.

In this paper, we present self-adjusting contraction trees, a set of data structures and algorithms for transparently updating the output of a sliding window computation as the window moves, while reusing, to the extent possible, results from prior computations. Self-adjusting contraction trees structure sub-computations of a data-parallel computation in the form of a shallow (logarithmic depth) balanced data dependence graph, through which input changes are efficiently propagated in asymptotically sub-linear time.

We implemented self-adjusting contraction trees in a system called Slider. The design of Slider incorporates several novel techniques, most notably: (i) a set of self balancing trees tuned for different variants of sliding window computation (append-only, fixed-width, or variable-width slides); (ii) a split processing mode, where a background pre-processing stage leverages the predictability of input changes to pave the way for a more efficient foreground processing when the window slides; and (iii) an extension of the data structures to handle multiple-job workflows such as data-flow query processing. We evaluated Slider using a variety of applications and real-world case studies. Our results show significant performance gains without requiring any changes to the existing application code used for non-incremental data processing.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems

General Terms

Algorithms, Design, Experimentation, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Middleware'14, December 08 – 12, 2014, Bordeaux, France.
Copyright 2014 ACM 978-1-4503-2785-5/14/12
http://dx.doi.org/10.1145/2663165.2663334 ...$15.00.

1 Introduction

There is a growing use of "big data" systems for the parallel analysis of data that is collected over a large period of time. Either due to the nature of the analysis, or in order to bound the computational complexity of analyzing a monotonically growing data set, applications often resort to a *sliding window* analysis. In this type of processing, the scope of the data analysis is limited to an interval over the entire set of collected data, and, periodically, newly produced inputs are appended to the window and older inputs are discarded from it as they become less relevant to the analysis.

The basic approach for sliding window analytics is to recompute over the entire window from scratch whenever the window slides. Consequently, even old, unchanged data items that remain in the window are reprocessed, thus consuming unnecessary computational resources and limiting the timeliness of results.

One way to improve on the basic approach is to use incremental update mechanisms, where the outputs are updated to accommodate the arrival of new data instead of recomputing them from scratch. Such incremental approaches can be significantly—often asymptotically—more efficient than the basic approach, particularly in cases when the size of the window is large relative to increment by which the window slides.

The most common way to support incremental computation is to rely on the application programmers to devise an incremental update mechanism [28, 30, 34]. In such an approach, the programmer has to design and implement a *dynamic algorithm* containing the logic for incrementally updating the output as the input changes. While dynamic algorithms can be efficient, research in the algorithms community shows that they are often difficult to design, analyze, and implement even for simple problems [8, 22, 25, 37]. Moreover, dynamic algorithms are overwhelmingly designed for the uniprocessor computing model, making them ill-suited for the parallel and distributed systems used in large-scale data analytics.

Given the efficiency benefits of incremental computation, our work answers the following question: *Is it possible to achieve the benefits of incremental sliding window analytics without requiring dynamic algorithms?* Previous work on incremental computation in batch-processing systems [18, 19, 27] shows that such gains are possible to obtain in a transparent way, i.e., without changing the original (single pass) data analysis code. However, these systems did not leverage the particular characteristics of sliding windows and resort solely to the memoization of sub-computations, which still requires time proportional to the size of the whole data rather (albeit with a small constant) than the change itself.

In this paper, we propose *self-adjusting contraction trees*, a set of data structures for incremental sliding window analytics, where the work performed by incremental updates is proportional to the size of the changes in the window rather than the whole data. Us-

ing these data structures only requires the programmer to devise a non-incremental version of the application code expressed using a conventional data-parallel programming model. We then guarantee an automatic and efficient update of the output as the window slides. Moreover, we make no restrictions on how the window slides, allowing it to shrink on one end and to grow on the other end arbitrarily. However, as we show, more restricted changes lead to simpler algorithms and more efficient updates.

Our approach for automatic incrementalization is based on the principles of self-adjusting computation [8, 9, 11], where the idea is to create a graph of data dependent sub-computations and propagate changes through this graph. Overall, our contributions include:

- **Self-adjusting contraction trees:** A set of self-adjusting data structures that are designed specifically for structuring different variants of sliding window computation as a (shallow) balanced dependence graph. These balanced graphs ensure that the work performed for incremental updates is proportional to the size of the changes in the window (the "delta") incurring only a logarithmic—rather than linear—dependency on the size of window (§3).
- **Split processing algorithms:** We introduce a *split processing model*, where the incremental computation is divided into a background pre-processing phase and a foreground processing phase. The background processing takes advantage of the predictability of input changes in sliding window analytics to pave the way for a more efficient foreground processing when the window slides (§4).
- **Query processing—multi-level trees:** We present an extension of the proposed data structures for multi-level workflows to support incremental data-flow query processing (§5).

We implemented self-adjusting contraction trees in a system called Slider, which extends Hadoop [1], and evaluated the effectiveness of the new data structures by applying Slider to a variety of micro-benchmarks and applications. Furthermore, we report on three real world use cases: *(i)* building an information propagation tree [38] for Twitter; *(ii)* monitoring Glasnost [26] measurement servers for detecting traffic differentiation by ISPs; and *(iii)* providing peer accountability in Akamai NetSession [12], a hybrid CDN architecture. Our experiments show that self-adjusting contraction trees can deliver significant performance gains for sliding window analytics without requiring any changes to the existing code.

The remainder of the paper is organized as follows. Section 2 presents an overview of the basic approach. The design of self-adjusting contraction trees is detailed in Sections 3, 4 and 5. The architecture of Slider is described in Section 6. Section 7 presents an experimental evaluation of Slider, and our experience with the case studies is reported in Section 8. Related work and conclusion are presented in Section 9 and Section 10, respectively.

2 Overview

Our primary goal is to design data structures for incremental sliding window analytics, so that the output is efficiently updated when the window slides. In addition, we want to do so transparently, without requiring the programmer to change any of the existing application code, which is written assuming non-incremental (batch model of) processing. In our prototype system called Slider, non-incremental computations are expressed under the MapReduce [24] programming model (or alternatively as Pig [20] programs), but the data structures can be plugged into other data-parallel programming models that allow for decomposing computations into associative sub-computations, such as Dryad [29] and Spark [32].

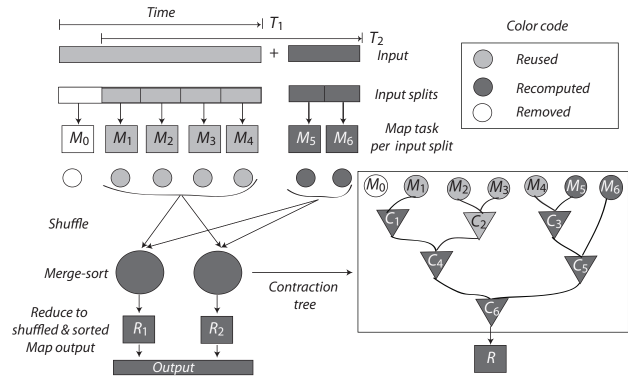


Figure 1: Strawman design and contraction phase

2.1 Strawman Design

The design of self-adjusting contraction trees is based on self-adjusting computation [8]. In this model, a computation is divided into sub-computations, and then a *dependence graph* is constructed to track control and data dependencies between all sub-computations. Thereafter, a *change propagation* algorithm is used to update the output by propagating the changes through the graph. The idea behind change propagation is to initially identify a set of sub-computations that directly depend on the changed data and re-execute them. The re-execution of a sub-computation may then modify other data, causing other data-dependent sub-computations to be re-executed. Change propagation terminates when all modified data and their dependent sub-computations complete. For change propagation to be efficient, it is important that *(a)* the computation is divided into small enough sub-computations; and *(b)* no long chain of dependencies exists between sub-computations. We call such computations ***stable*** because few of their sub-computations change when the overall input is modified by a small amount.

To apply the principles of self-adjusting computation to a data parallel programming model, the starting point for our strawman design is to make the dependence graph match the data-flow graph generated by the data-parallel model. The data-flow graph is represented by a DAG, where vertices are sub-computations or tasks, and directed edges correspond to the data dependencies between tasks. For incremental computation, the strawman approach memoizes the outputs of all tasks, and, given a set of input changes, it propagates these changes through the data-flow graph by reusing the memoized output for sub-computations unaffected by the changes, and re-computing the affected sub-computations.

In the case of MapReduce, vertices correspond to Map and Reduce tasks, and edges represent data transferred between tasks (as depicted in Figure 1). For sliding window analytics, new data items are appended at the end of the previous window and old data items are dropped from the beginning. To update the output incrementally, we launch a Map task for each new "split" (a partition of the input that is handled by a single Map task) and reuse the results of Map tasks operating on old but live data. We then feed the newly computed results together with the reused results to Reduce tasks to compute the final output.

This initial strawman design highlights an important limitation: even a small change to the input can preclude the reuse of all the work after the first level of nodes in the graph. This is because the second level nodes (i.e., the Reduce tasks) take as input all values for a given key ($< k_i >, < v_1, v_2, .., v_n >$). In the example shown in Figure 1, as the computation window slides from time T_1 to T_2,

Algorithm 1 Basic algorithm for sliding windows

```
Require: changes: △ ← (−δ1, +δ2)
 1: /* Process new input +δ2 by running Map tasks*/
 2: for i = T_end to T_end + (+δ2) do
 3:    M_i({k}) ← run_maptask(i);
 4: end for
 5: /* Propagate △ using contraction tree*/
 6: for all keys k do
 7:    /* Delete Map outputs for −δ1 */
 8:    for i = T_start to T_start − (−δ1) do
 9:       contraction_tree·delete(M_i(k));
10:    end for
11:    /* Insert Map outputs for +δ2*/
12:    for i = T_end to T_end + (+δ2) do
13:       contraction_tree·insert(M_i(k)));
14:    end for
15:    /* Perform change propagation*/
16:    contraction_tree·update(k);
17: end for
18: /* Adjust the window for the next incremental run*/
19: T_start ← T_start − (−δ1);
20: T_end ← T_end + (+δ2);
```

it invalidates the input of all Reduce tasks because of the removal of the M_0 output and the addition of new Map outputs (M_5 & M_6) to the window. To address this limitation, we refine the strawman design by organizing the second level nodes (corresponding to the Reduce phase) into a *contraction phase*, which is interposed between the Map and the Reduce phase, and makes use of the data structures proposed in this paper.

2.2 Adding the Contraction Phase

The idea behind the contraction phase is to break each Reduce task into smaller sub-computations, which are structured in a *contraction tree*, and then propagate changes through this tree.

We construct the contraction tree by breaking up the work done by the (potentially large) Reduce task into many applications of the Combiner function. Combiner functions [24] were originally designed to run at the Map task for saving bandwidth by doing a local reduction of the output of Map, but instead we use Combiners at the Reduce task to form the contraction tree. More specifically, we split the Reduce input into small partitions (as depicted in Figure 1), and apply the Combiner to pairs of partitions recursively in the form of a binary tree until we have a single partition left. Finally, we apply the Reduce function to the last Combiner, to get the final output. This requires Combiner functions to be associative, an assumption that is met by every Combiner function we have come across.

The final strawman design we obtain after adding the contraction phase is shown in Algorithm 1. As a starting point, changes ($\triangle$) in the input are specified by the user as the union of old items ($-\delta$) that are dropped and new items ($+\delta$) that are added to the window. Subsequently,

1. the items that are added to the window ($+\delta$) are handled by breaking them up into fixed-sized chunks called "splits", and launching a new Map task to handle each split (*line 1-4*);
2. the outputs from these new Map tasks along with the old splits that fall out from the sliding window ($-\delta$) are then fed to the contraction phase instead of the Reduce task *for each emitted key k* (*line 5-20*);
3. finally, the computation time window is adjusted for the next incremental run (*line 18-20*).

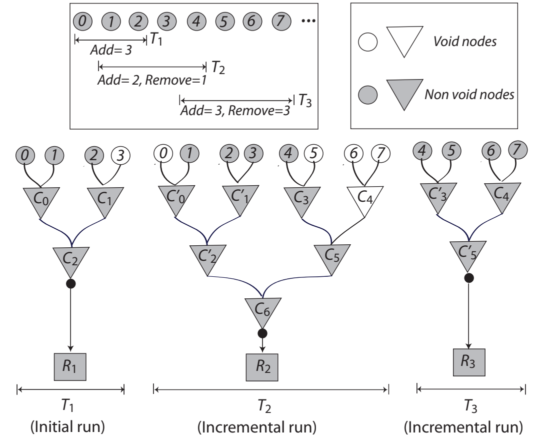


Figure 2: Example of folding contraction tree

The remainder of the paper presents a set of novel data structures, called *self-adjusting contraction trees*, that replace the simple binary tree in the strawman design of the contraction phase. The goal of these data structures is to ensure that the path from the newly added and dropped inputs to the root has a low depth, and that as many unaffected sub-computations as possible are outside that path. Furthermore, these data structures must perform a form of rebalancing after each run, i.e., a change in the sliding window not only triggers an update to the output but also to the structure of the contraction tree, to ensure that the desirable properties of our data structures hold for subsequent runs.

3 Self-Adjusting Contraction Trees

In this section, we present the general case data structures for incremental sliding window analytics. When describing our algorithms, we distinguish between two modes of running: an *initial run* and an *incremental run*. The *initial run* assumes all input data items are new and constructs the self-adjusting contraction tree from scratch. The *incremental run* takes advantage of the constructed tree to incrementally update the output.

3.1 Folding Contraction Tree

Our first data structure, called a *self-adjusting folding tree*, permits shrinking and extending the data window arbitrarily, i.e., supports *variable-width window* slides. The goal of this data structure is to maintain a small height for the tree, since this height determines the minimum number of Combiner functions that need to be recomputed when a single input changes.

Initial run. Given the outputs of the Map phase consisting of M tasks, we construct a folding tree of height $\lceil \log_2 M \rceil$ and pair each leaf with the output of a Map task, such that all Map tasks are mapped to a contiguous subset of the leaves with no unpaired leaves in between. The leaf nodes that cannot be filled in the complete binary tree (adding up to $2^{height} - M$ nodes) are marked as void nodes; these nodes will be occupied by future Map tasks. To compute the output, we apply Reduce to the root of the folding tree.

Figure 2 illustrates an example. At time T_1, we construct a complete binary tree of height two, where the leaves are the Map outputs of $\{0,1,2\}$, and with an additional void node to make the number of leaves a power of two. We then apply combiners $\{C_0, C_1, C_2\}$ to pairs of nodes to form a binary tree.

Incremental run. When the window slides, we want to keep the folding tree balanced, meaning that the height of the tree should be

roughly be equal to logarithmic to the current window size ($H = \lceil \log_2 M \rceil$), which is the minimum possible height. The basic idea is to assign the outputs of new Map invocations to the void leaf nodes on the right hand side of the tree, and mark the leaf nodes on the left hand side corresponding to Map tasks that were dropped from the window as void. Adding new nodes on the right hand side may require a change in the tree size, when the new Map outputs exceed the number of void nodes. Conversely, dropping nodes from the left hand side can cause the entire left half of the tree to contain void leaves. These are the two cases that lead to a change in the height of the tree, and they are handled by folding and unfolding units of *complete (sub-)trees*, i.e., increasing or decreasing the tree height by one, while maintaining a complete tree.

In particular, when inserting items, we first try to fill up the void nodes to the right of the non-void leaves that still have not been used in the previous run. If all void nodes are used, a new complete contraction tree is created, whose size is equal to the current tree, and we merge the two trees. This increases the height of the tree by one. When removing items, we always attempt to reduce the tree height in order to make incremental processing more efficient by checking if the entire left half of the leave nodes are void. If so, we discard half of the tree by promoting the right hand child of the root node to become the new root.

Figure 2 shows a set of example incremental runs for this algorithm. At time T_2, two Map outputs (nodes 3 & 4) are inserted, causing the tree to expand to accommodate node 4 by constructing another subtree of height two and joining the new subtree with the previous tree, which increases the height to three. Conversely, at time T_3 the removal of three Map outputs (nodes 1, 2, & 3) causes the tree height to decrease from three to two because all leaf nodes in the left half of the tree are void.

3.2 Randomized Folding Tree

The general case algorithm performs quite well in the normal case when the size of the window does not change drastically. However, the fact that tree expansion and contraction is done by doubling or halving the tree size can lead to some corner cases where the tree becomes imbalanced, meaning that its height is no longer $H = \lceil \log_2 M \rceil$. For example, if the window suddenly shrinks from a large value of M elements to $M' = 2$ elements, and the two remaining elements happen to be on different sides with respect to the root of the tree, then the algorithm ends up operating on a folding tree with height $\lceil log_2(2M+1) \rceil$ when the window is of size $M' << M$.

One way to address this problem is to perform an initial run whenever the size of the window is more than some desired constant factor (e.g., 8, 16) smaller than the number of leaves of the folding tree. On rebalancing, all void nodes are garbage collected and a freshly balanced folding tree is constructed ($H = \lceil log_2(M') \rceil$) similar to the initial run. This strategy is attractive for workloads where large variations in the window size are rare. Otherwise, frequently performing the initial run for rebalancing can be inefficient.

For the case with frequent changes in the window size, we designed a randomized algorithm for rebalancing the folding tree. This algorithm is very similar to the one adopted in the design of *skip lists* [36], and therefore inherits its analytical properties. The idea is to group nodes at each level probabilistically instead of folding/unfolding complete binary trees. In particular, each node forms a group boundary with a probability $p = 1/2$. In the expected case, and by analogy to the skip list data structure, the average height of the tree is $H = \lceil log_2(current_window_size) \rceil$.

Figure 3 shows an example of a randomized folding tree with 4 levels for 16 input leaf nodes. The tree is constructed by combining nodes into groups, starting from left to right, where for each node a

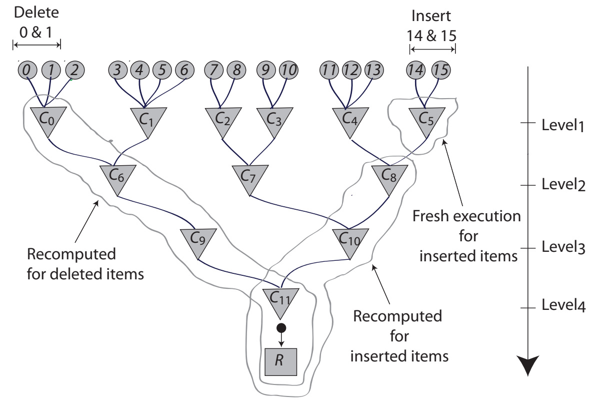


Figure 3: Example of randomized folding tree

coin toss decides whether to form a group boundary: with probability $p = 1/2$, a node either joins the previous group or creates a new group. In the example, leaf nodes $0, 1, 2$ join the same group C_0, and leaf node 3 creates a new group C_1 which is joined by nodes $4, 5, 6$. This process is repeated at all the levels. When nodes are deleted, all nodes on paths from the deleted nodes to the root are recomputed. In the example, after nodes 0 and 1 are deleted, node C_0, C_6, C_9, C_{11} are recomputed. Similarly, the newly inserted items are grouped probabilistically at all the levels, and then the merged nodes (combination of new and old nodes) are re-computed.

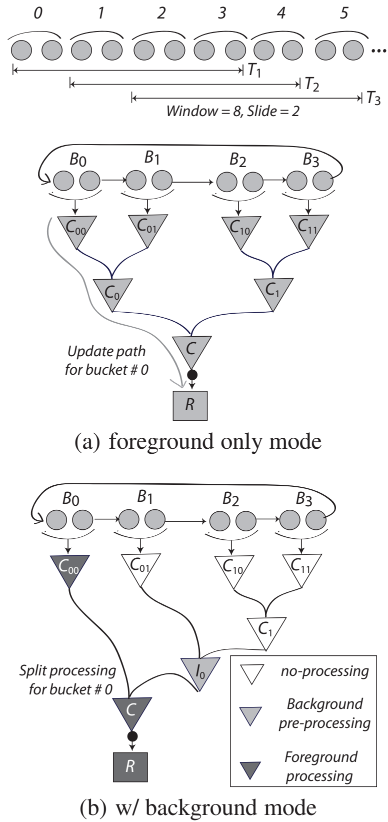
4 Split Processing Algorithms

We now consider two special cases of sliding windows, where, in addition to offering specialized data structures, we also introduce a split processing optimization. In the first special case, the window can be extended on one end and reduced on the other, as long as the size of the window remains the same (§4.1). (This is also known as *fixed-width window processing*.) In the second case, the window is only extended monotonically on one end by append operations (§4.2). (This is also known as *bulk-appended data processing*.)

In both these cases, since we know more in advance about the type of change that is going to take place in the next run, we leverage this fact for improving the responsiveness of incremental updates by preparing for the incremental run before it starts. More precisely, we split the change propagation algorithm into two parts: a *foreground processing* and a *background pre-processing*. The foreground processing takes place right after the update to the computation window, and minimizes the processing time by combining new data with a pre-computed intermediate result. The background pre-processing takes place after the result is produced and returned, paving the way for an efficient foreground processing by pre-computing an intermediate result that will be used in the next incremental update. The background pre-processing step is optional and we envision performing it on a best-effort basis and bypassing it if there are no spare cycles in the cluster.

4.1 Rotating Contraction Trees

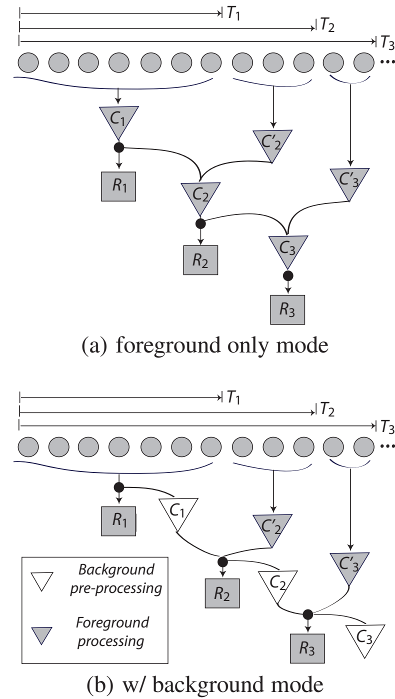
In *fixed-width* sliding window computations, new data is appended at the end, while the same amount of old data is dropped from the beginning of the window, i.e., w new splits (each processed by a new Map task) are appended and w old splits are removed. To perform such computations efficiently, we use *rotating contraction trees* (depicted in Figure 4). Here, w splits are grouped using the combiner function to form what we call a bucket. Then, we form a balanced binary contraction tree where the leaves are the buckets.



(a) foreground only mode

(b) w/ background mode

Figure 4: Example of rotating contraction trees



(a) foreground only mode

(b) w/ background mode

Figure 5: Example of coalescing contraction trees

Since the number of buckets remains constant when the window slides, we just need to rotate over the leaves in a round-robin fashion, replacing the oldest bucket with the newly produced one.

Initial run. In this case, the steady state of incremental runs is only reached when the window fills up. As such, we need to consider the sequence of initial runs during which no buckets are dropped. At each of these runs, we combine the w newly produced Map outputs to produce a new bucket. By the time the first window fills, we construct the contraction tree by combining all bucket outputs in pairs hierarchically, to form a balanced binary tree of height $\lceil log_2(N) \rceil$, where N is the total number of buckets in a window. Figure 4(a) shows an example with $w = 2$ and $N = 4$. At T_1, 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 task.

Incremental run. We organize the leaf nodes of the contraction tree as a circular list. When w new splits arrive and w old splits are removed from the data set, we replace the oldest bucket with the new bucket and update the output by recomputing the path affected by the new bucket. Figure 4(a) shows an example. At T_2 the new bucket 4 replaces the oldest bucket 0. This triggers a propagation of this change all the way to the root, where each step combines a memoized combiner output with a newly produced combiner output. In this example, we reuse the memoized outputs of combiners C_{01}, and C_1. In total, this requires recomputing a number of combiners that is equal to $log(N)$. The rotation of buckets requires commutativity in addition to the associativity of the combiner invocations. Both properties were held by Combiner functions in the applications we analyzed.

Background pre-processing. As explained before, this background step anticipates part of the processing since in this case we can predict the window change that will take place. In particular, in this case we know exactly what are the subtrees of the next incremental run whose outputs will be reused – these are the subtrees that fall outside the path from the next bucket to be replaced to the root. We take advantage of this by pre-combining all the combiner outputs that are at the root of those subtrees. For example, in Figure 4(b), we can pre-compute the combiner output I_0 by combining C_{01} and C_1 along the update path of bucket 0 in the background. This way, the incremental run only needs to invoke the Reduce task with the output of this pre-computed Combiner invocation (I_0) and the outputs of the newly run Map tasks.

4.2 Coalescing Contraction Trees

In the *append only* variant, the window grows monotonically as the new inputs are appended at the end of the current window, i.e., old data is never dropped. For this kind of workflow we designed a data structure called a *coalescing contraction tree* (depicted in Figure 5).

Initial run. The first time input data is added, a single-level contraction tree is constructed by executing the Combiner function (C_1 in Figure 5(a)) for all Map outputs. The output of this combiner is then used as input to the Reduce task, which produces the final output (R_1 in Figure 5(a)).

Incremental run. The outputs of the new Map tasks (C'_2 in Figure 5(a)) are combined, and the result is combined with the output of the contraction tree from the previous run to form a new contraction tree (C_2 combines the outputs of C_1 and C'_2). The output of the root of this new tree is then provided to a Reduce task (R_2 in the example), which produces the new output.

Background pre-processing. In the foreground processing step (see Figure 5(b)) the new output is computed directly by invoking

the Reduce task on the root of the old contraction tree and on the output of a Combiner invocation on the new Map inputs. In the background pre-processing phase, we prepare for the next incremental run by forming a new root of the contraction tree to be used with the next new input data. This is done by combining the root of the old tree with the output of the previous Combiner invocation on the new Map inputs. Figure 5(b) depicts an example for background pre-processing. 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, which aggregates the outputs of the newly run Map tasks (C_2'). In the background, we run the new combiner that will be used in the next incremental run (C_2), using the same inputs as the Reduce task, to anticipate the processing that will be necessary in the next run.

5 Query Processing: Multi-Level Trees

We next present an extension of self-adjusting contraction trees to integrate them with tools that support declarative data-flow query languages, such as Pig [20] or DryadLINQ [29]. These languages have gained popularity in the context of large-scale data analysis due to the ease of programming using their high-level primitives. To support these systems, we leverage the observation that programs written in these query languages are compiled to a series of pipelined stages where each stage corresponds to a program in a traditional data-parallel model (such as MapReduce or Dryad), for which we already have incremental processing support.

In particular, our query processing interface is based on Pig [20]. Pig consists of a high-level language (called Pig-Latin) similar to SQL, and a compiler that translates Pig programs to a workflow of multiple pipelined MapReduce jobs. Since our approach handles MapReduce programs transparently, each stage resulting from this compilation can run incrementally by leveraging contraction trees. A challenge, however, is that not all the stages in this pipeline are amenable to a sliding window incremental computation. In particular, after the first stage MapReduce job that processes the input from the sliding window, changes to the input of subsequent stages could be at arbitrary positions instead of the window ends. Thus, we adapt the strategy we employ at different stages as follows: (1) in the first stage, we use the appropriate self-adjusting contraction tree that corresponds to the desired type of window change; and, (2) from the second stage onwards in the pipeline, we use the strawman contraction tree (§2) to detect and propagate changes.

6 Slider Architecture & Implementation

We implemented self-adjusting contraction trees in a system called Slider, which is based on Hadoop-0.20.2. Our data structures are implemented by inserting an additional Contraction phase between the shuffle stage and the sort stage. To prevent unnecessary data movement in the cluster, the new Contraction phase runs on the same machine as the Reduce task that will subsequently process the data. An overview of the implementation is depicted in Figure 6. We next present some of its key components.

In-memory distributed cache. The implementation includes an in-memory distributed data caching layer to provide fast access to memoized results. The use of in-memory caching is motivated by two observations: first, the number of sub-computations that need to be memoized is limited by the size of the sliding window; second, main memory is generally underutilized in data-centric computing, thus creating an opportunity for reusing this resource [13]. We designed a simple distributed caching service that memoizes the outputs of sub-computations. The distributed cache is coordinated by a master node (in our case, the namenode of Hadoop), which maintains an index to locate the data items.

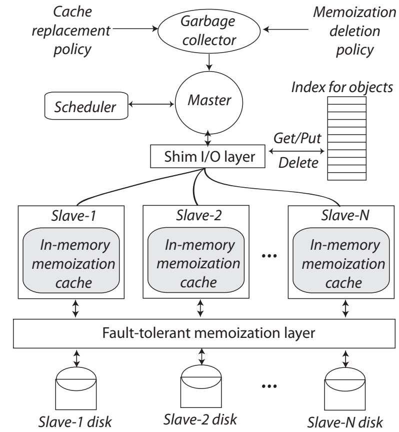


Figure 6: Slider architecture

Fault-tolerance. Storing memoized results in the in-memory data cache is beneficial for performance, but it can lead to reduced memoization effectiveness when machines fail, as the loss of memoized results will trigger otherwise unnecessary recomputations. To avoid this situation, we built a fault-tolerant memoization layer, which, in addition to storing memoized data in the in-memory cache, creates two replicas of this data in persistent storage. The replication is transparently handled by a shim I/O layer that provides low-latency access to the in-memory cache when possible and falls back to the persistent copies when necessary.

Garbage collection. To ensure that the storage requirements remain bounded, we developed a garbage collector (implemented at the master node) that manages the space used by the memoization layer. The garbage collector can either automatically free the storage occupied by data items that fall out of the current window, or have a more aggressive user-defined policy.

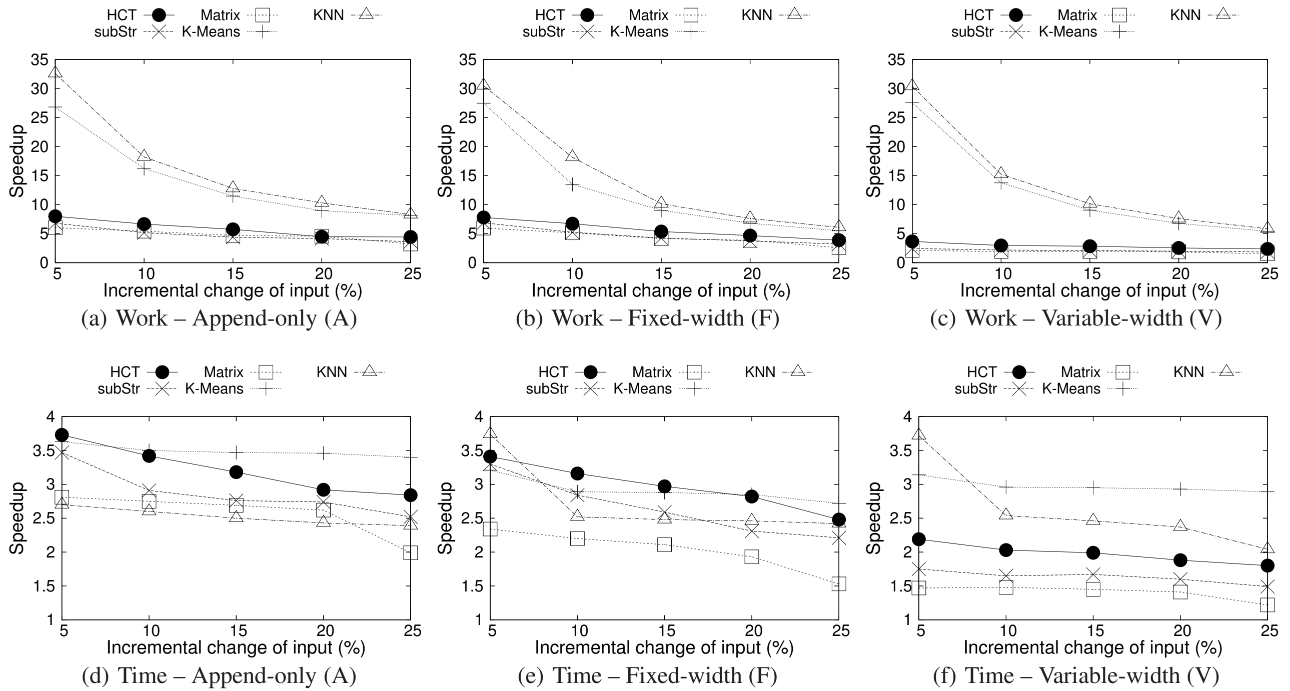
Memoization-aware scheduling. The original Hadoop scheduler takes into account the input data locality *only* when scheduling Map tasks, but chooses the first available machine to run a pending Reduce task (without considering any data locality). Slider modifies the original scheduler from Hadoop, based on previous work in data-locality scheduling [10], to schedule Reduce tasks where the previously run objects are memoized.

Straggler mitigation for incremental computation. A limitation of a *strict* memoization-aware scheduling policy is that it can become inefficient in the presence of straggler tasks, i.e., tasks that are slowed down, e.g., due to a high load on the machine where they execute [42]. This is because the scheduler can be left waiting for one of these tasks to complete in order to schedule a new task on its preferred location. To overcome the straggler effect while still exploiting the locality of memoized data, we designed a simple hybrid scheduling scheme that first tries to exploit the locality of memoized data, and, when the execution of a node is detected to be slow, dynamically migrates tasks from the slow node to another node. In the case of a migration, the memoized data is fetched over the network by the new node where the task is scheduled.

7 Evaluation

Our evaluation answers the following questions:

- How does the perfomance of Slider compare to recomputing over the entire window of data and with the memoization-based strawman approach? (§ 7.2)



(a) Work – Append-only (A) (b) Work – Fixed-width (F) (c) Work – Variable-width (V)

(d) Time – Append-only (A) (e) Time – Fixed-width (F) (f) Time – Variable-width (V)

Figure 7: Performance gains of Slider compared to recomputing from scratch

- How effective are the optimizations we propose in improving the performance of Slider? (§ 7.3)
- What are the overheads imposed during a fresh run of an application? (§ 7.4)

7.1 Experimental Setup

Applications and dataset. Our micro-benchmarks span five Map-Reduce applications that implement typical data analysis tasks. Two are compute-intensive applications: K-means clustering **(K-Means)**, and *K*-nearest neighbors **(KNN)**. As input to these tasks we use synthetically generated data by randomly selecting points from a 50-dimensional unit cube. The remaining three are data-intensive applications: a histogram-based computation **(HCT)**, a co-occurrence matrix computation **(Matrix)**, and a string computation extracting frequently occurring sub-strings **(subStr)**. As input we use a publicly available dataset of Wikipedia [7].

Cluster setup. Our experiments run on a cluster of 25 machines. We configured Hadoop to run the namenode and the job tracker 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.

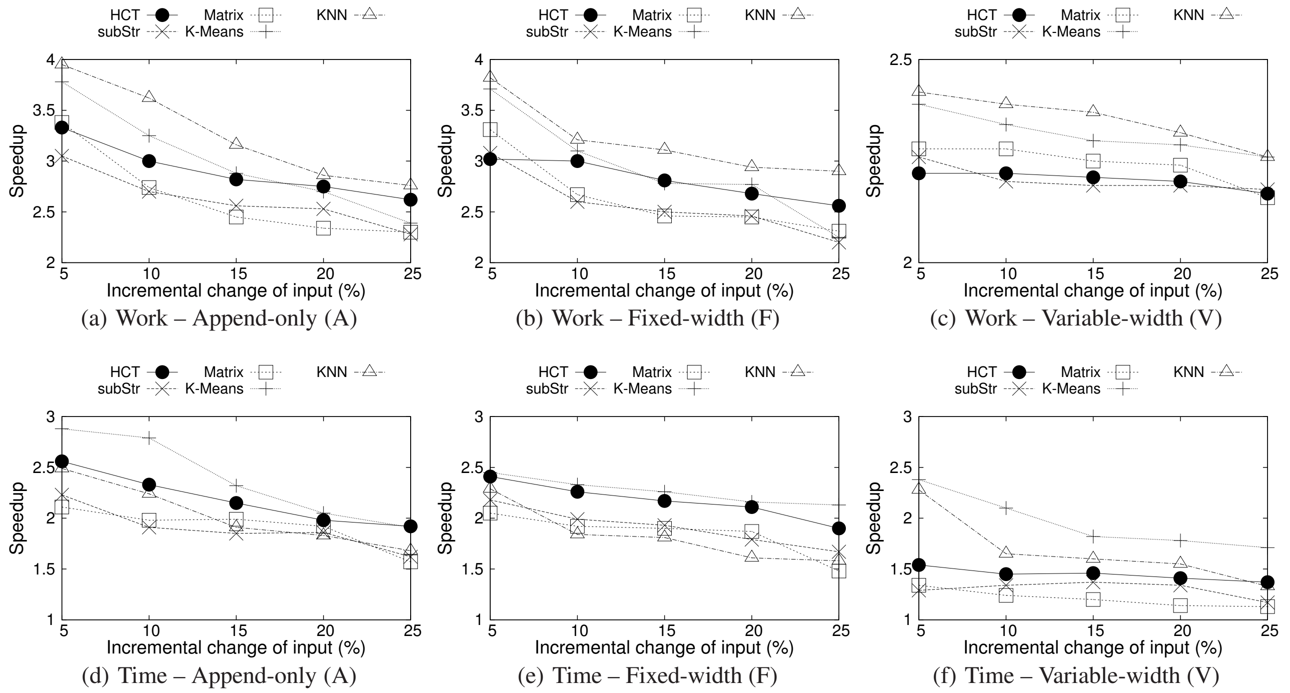
Measurements. We consider two types of measures: *work* and *run-time* (or *time*). Work refers to the total amount of computation performed by all tasks (Map, contraction, and Reduce) and is measured as the sum of the active time for all the tasks. Time refers to the (end-to-end) total amount of running time to complete the job.

Methodology. To assess the effectiveness of Slider, we measured the work and run-time of each micro-benchmark for different dynamic update scenarios, i.e., with different amounts of modified inputs, ranging from 5% 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-width and variable-width sliding window cases, the window is moved such that *p*% of the input buckets are dropped from the window's beginning, and replaced with the same number of new buckets containing new content appended to the window's end.

7.2 Performance Gains

Speedups w.r.t. recomputing from scratch. We first present the performance gains of Slider in comparison with recomputing from scratch. For the comparison, we compared the work and run-time of Slider to an unmodified Hadoop implementation. Figure 7 shows that the gains for compute-intensive applications (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 depending on the application. Speedups for data-intensive applications (HCT, Matrix, and subStr) are between 1.5-fold and 8-fold. Despite these also being positive results, the speedup figures are lower than in the case of applications 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 the outputs of sub-computations, even if they were memoized. The performance gains for variable-width sliding windows are lower than for the append-only and fixed-width window cases because updates require rebalancing the tree, and thus incur a higher overhead.

Performance breakdown. Figure 9(a) and Figure 9(b) show the normalized execution time breakdown in the incremental run with 5% and 25% changes in the input, respectively. The Map and Reduce contributions to the total time for the baseline vanilla Hadoop are shown in the bar labelled "H". The "H" bar breakdown shows that the compute-intensive applications (Kmeans and KNN) perform around 98% of the work in the Map phase, whereas the other



(a) Work – Append-only (A) (b) Work – Fixed-width (F) (c) Work – Variable-width (V)

(d) Time – Append-only (A) (e) Time – Fixed-width (F) (f) Time – Variable-width (V)

Figure 8: Performance gains of Slider compared to the memoization based approach (the strawman design)

applications (HCT, Matrix, and SubStr) perform roughly the same amount of work in each phase.

The same figures also show the breakdown for all three modes of operation ("A" for Append, "F" for Fixed-width, and "V" for Variable-width windowing), where the Slider-Map and Slider-contraction + Reduce portions in these bars represent the execution time computed as a percentage of the baseline Hadoop-Map and Hadoop-Reduce (H) times, respectively. In other words, the percentage execution time for Slider-Map is normalized to Hadoop-Map, while the percentage execution time for Slider-contraction + Reduce is normalized to Hadoop-Reduce.

For the Map phase of Slider, the breakdown shows that the percentage of Map work (compared to the non-incremental baseline) in the incremental run is proportional to the input change, as expected. In contrast, the work done by the Reduce phase is less affected by the amount of input change. In particular, the contraction + Reduce phase execution averages 31% of the baseline Reduce execution time (min: 18.39%, max: 59.52%) for 5% change, and averaged 43% (min: 26.39%, max: 80.95%) for 25% change across all three modes of operation.

Speedups w.r.t. memoization (strawman approach). Figure 8 presents the work and time speedup of Slider w.r.t. the memoization-based strawman approach (as presented in Section 2). The processing performed in the Map phase is the same in both approaches, so the difference lies only in the use of self-adjusting contraction trees instead of the strawman contraction tree. The work gains range from $2X$ to $4X$ and time gains range from $1.3X$ to $3.7X$ for different modes of operation with changes ranging from 25% to 5% of the input size. The work speedups for the compute-intensive applications (Kmeans and KNN) decrease faster than other applications as the input change increases because most performance gains were due to savings in the Map phase. Overall, although less pronounced than in the comparison to recomputing from scratch, the results show considerable speedups due to the data structures that are specific to each type of sliding window processing, when compared to the strawman approach.

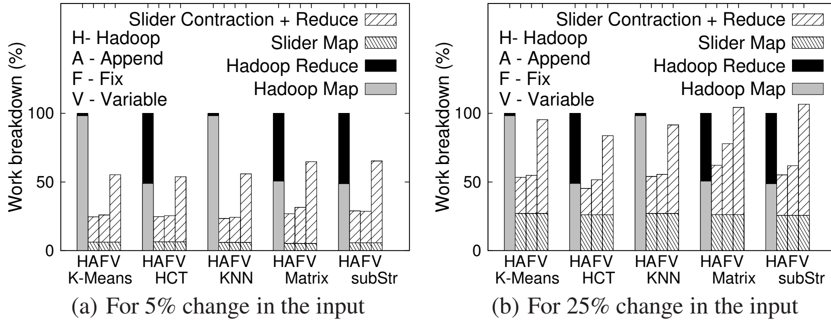
7.3 Effectiveness of Optimizations

We now evaluate the effectiveness of the individual optimizations in improving the overall performance.

Split processing. Slider is designed to take advantage of the predictability of future updates by splitting the work between background and foreground processing. To evaluate the effectiveness in terms of latency savings from splitting the execution, we compared the cost of executing with and without it, for both the append-only and the fixed-width window categories. Figures 11(a) and 11(b) show the time required for background preprocessing and foreground processing, normalized to the total time (total update time = 1) for processing the update without any split processing. Figure 11(a) shows this cost when a new input with 5% of the original input size is appended, for different benchmarking applications, whereas Figure 11(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, we are on average able to perform foreground updates up to 25%-40% faster, while offloading around 36%-60% of the work to background pre-processing.

The results also show that the sum of the cost of background preprocessing and foreground processing exceeds the normal update time (total update time = 1) because of the extra merge operation performed in the split processing model. Our results show that the additional CPU usage for the append-only case is in the range of 1% to 23%, and 6% to 36% for the fixed-window processing.

Scheduler modification. Table 1 shows the effectiveness of our hybrid scheduler in improving the performance of Slider compared to the scheduler of Hadoop (normalized to 1 as the baseline). The new scheduler saves on average 23% of run-time for data-intensive applications, and 12% of time for compute intensive applications.



(a) For 5% change in the input (b) For 25% change in the input

Figure 9: Performance breakdown for work

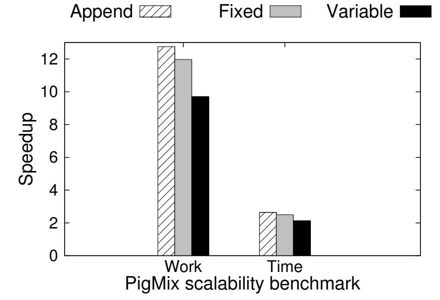
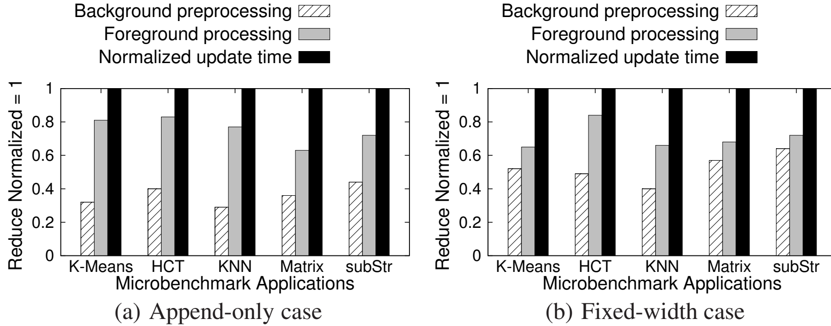


Figure 10: Query processing



(a) Append-only case (b) Fixed-width case

Figure 11: Effectiveness of Split processing

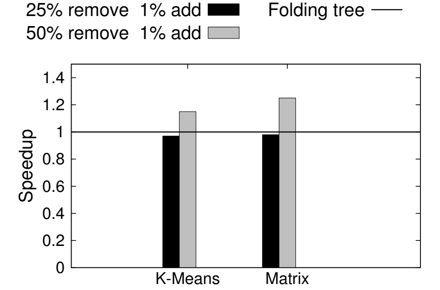


Figure 12: Randomized folding tree

K-Means	HCT	KNN	Matrix	subStr
0.94	0.72	0.82	0.83	0.76

Table 1: Normalized run-time for Slider scheduler with respect to Hadoop scheduler (run-time = 1).

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

Table 2: Reduction in the time for reading the memoized state with in-memory caching.

These results show that scheduling tasks on the machines where memoized results are stored lead to improved performance.

Data-flow query interface. To demonstrate the potential of incremental query-based sliding-window computations, we evaluate Slider using the PigMix [2] benchmark, which generates a long pipeline of MapReduce jobs derived from Pig Latin query scripts. We ran the benchmark in our three modes of operation with changes to 5% of its input. Figure 10 shows the resulting run-time and work speedups. As expected, the results are in line with the previous evaluation, since ultimately the queries are compiled to a set of MapReduce analyses. We observe an average speedup of 2.5X and 11X for time and work, respectively.

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 an additional cost of fetching the data from the disk or network. Table 2 shows reduction in the time for reading the memoized state with in-memory caching for fixed-width windowing. This shows that we can achieve 50% to 68% savings in the read time by using the in-memory caching.

Randomized folding tree. To evaluate the effectiveness of the randomized folding tree, we compared the gains of the randomized version with the normal folding tree (see Figure 12). We compare the performance for two update scenarios: reducing the window size by two different amounts (25% and 50%) and, in both cases, performing a small update adding 1% of new items to the window. We report work speedups for two applications (K-Means and Matrix), representing both compute and data-intensive applications.

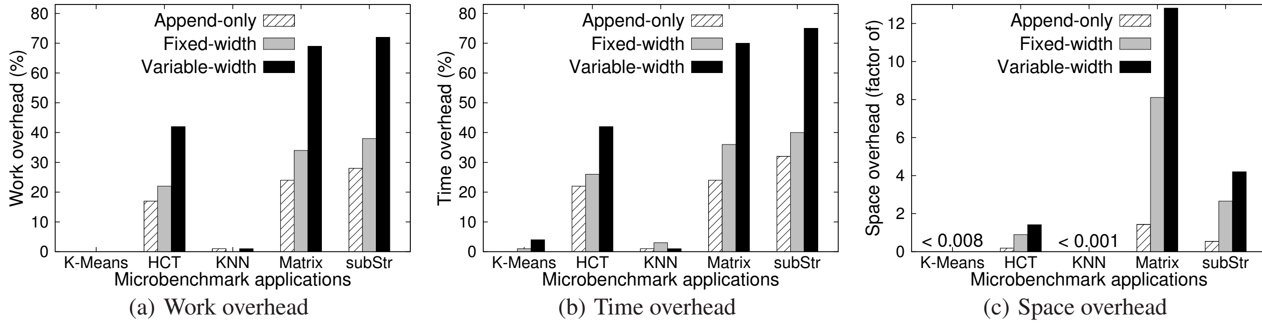
The experiments show that a large imbalance (of 50% removals to 1% additions) is required for the randomized data structure to be beneficial. In this case, the randomized version leads to a performance improvement ranging from 15% to 22%. This is due to the fact that decreasing the window size by half also reduces the height of the randomized folding tree by one when compared to the original version (leading to more efficient updates). In contrast, with 25% removals to the same 1% additions the standard folding tree still operates at the same height as the randomized folding tree, which leads to a similar, but slightly better performance compared to the randomized structure.

7.4 Overheads

Slider adds two types of overhead. First, the *performance overhead* for the initial run (a one time cost only). Second, the *space overhead* for memoizing intermediate results.

Performance overheads. Figure 13(a) and Figure 13(b) show the work and time overheads for the initial run, respectively. Compute-intensive applications (`K-means` & `KNN`) show low overhead as their run-time is dominated by the actual processing time and are less affected by the overhead of storing intermediate nodes of the tree. For data-intensive applications, the run-time overhead is higher because of the I/O costs for memoizing the intermediate results.

The overheads for the variable-width variant are higher than those



(a) Work overhead (b) Time overhead (c) Space overhead

Figure 13: Overheads of Slider for the initial run

for fixed-width computations, and significantly higher than the append case. This additional overhead comes from having more levels in the corresponding self-adjusting contraction tree.

Space overhead. Figure 13(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 application. Matrix has the highest space overhead of 12X, while K-Means and KNN have almost no space overhead.

The results show that the overheads, despite being visible, are a one-time cost likely to be worth paying in workloads where the initial run is followed by many incremental runs that reap the benefit of incremental computations. Furthermore, the garbage collection policy can further limit the space overheads.

8 Real-world Case Studies

We used Slider to evaluate three real-world case studies covering all three operation modes for sliding windows. Our case studies include: *(i)* building an information propagation tree [38] for Twitter for append-only windowing; *(ii)* monitoring Glasnost [26] measurement servers for detecting ISPs traffic differentiation for fixed-width windowing; and *(iii)* providing peer accountability in Akamai NetSession [12], a hybrid CDN for variable-width windowing.

8.1 Information Propagation in Twitter

Analyzing information propagation in online social networks is an active area of research. We used Slider to analyze how web links are spread in Twitter, repeating an analysis done in [38].

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

Dataset. We used the complete Twitter snapshot data from [38], which comprises 54 million users, 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 into 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 cumulative change of 5% was performed with every new append.

Performance gains. We present the performance gains of incrementally building the information propagation tree using Slider in

Time interval	Mar'06 Jun'09	Jul'09 1-7	Jul'09 8-14	Jul'09 15-21	Jul'09 22-28
Tweets (M)	1464.3	74.2	81.5	79.4	85.6
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

Table 4. The speedups are almost constant for the four time intervals, at about $8X$ for run-time and about $14X$ for work. The run-time overhead for computing over the initial interval is 22%.

8.2 Monitoring of a Networked System

Glasnost [26] is a system that enables users to detect whether their broadband traffic is shaped by their ISP. The Glasnost system tries to direct users to a nearby measurement server. Slider enabled us to evaluate the effectiveness of this server selection.

Implementation. For each Glasnost test run, a packet trace of the measurement traffic between the 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 across all users that were directed to this server.

Dataset. For this analysis, we used the data collected by one 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 computed the mean 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.

Performance gains. We measured both work and time speedups as shown in Table 3. The results show that we get an average speedup on the order of $2.5X$, with small overheads of less than 5%.

8.3 Accountability in Hybrid CDNs

Content distribution networks (CDN) operators like Akamai recently started deploying hybrid CDNs, which employ P2P technology to add end user nodes to the distribution network, thereby cutting costs as fewer servers need to be deployed. However, this also raises questions about the integrity of the answers that are provided by these untrusted clients [12].

Implementation. Aditya et al. [12] presented a design of a hybrid CDN that employs a tamper-evident log to provide client accountability. This log is uploaded to a set of servers that need to audit

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

% clients online to upload logs	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: Akamai NetSession data analysis summary

the log periodically using techniques based on PeerReview. 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 to upload their logs to the central infrastructure in the hybrid CDN.

Dataset. To evaluate the effectiveness of Slider, we used a synthetic dataset generated using trace parameters available from 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 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.

Performance gains. 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 a fraction of clients uploading the log in the final week that varied from 75% to 100%. Similarly, the run-time speedups are between $1.5X$ and $2X$.

9 Related Work

We compare our work to two major classes of systems that are suitable for sliding window analytics: trigger-based windowing systems, and batch-based windowing systems. We conclude with a broader comparison to incremental computation mechanisms.

Trigger-based windowing systems. These systems follow *record-at-a-time* processing model, where every new data entry triggers a state change and possibly produces new results, while the application logic, known as a *standing query*, may run indefinitely. This query can be translated into a network with stateless and/or stateful nodes. A stateful node updates its internal state when processing incoming records, and emits new records based on that state. Examples of such systems include Storm [5], S4 [3], StreamInsight [31], Naiad [33], Percolator [34], Photon [14], and streaming databases [15]. Despite achieving low latency, these systems also raise challenges [41]:
(1) Fault-tolerance: To handle faults, these systems either rely on *replication with synchronization* protocols such as Flux [39] or Borealis's DPC [15], which have a high overhead, or on checkpointing *upstream backup* mechanisms, which have a high recovery time. In addition, neither fault tolerance approach handles stragglers.
(2) Semantics: In a trigger-based system, it can be difficult to reason about global state, as different nodes might be processing different updates at different times. This fact, coupled with faults, can lead to weaker semantics. For instance, S4 provides at most once semantics, and Storm [5] provides at-least-once semantics.
(3) Programming model: The *record-at-a-time* programming model in trigger-based systems requires the users to manage the intermediate state and wire the query network topology manually. Furthermore, programmers need to understand how to update the output of each node in the query network as its input evolves. The design of the update logic is further complicated by the weak semantics provided by the underlying platform. While supporting incremental computation for aggregate operations is straightforward, this can be very challenging for non-trivial computations like matrix operations or temporal joins [40, 6].

Batch-based windowing systems. These systems model sliding window analytics as a series of deterministic batch computations on small time intervals. Such systems have been implemented both on top of trigger-based systems (e.g., Trident [6] built over Storm [5] or TimeStream [40] built over StreamInsight [31]) and systems originally designed for batch processing (e.g., D-Streams [41] built over Spark [32] or MapReduce online [23] and NOVA [21] built over MapReduce [24]). These systems divide each application into a graph of short, deterministic tasks. This enables simple yet efficient fault recovery using recomputation and speculative execution to handle stragglers [42]. In terms of consistency, these systems trivially provide "exactly-once" semantics, as they yield the same output regardless of failures. Finally, the programming model is the same as the one used by traditional batch processing systems.

We build on this line of research, but we observe that these systems are not geared towards incremental sliding window computation. Most systems recompute over the entire window from scratch, even if there is overlap between two consecutive windows. The systems that allow for an incremental approach require an inverse function to exist [41], which may not be trivial to devise for complex computations. In this work, we address these limitations in batch-based windowing systems by proposing a transparent solution for incremental sliding window analytics.

Incremental Computation. While there exists some prior work on enabling incremental computations in batch processing systems, this work did not leverage the particular characteristics of sliding windows, among other important differences. In more detail, incremental computation in batched processing systems such as Incoop [17, 18], Haloop [19], Nectar [27], DryadInc [35] requires linear time in the size of the input data, even to process a small slide in the window. The reasons for this are twofold: Firstly, these systems assume that inputs of consecutive runs are stored in separate files and simply compute their `diffs` to identify the input changes. The change detection mechanism relies on techniques such as content-based chunking (as in Incoop using IncHDFS [18]), which requires performing linear work in the size of the input [16]. In contrast, sliding window computation provides diffs naturally, which can be leveraged to overcome the bottleneck of identifying changes. Second, and more importantly, these systems do not perform change propagation, relying instead on memoization to recover previously

computed results. Consequently, they require visiting all tasks in a computation even if the task is not affected by the modified data, i.e. the delta, thus requiring an overall linear time. In contrast, this paper proposes an approach that only requires time that is linear in the delta, and not the entire window, and we propose new techniques that are specific to sliding window computation.

10 Conclusions

In this paper, we presented self-adjusting contraction trees for incremental sliding window analytics. The idea behind our approach is to structure distributed data-parallel computations in the form of balanced trees that can perform updates in asymptotically sublinear time, thus much more efficiently than recomputing from scratch. We present several algorithms and data structures for supporting this type of computation, describe the design and implementation of Slider, a system that uses our algorithms, and present an extensive evaluation showing that Slider is effective on a broad range of applications. This shows that our approach provide significant benefit for sliding window analytics, without requiring the programmer to write the logic for handling updates.

Asymptotic analysis. The asymptotic efficiency analysis of self-adjusting contraction trees is available online [4].

Acknowledgements

We are thankful to Paarijaat Aditya, Marcel Dischinger, and Farshad Kooti for helping us with the evaluation of the case studies. We also thank Ashok Anand, Haibo Chen, Rose Hoberman, Kostas Kloudas, Dionysios Logothetis, Sue Moon, Malte Schwarzkopf, and the SysNets group members at MPI-SWS and MSR Cambridge for the valuable feedback on this work. The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC starting grant. Computing resources for this work were supported by an Amazon Web Services (AWS) Education Grant. The research of Umut Acar is partially supported by the European Research Council under grant number ERC-2012-StG-308246 and the National Science Foundation under grant number CCF-1320563.

References

[1] Apache Hadoop: http://hadoop.apache.org.
[2] Apache PigMix: http://wiki.apache.org/pig/PigMix.
[3] Apache S4: http://incubator.apache.org/s4.
[4] Asymptotic analysis of self-adjusting contraction trees: http://www.mpi-sws.org/~bhatotia/publications.
[5] Storm: http://storm-project.net.
[6] Trident: http://storm.incubator.apache.org/.
[7] Wikipedia dataset: http://wiki.dbpedia.org.
[8] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.
[9] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM TOPLAS*, 2009.
[10] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The Data Locality of Work Stealing. In *SPAA*, 2000.
[11] U. A. Acar, G. E. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Türkoğlu. Traceable data types for self-adjusting computation. In *ACM PLDI*, 2010.
[12] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable Client Accounting for P2P-Infrastructure Hybrids. In *NSDI*, 2012.
[13] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI*, 2012.
[14] Ananthanarayanan et al. Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. In *SIGMOD*, 2013.
[15] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD*, 2005.
[16] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *FAST*, 2012.
[17] P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *HotCloud*, 2011.
[18] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *SoCC*, 2011.
[19] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB*, 2010.
[20] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.
[21] C. Olston et al. Nova: Continuous Pig/Hadoop Workflows. In *SIGMOD*, 2011.
[22] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 1992.
[23] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *NSDI*, 2010.
[24] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
[25] C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*.
[26] 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.
[27] 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.
[28] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *SoCC*, 2010.
[29] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.
[30] D. Logothetis, C. Olston, B. Reed, K. Web, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, 2010.
[31] M. Ali et al. Microsoft CEP server and online behavioral targeting. In *VLDB*, 2009.
[32] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.
[33] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *SOSP*, 2013.
[34] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, 2010.
[35] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *HotCloud*, 2009.
[36] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *CACM*, 1990.
[37] G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *POPL*, 1993.
[38] T. Rodrigues, F. Benevenuto, M. Cha, K. Gummadi, and V. Almeida. On Word-of-Mouth Based Discovery of the Web. In *IMC*, 2011.
[39] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD*, 2004.
[40] Z. Qian et al. TimeStream: Reliable Stream Computation in the Cloud. In *EuroSys*, 2013.
[41] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*, 2013.
[42] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.