Incremental MapReduce Computations

Pramod Bhatotia[1], Alexander Wieder[2], Umut A. Acar[3], Rodrigo Rodrigues[4]

June 17, 2013

[1]Max Planck Institute for Software Systems (MPI-SWS)
[2]Max Planck Institute for Software Systems (MPI-SWS)
[3]Carnegie Mellon University and INRIA Paris-Rocquencourt
[4]CITI / Universidade Nova de Lisboa

Contents

0.1 Abstract

Distributed processing of large data sets is an area that received much attention from researchers and practitioners over the last few years. In this context, several proposals exist that leverage the observation that data sets evolve over time, and as such there is often a substantial overlap between the input to consecutive runs of a data processing job. This allows the programmers of these systems to devise an efficient logic to update the output upon an input change. However, most of these systems lack compatibility existing models and require the programmer to implement an application-specific dynamic algorithm, which increases algorithm and code complexity.

In this chapter, we describe our previous work on building a platform called Incoop, which allows for running MapReduce computations incrementally and transparently. Incoop detects changes between two files that are used as inputs to consecutive MapReduce jobs, and efficiently propagates those changes until the new output is produced. The design of Incoop is based on memoizing the results of previously run tasks, and reusing these results whenever possible. Doing this efficiently introduces several technical challenges that are overcome with novel concepts, such as a large-scale storage system that efficiently computes deltas between two inputs, a Contraction phase to break up the work of the Reduce phase, and an affinity-based scheduling algorithm. This chapter presents the motivation and design of Incoop, as well as a complete evaluation using several application benchmarks. Our results show significant performance improvements without changing a single line of application code.

0.2 Introduction

Distributed processing of large data sets has become an important task in the life of various companies and organizations, for whom data analysis is an important vehicle to improve the way they operate. This area has attracted a lot of attention from both researchers and practitioners over the last few years, particularly after the introduction of the MapReduce paradigm for large-scale parallel data processing [19].

A usual characteristic of the data sets that are provided as inputs to large-scale data processing jobs is that they do not vary dramatically over time. Instead, the same job is often invoked consecutively with small changes in this input from one run to the next. For instance, researchers have reported that the ratio between old and new data when processing consecutive web crawls may range from 10 to 1000X [28].

Motivated by this observation, there have been several proposals for large-scale *incremental* data processing systems, such as Percolator [33] or CBP [28], to name a few early and prominent examples. In these systems, the programmer is able to devise an incremental update handler, which can store state across successive runs, and contains the logic to update the output as the program is notified about input changes. While this approach allows for significant improvements when compared to the "single shot" approach, i.e., re-processing all the data each time that part of the input changes or that inputs are added and deleted, it also has the downside of requiring programmers to adopt a new programming model and API. This has two negative implications. First, there is the programming effort to port a large set of existing applications to the new programming model. Second, it is often difficult to devise the logic for incrementally updating the output as the input changes: research in the area of dynamic algorithms (i.e., algorithms to solve problems that are formulated in terms of deltas to their input) shows that such algorithms can be very complex, even in cases where the normal, non-incremental algorithm was easy to devise [16, 20].

In this chapter, we present an overview of our prior work on designing and building a system called Incoop for large-scale incremental computations [11]. Incoop extends the Hadoop open source implementation of the MapReduce paradigm to run unmodified MapReduce programs in an incremental way. The design and implementation of Incoop is inspired by recent advances on self-adjusting computation [3, 6, 5, 24, 15], which offers a solution to the problem of automatic incrementalization of programs, and draws on techniques developed in that line of work. The idea behind Incoop is to enable the programmer to incrementalize automatically existing MapReduce programs without the need to make any modifications to the code. To this end, Incoop records information about previously executed MapReduce tasks so that it can be reused in future MapReduce computations when possible.

The basic approach taken by Incoop consists of (1) splitting the computation into sub-computations, where the natural candidate for a sub-computation is a MapReduce task; (2) memoizing the inputs and outputs of each sub-computation; (3) in an incremental run, checking the inputs to a sub-computation and using the memoized output without rerunning the task when the input remains unchanged. Despite being a good starting point, this basic approach has several shortcomings that motivated us to introduce several technical innovations in Incoop, namely:

- **Incremental HDFS.** We introduce a file system called Inc-HDFS that provides a scalable way of identifying the deltas in the inputs of two consecutive job runs. This reuses an idea from the LBFS local file system [31], which is to avoid splitting the input into fixed-size chunks, and instead split it based on the contents such that small changes to the input keep most chunk boundaries. The new file system is able to achieve a large reuse of input chunks while maintaining compatibility with HDFS, which is the most common interface to provide the input to a job in Hadoop.

- **Contraction phase.** To avoid rerunning a large Reduce task when only a small subset of its input changes, we introduce a new phase in the MapReduce framework called the Contraction phase. This consists of breaking up the Reduce task into smaller sub-computations that form an inverted tree, such that, when a small portion of the input changes, only the path from the corresponding leaf to the root needs to be recomputed.

- **Memoization-aware scheduler.** We modify the scheduler of Hadoop to take advantage of the locality of memoized results. The new scheduler uses a work stealing strategy to decrease the amount of data movement across machines when reusing memoized outputs, while still allowing tasks to execute on machines that are available.

We present an overview of the design and implementation of Incoop, and we report experimental results using five MapReduce applications. The results from this evaluation show that we achieve significant performance gains, while paying a modest cost during the initial run or during runs that cannot take advantage of previously computed results.

The rest of this chapter is organized as follows. We first present an overview of Incoop

in Section 0.3. The system design is detailed in Sections 0.4, 0.5, and 0.6. We present the experimental evaluation in Section 0.7. Related work and conclusions are discussed in Section 0.8 and Section 0.9, respectively.

0.3 System overview

We present first a basic design that we use as a starting point, highlight the limitations of this basic design, the challenges in overcoming them, and briefly overview the main ideas behind Incoop, which addresses the limitations of the basic design. Our basic strategy is to adapt the principles of self-adjusting computation to the MapReduce paradigm, and in particular to Hadoop. We start with some background on self-adjusting computation.

0.3.1 Self-adjusting computation

Self-adjusting computation [3, 6, 5, 24, 15] offers a solution to the incremental-computation problem by enabling any computation to respond to changes in its data by efficiently recomputing only the subcomputations that are affected by the changes. To this end, a self-adjusting computation tracks dependencies between the inputs and outputs of subcomputations, and, in incremental runs, only rebuilds subcomputations affected (transitively) by modified inputs. To identify the affected subcomputations, the approach represents a computation as a dependency graph of subcomputations, where two sub-computations are data-dependent if one of them uses the output of the other as input and control-dependent if one takes place within the dynamic scope of another. Subcomputations are also memoized based on their inputs to enable reuse even if they are control-dependent on some affected subcomputation. Given the "delta", the modifications to the input, a *change-propagation algorithm* pushes the modifications through the dependency graph, rebuilding affected subcomputations, which it identifies based on both data and control dependencies. Before rebuilding a subcomputation, change propagation recovers subcomputations that can be reused, even partially, by using a computation memoization technique that remembers (and re-uses) not just input-output relationships but also the dependency graphs of memoized

subcomputations [5].

The efficiency of a self-adjusting-computations in responding to an input modification is determined by the stability of the computation. Informally speaking, we call a computation *stable* when the set of subcomputations performed on similar input data sets themselves are similar, i.e., many of the subcomputations are in fact the same and thus can be re-used. For a more precise definition of stability, we refer the interested reader to the previous work [3, 27]. One way to ensure stability is to make sure that 1) the computation is divided into small subcomputations, and 2) no long chain of dependencies exists between computations. Since MapReduce framework is naturally parallel, it naturally has short dependency chains at the granularity of tasks. As we will see, however, it can yield unstable computations, because a small change to the input can cause the input for many MapReduce tasks to change, and because reduce tasks can be large.

For the sake of simplicity in design and implementation, Incoop, does not construct the dependency graph of subcomputations explicitly, and thus does not perform change propagation on the dependency graph. Instead, the graph is recorded implicitly by memoizing subcomputations—MapReduce tasks—and change propagation is performed by re-visiting all subcomputations and reusing those that can be reused via memoization. While this approach simplifies the design and the implementation, it can yield asymptotically suboptimal performance, because it requires touching all subcomputations (for the purposes of memoization and reuse) even if they may not be affected by the input modifications. Since, however, subcomputations (tasks) are relatively large, the cost of re-using an unaffected subcomputation is small compared to rebuilding it. The approach therefore can perform well in practice (Section 0.7).

0.3.2 Basic design

Our goal is to design a system for large-scale incremental data processing that is able to leverage the performance benefits of incremental computation, while also being transparent, meaning that it does not require changes to existing programs. In particular, we consider

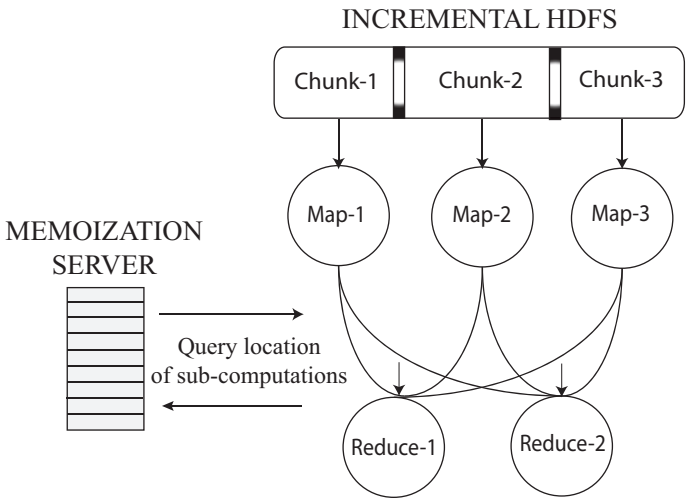


Figure 1: Basic design of Incoop.

MapReduce programs for distributed large-scale data processing. We assume the reader is familiar with the MapReduce paradigm, and refer to prior publications on the subject for more information [19].

To achieve this goal, we apply the principles of self-adjusting computation to the MapReduce paradigm. To this end, we first need to decide what forms a sub-computation. The natural candidate in our system is to use MapReduce tasks as sub-computations; this makes it possible to view the data-flow graph of the MapReduce job as a subgraph of the dependency graph, which in addition has control dependencies. Since MapReduce frameworks implicitly keep track of this graph when implementing the data movement and synchronization between the various tasks, building the dependency graph becomes easy and natural.

This decision leads to our basic design, which is shown in Figure 1. In this design, the MapReduce scheduler orchestrates the execution of every MapReduce job normally, by spawning and synchronizing tasks and performing data movement as in a normal MapReduce execution. To record and update the dependency graph implicitly, our design includes a *memoization server* that stores a mapping from the input of a previously run task to the location of the corresponding memoized output. When a task completes, its output is memoized persistently, and a mapping from the input to the location of the output is stored in the memoization server. Then, during an incremental run, when a task is instantiated, the

memoization server is queried to check if the inputs to the task match those of a previous run. If so, the system reuses the outputs from the previous run. Otherwise, the task runs normally and the mapping from its input to the location of the newly produced output is stored in the memoization server.

This basic design raises a series of challenges, which we describe next. In subsequent sections, we describe our key technical contributions that we propose to address these challenges.

0.3.3 Challenge: Transparency

Self-adjusting computation requires knowing the modifications to the input in order to update the output. To this end, it requires a new interface for making changes to the input, so that the edits, which are clearly identified by the interface, can be used to trigger an incremental update. We wish to achieve the efficiency benefits of self-adjusting computation *transparently* without requiring the programmer to change the way they run MapReduce computations. This goal seems to conflict with the fact that HDFS (the system employed to store inputs to MapReduce computations in Hadoop) is an append-only file system, making it impossible to convey input deltas. To overcome this challenge, we store the inputs and outputs of consecutive runs in separate HDFS files and compute a delta between two HDFS files in a way that is scalable and performs well.

0.3.4 Challenge: Efficiency

To achieve efficient incremental updates, we must ensure that MapReduce computations remain stable under small changes to their input, meaning that, when executed with similar inputs, many tasks are repeated and their results can be reused. To define stability more precisely, consider performing MapReduce computations with inputs I and I' and consider the respective set of tasks that are executed, denoted T and T'. We say that a task $t \in T'$ is not *matched* if $t \notin T$, i.e., the task that is performed with input I' is not performed with the input I. We say that a MapReduce computation is *stable* if the time required to execute

the unmatched tasks is small, where small can be more precisely defined as sub-linear in the size of the input.

In the case of MapReduce, stability can be affected by several factors, which we can group into the following two categories: (a) making a small change to the input can change the input to many tasks, causing these tasks to become unmatched; (b) even if a small number of tasks is unmatched, these tasks can take a long time to execute or to transfer possibly reused data. To address these issues, we introduce techniques for (1) performing a stable input partitioning; (2) controlling the granularity and stability of both Map and Reduce tasks; and (3) finding efficient scheduling mechanisms to avoid unnecessary movement of memoized data.

Stable input partitioning. To see why using HDFS as an input to MapReduce jobs leads to unstable computations, consider inserting a single data item in the middle of an input file. Since HDFS files are partitioned into fixed-sized chunks, this small change will shift each partition point following the input change by a fixed amount. If this amount is not a multiple of the chunk size, all subsequent Map tasks will be unmatched. (On average, a single insert will affect half of all Map tasks.) The problem gets even more challenging when we consider more complex changes, like the order of records being permuted; such changes can be common, for instance, if a crawler uses a depth-first strategy to crawl the web, and a single link change can move the position of an entire subtree in the input file. In this case, using standard algorithms to compute the differences between the two input files is not viable, since this would require running a polynomial-time algorithm (e.g., an edit-distance algorithm). We explain how our new file system called Inc-HDFS leads to stable input partitioning without compromising efficiency in Section 0.4.

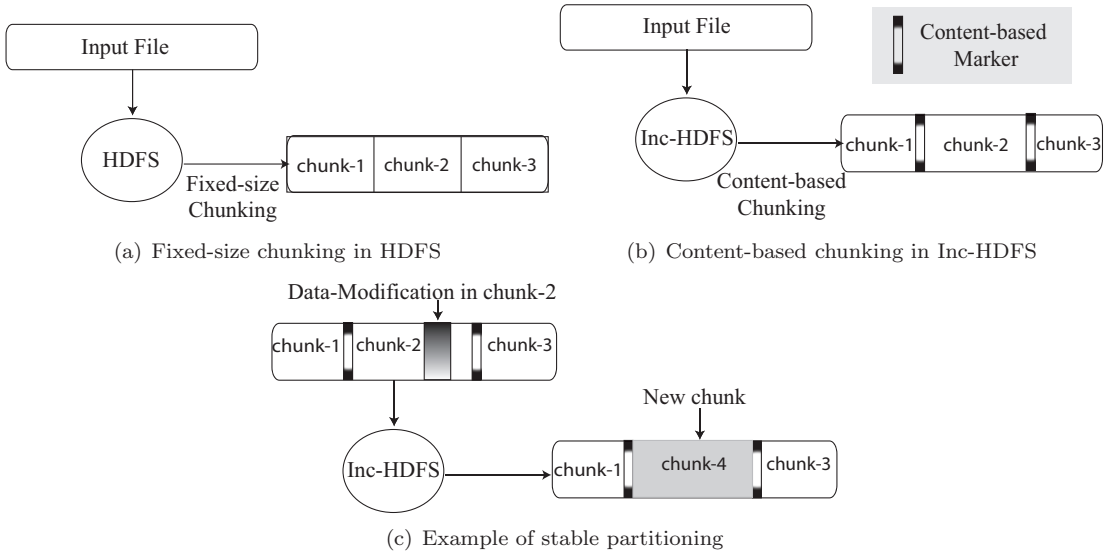
Granularity control. A stable partitioning leads directly to the stability of Map tasks. The input to the Reduce tasks, however, is determined only by the outputs of the Map tasks, since each Reduce task processes all values produced in the Map phase and associated with a given key. Consider, for instance, the case when a single key-value pair is added to a Reduce task that processes a large number of values (e.g., linear in the size of the input). This is problematic since it causes the entire task to be re-computed. Furthermore, even if

we found a way of dividing large Reduce tasks into multiple smaller tasks, this per se would not solve the problem, since we still need to aggregate the results of the smaller tasks in a way that avoids a large recomputation. Thus, we need a way to (i) split the Reduce task into smaller tasks and (ii) eliminate potentially long (namely linear-size) dependencies between these smaller tasks. We solve this problem with a new Contraction phase, where Reduce tasks are broken into sub-tasks organized in a tree. This breaks up the Reduce task while ensuring that long dependencies between tasks are not formed, since all paths in the tree will be of logarithmic length. Section 0.5 describes our proposed approach.

Scheduling. To avoid a large movement of memoized data, it is important to schedule a task on the machine that stores the memoized results that are being reused. To ensure this, we introduce a modification to the scheduler used by Hadoop, in order to incorporate a notion of *affinity*. The new scheduler takes into account affinities between machines and tasks by keeping a record of which nodes have executed which tasks. This allows for scheduling tasks in a way that decreases the movement of memoized intermediate results, but at the cost of a potential degradation of job performance due to stragglers [38]. This is because a strict affinity of tasks results in deterministic scheduling, which prevents a lightly loaded node from performing work when the predetermined node is heavily loaded. Our scheduler therefore needs to strike a balance between work stealing and affinity of memoized results. Section 0.6 describes our modified scheduler.

0.4 Incremental HDFS

In this section we present Incremental HDFS (Inc-HDFS), a distributed file system that enables stable incremental computations in Incoop, while keeping the interface provided by HDFS. Inc-HDFS builds on HDFS, but modifies the way that files are partitioned into chunks to use content-based chunking, a technique that was introduced in LBFS [31] for data deduplication. At a high-level, content-based chunking defines chunk boundaries based on finding certain patterns in the input, instead of using fixed-size chunks. As such, insertions and deletions cause small changes to the set of chunks. In the context of MapReduce, this



(a) Fixed-size chunking in HDFS

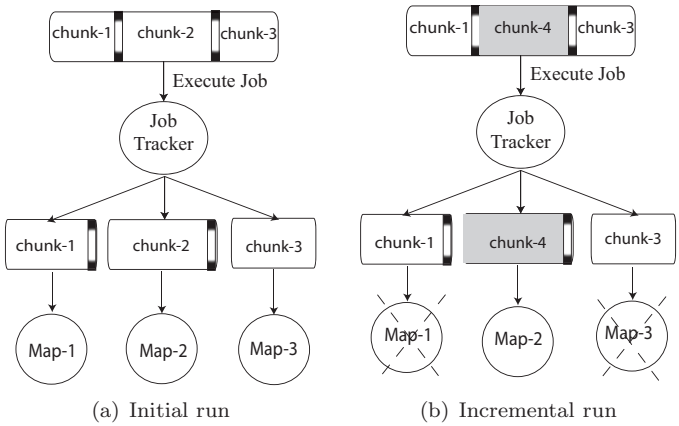
(b) Content-based chunking in Inc-HDFS

(c) Example of stable partitioning

Figure 2: Chunking strategies in HDFS and Inc-HDFS

ensures that the input to Map tasks remains mostly unchanged, which translates into a stable recomputation. Figure 2 illustrates the differences in the strategies for determining chunk boundaries in HDFS and Inc-HDFS. To perform content-based chunking, we scan the entire file, examining the contents of a fixed-width window whose initial position is incremented one byte at a time. For each window, we compute its Rabin fingerprint, and if the fingerprint matches a certain pattern (called a *marker*) we place a chunk boundary at that position. In addition, this approach can be extended to avoid creating chunks that are too small or too large, which could affect the overheads and load balancing properties of MapReduce. (Note that all the system designer can tune is the likelihood of finding a marker, but the actual spacing depends on the input.) This is achieved by setting minimum and maximum chunk sizes: after we find a marker m_i at position p_i, we skip a fixed *offset* O and continue to scan the input after position $p_i + O$. In addition, we bound the chunk length by setting a marker after M content bytes even if no marker is found. Despite the possibility of affecting stability in rare cases, e.g., when skipping the offset leads to skipping disjoint sets of markers in two consecutive runs, we found this to be a very limited problem in practice.

An important design decision is whether to perform chunking during the creation of the input or when the input is read by the Map task. We chose the former because the cost of chunking can be amortized when chunking and producing the input data are done in parallel.



(a) Initial run (b) Incremental run

Figure 3: Incremental Map tasks

This is relevant in cases where the generation of input data is not limited by the storage throughput.

In order to parallelize the chunking process on multicore machines, our implementation uses multiple threads, each of which starts the search for the marker at a different position. The markers that each thread finds cannot be used immediately to define the chunk boundaries, since some of them might have to be skipped due to the minimum chunk size. Therefore, we collect the markers in a centralized list, and scan the list to determine which markers are skipped; the remaining ones form the chunk boundaries. Our experimental evaluation (Section 0.7.4) highlights the performance gains of this optimization, showing that it is instrumental in keeping the performance of Inc-HDFS close to that of HDFS.

0.5 Incremental MapReduce

This section presents our design for incremental MapReduce computations. We split the presentation by describing the Map and Reduce phases separately.

Incremental Map. For the Map phase, the main challenges have already been addressed by Inc-HDFS, which partitions data in such a way that the input to Map tasks ensures stability and also allows for controlling the average granularity of the input that is provided to these tasks. In particular, this granularity can be adjusted by changing how likely it is

to find a marker, and it should be set in a way that strikes a good balance between the following two characteristics: incurring the overhead associated with scheduling many Map tasks when the average chunk size is low, and having to recompute a large Map task if a small subset of its input changes when the average chunk size is large.

Therefore, the main job of Map tasks in Incoop is to implement task-level memoization. To do this, after a Map task runs, we store its results persistently (instead of discarding them after the job execution) and insert a corresponding reference to the result in the memoization server.

During incremental runs, Map tasks query the memoization server to determine if their output has already been computed. If so, they output the location of the memoized result, and conclude. Figure 3 illustrates this process: part (a) describes the initial run and part (b) describes the incremental run where chunk 2 is modified (and replaced by chunk 4) and the Map tasks for chunks 1 and 3 can reuse the memoized results.

Incremental Reduce. The Reduce task processes the output of the Map phase: each Reduce task has an associated key k, collects all the key-value pairs generated by all Map tasks for k, and applies the Reduce function. For efficiency, we apply two levels of memoization in this case. First, we memoize the inputs and outputs of the entire Reduce task to try to reuse these results in a single step. Second, we break down the Reduce phase into a Contraction phase followed by a smaller invocation of the Reduce function to address the stability issues we discussed.

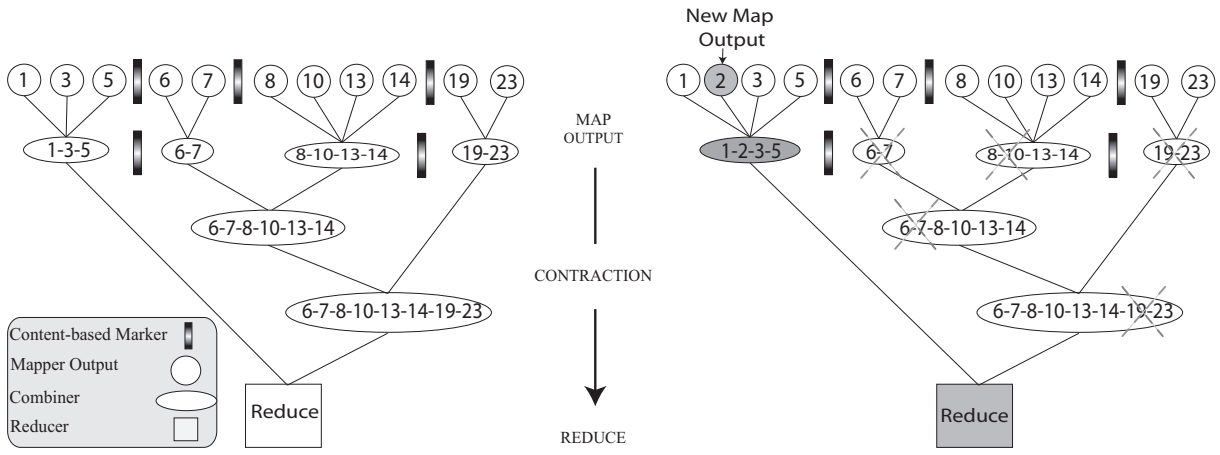
The first level of memoization is very similar to that of Map tasks: the memoization server maintains a mapping from a hash of the input to the location of the result of the Reduce task. A minor difference is that a Reduce task receives input from several Map tasks, and as such the key of that mapping is the concatenation of the collision-resistant hashes from all these outputs. For the Reduce task to compute this key, instead of immediately copying the output from all Map tasks, it fetches the hashes only to determine if the Reduce task can be skipped entirely. Only if this is not the case the data is transferred from Map to Reduce tasks.

As we mentioned, this first level has the limitation that small changes in the input cause the entire Reduce task to be re-executed, which can result in work that is linear in the size of the original input, even if the delta in the input is small. In fact it may be argued that the larger the Reduce task the more likely it is that a part of its input may change. To prevent this stability problem, we need to find a way to control the granularity of the sub-computations in the Reduce phase, and organize these sub-computations in way that avoids creating a long dependence chain between sub-computations, otherwise a single newly computed sub-computation could also trigger a large amount of recomputation.

To reduce the granularity of Reduce tasks, we propose a new *Contraction Phase*, which is run by Reduce tasks. This new phase takes advantage of *Combiners*, a feature of the MapReduce framework [18], also implemented by Hadoop, which originally aims at saving bandwidth by offloading part of the computation performed by the Reduce task to the Map task. To this end, the programmer specifies a Combiner function, which is invoked by the Map task, and pre-processes a part of the Map output, i.e., a set of ⟨key,value⟩ pairs, merging them into a smaller number of pairs. The signature of the combiner function uses the same input and output type in order to be interposed between the Map and Reduce phase. Its inputs and output arguments are a sequence of ⟨key,value⟩ pairs. In all the MapReduce applications we analyzed so far, the Combiners and the Reduce functions perform similar work.

The Contraction phase uses Combiners to break up Reduce tasks into several applications of the Combine function. In particular, we start by splitting the Reduce input into chunks, and apply the Combine function to each chunk. Then we recursively form chunks from the aggregate result of all the Combine invocations and apply the Combine function to these new chunks. The data size gets smaller in each level, and, in the last level, we apply the Reduce function to the output of all the Combiners from the second to last level.

Given the signature of Combiner functions we described before, it is syntactically correct to interpose any number of Combiner invocations between the Map and Reduce functions. However, semantically, Combiners are invoked by the MapReduce or Hadoop frameworks at most once per key/value pair that is output by a Map task, and therefore MapReduce



(a) First run (c) Incremental run

Figure 4: Stability of the Contraction phase

programs are only required to ensure the correctness of the MapReduce computation for a single Combiner invocation, that is:

$$R \circ C \circ M = R \circ M$$

where R, C, and M represent the Reduce, Combiner and Map function, respectively. Our new use of Combiner functions introduces a different requirement, namely:

$$R \circ C^n \circ M = R \circ M, \forall n > 0$$

It is conceivable to write a Combiner that meets the original requirement but not the new one. However, we found that, in practice, all of the Combiner functions we have seen obey the new requirement.

Stability of the Contraction phase. When deciding how to partition the input to the Contraction phase, the same issue that was faced by the Map phase arises: if a part of the input to the Contraction phase is removed or a new part is added, then a fixed-size partitioning of the input would not ensure the stability of the dependence graph. This problem is illustrated in Figure 4, which shows two consecutive runs of a Reduce task, where a Map task (#2) produces in the second but not in the first run a value associated with the key being processed by this Reduce task. In this case, a partitioning of the input into groups with a fixed number of input files would cause all groups of files to become different from

one run to the next.

To solve this, we again employ content-based chunking, which is applied to every level of the tree of combiners that forms the Contraction phase. The way we perform content-based chunking in the Contraction phase differs slightly from the approach we took in Inc-HDFS, for both efficiency and simplicity reasons. In particular, given that the Hadoop framework splits the input to the Contraction phase into multiple files coming from different Map tasks, we require chunk boundaries to be at file boundaries. This way we leverage the existing input partitioning, which not only simplifies the implementation, but also avoids reprocessing this input, since we can use the hash of each input file to determine if a marker is present: we do this by testing if the hash modulo a pre-determined integer M is equal to a constant $k < M$.

Figure 4 also illustrates the importance of content-based chunking. In this example, the marker, which delimits the boundaries between groups of input files, is present only in outputs #5, 7, and 14. Therefore, inserting a new map output will change the first group of inputs but none of the remaining ones. This figure also illustrates how this change propagates to the output: it leads to a new Combiner invocation (labelled 1-2-3-5) and the final Reduce invocation. For all the remaining Combiners we can reuse their memoized outputs without re-executing them.

0.6 Memoization-aware scheduler

The main job of the Hadoop scheduler is to assign Map and Reduce tasks to cluster machines, taking into account machine availability, cluster topology, and the locality of input data. However, this scheduler does not fit well with Incoop because it does not consider the location of memoized results.

The memoization-aware scheduler addresses this shortcoming. To understand the goals that guide its design we need to consider the Map and Reduce phases separately.

For the Map phase, the location of memoized results is irrelevant, since, in case the Map task is able to reuse these results, it can just communicate the location of the results to the

scheduler, who then points the Reduce tasks to this location. Therefore the memoization-aware scheduler works like the Hadoop scheduler in the Map phase.

For scheduling Reduce tasks, which now perform the Contraction phase as well as the final Reduce invocation, the memoization-aware scheduler must try to schedule them in nodes where the memoized results they may use are stored. This is important because the Contraction phase often uses a combination of newly computed and memoized results, whenever only a part of its inputs has changed. In addition to this design goal, the scheduler must provide some flexibility by allowing tasks to be scheduled on nodes that do not store memoized results, otherwise it can lead to the presence of stragglers, i.e., individual poorly performing nodes that can delay the job completion [38].

The new scheduler for Reduce tasks strikes a balance between these two goals by being aware of the location of memoized results, while at the same time implementing a simple work-stealing algorithm to adapt to varying resource availability. The scheduler maintains a separate queue of pending Reduce tasks for each node in the cluster (instead of a single queue for all nodes). Each queue is populated with the tasks that should run on that node in order to exploit the location of memoized results. Whenever a node requests more work, the scheduler dequeues the first task from the corresponding queue and assigns the task to the node for execution. In case the queue for the requesting node is empty, the scheduler attempts to steal work from other task queues, by choosing a pending task from the task queue with maximum length. Our experimental evaluation (Section 0.7.6) shows the effectiveness of the new scheduler.

0.7 Implementation and Evaluation

This section describes the implementation and experimental evaluation of Incoop.

Application	Description
`K-Means`	K-means clustering is a method of cluster analysis for partitioning n data points into k clusters, in which each observation belongs to the cluster with the nearest mean.
`Word-Count`	Word count determines the frequency of words in a document.
`KNN`	K-nearest neighbors classifies objects based on the closest training examples in a feature space.
`CoMatrix`	Co-occurrence matrix generates an $N \times N$ matrix, where N is the number of unique words in the corpus. A cell m_{ij} contains the number of times word w_i co-occurs with word w_j.
`BiCount`	Bigram count measures the prevalence of each subsequence of two items within a given sequence.

Table 1: Applications used in the performance evaluation

0.7.1 Implementation

We built a prototype of Incoop based on Hadoop-0.20.2. The implementation of Inc-HDFS extends HDFS with stable input partitioning, and incremental MapReduce extends Hadoop with support for memoization, the Contraction phase, and the memoization-aware scheduler.

The Inc-HDFS file system keeps exactly the same interface and semantics for all existing HDFS calls, and implements the parallel scanning scheme to find markers that we described in §0.4. For our experiments, we set the minimum chunk size (i.e., the number of Bytes skipped after finding a marker) to 40MB, unless otherwise noted. In HDFS, we use a chunk size of 64 MB.

The memoization server is built as a wrapper around the in-memory key/value store memcached v1.4.5 system. The memcached server is co-located with the name node, which is the directory server from Hadoop. All the memoized results are stored on Inc-HDFS with a replication factor of 1. This implies that in case of a data node crash these results need to be recomputed. We implemented a simple garbage collection scheme to discard old memoized results, which only retains the results from the most recent run of a given MapReduce job, and discards all results from previous runs.

Finally, the Contraction phase is implemented by aggregating all keys that are processed by each node, instead of building one contraction tree for each key processed by that node, since this is closer to the original Hadoop implementation.

0.7.2 Applications

We evaluated Incoop using a set of MapReduce applications from the open source Apache Mahout project summarized in Table 1. These applications cover a wide range of domains such as machine learning, natural language processing, pattern recognition, and document analysis. Furthermore, this set includes both data-intensive (`WordCount, Co-Matrix, BiCount`), and CPU-intensive (`KNN` and `K-Means`) computations, corresponding to different ratios of I/O to CPU load. We did not have to modify any of these applications to work with Incoop.

The three data-intensive applications use documents written in a natural language as input. In our benchmarks, we used as input the contents of Wikipedia from a public data set[1]. The two CPU-intensive applications use a set of points in a d-dimensional space as input. In this case we used a set of randomly generated points in a 50-dimensional unit cube. To obtain reasonable running times, we chose all the input sizes in a way that the running time of each job would be approximately one hour.

0.7.3 Overview of the experiments

Our evaluation tries to answer the following questions:

- What are the overheads introduced by Inc-HDFS compared to HDFS? (§0.7.4)
- What are the performance gains of using Incoop when compared to recomputing from scratch? (§0.7.5)
- How important are each of the design features we introduce? (§0.7.6)
- What are the overheads introduced by Incoop when a job is executed for the first time? (§0.7.7)

To answer these questions, we ran experiments using the following setting and measured the following data.

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

Experimental setup. We ran experiments on a cluster of 20 machines, running the Linux kernel 2.6.32 in 64-bit mode, connected by a gigabit ethernet. The name node and the job tracker of Hadoop ran on a master machine, which had a 12-core Intel Xeon processor and 12 GB of RAM. When we run an Inc-HDFS client on this machine, the parallel chunking code in Inc-HDFS is parameterized to spawn 12 threads, i.e., one thread per core. The data nodes and task trackers of Hadoop ran on the remaining 19 machines, which had AMD Opteron-252 processors and 4GB of RAM. The task trackers were parameterized to use two Map and two Reduce slots per worker machine.

Work and time. We separately measure work and time to compare the performance across runs. *Work* is the sum of all the computation time performed by all the tasks, which eliminates the effects of having some machines idling waiting to synchronize with other machines. *(Parallel) time* refers to the total running time for the job. The two metrics are related through the work-time principle, which states that a computation with W work can be executed on P machines (or processors) in $\frac{W}{P}$ time if there are no scheduling overheads. 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. When evaluating Incoop, we need to consider two types of runs. The *initial run* operates on data that was never seen before, and therefore can start with an empty memoization server, which is then populated by the initial run. The *incremental run* corresponds to a subsequent run where the input is modified by a certain fraction, and the system tries to reuse sub-computations to the extent possible.

Speedup. We present the results comparing the performance of Incoop and Hadoop by plotting the speedup, i.e., the ratio of the work or parallel time required by Hadoop to the work or time required by Incoop. In most cases we plot how this speedup varies as we change the fraction of the input that differs from the initial to the incremental run. To run an experiment where $x\%$ of the input data differs between the two runs, we randomly chose

$x\%$ of the chunks in the input and replaced them with new equally-sized chunks with new content.

0.7.4 Incremental HDFS

Version	Skip Offset [MB]	Throughput [MB/s]
HDFS	-	34.41
Incremental HDFS	20	32.67
	40	34.19
	60	32.04

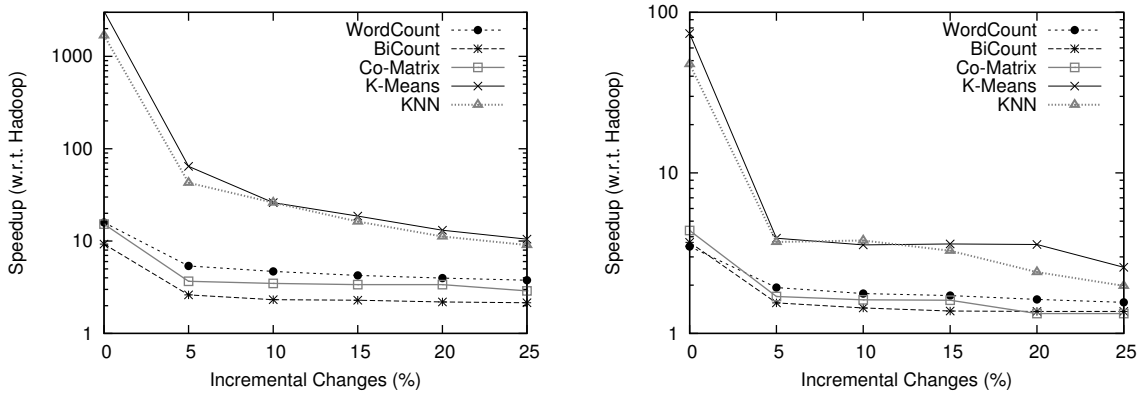
Table 2: Throughput of HDFS and Inc-HDFS

We compare the throughput during an upload of a dataset of 3 GB in HDFS and Inc-HDFS, for a varying number of skipped bytes in Inc-HDFS. The client machine that is writing to the file system runs on the same machine as the name node of Hadoop. The results of this experiment are summarized in Table 2. Overall, Inc-HDFS adds only a small throughput overhead compared to HDFS, which can be attributed to the fingerprint computation.

This overhead becomes more visible for the smallest skip offset of 20MB. This was expected since the Rabin fingerprint needs to be computed for a larger fraction of the data. Somewhat more surprising was the reduction in throughput for the largest skip offset of 60MB. This is due to the fact that increasing the skip offset leads to an increase in the average chunk size, which in turn leads to decreasing the amount of parallelism towards the end of the data upload. We therefore found 40MB to be a reasonable compromise between these two negative factors.

0.7.5 Work and time speedup

We report the speedup of Incoop relative to Hadoop in terms of work and time in Figure 5(a) and Figure 5(b), respectively. The results show that incremental computations Incoop are significantly faster than recomputing the data from scratch using Hadoop, with work and time speedups ranging from 3 to 1000X for incremental modifications of up to 25%. The



(a) Work speedups versus change size

(b) Time speedups versus change size.

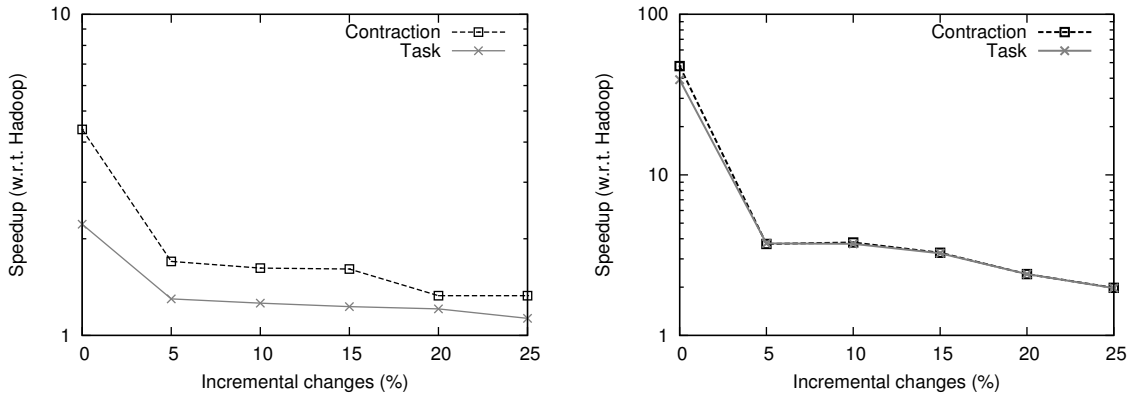
Figure 5: Performance gains for `Incoop` in comparison to `Hadoop`

results also point out that CPU-intensive applications can obtain larger speedups than data-intensive ones, most likely because some data movement is unavoidable in the incremental run (e.g., to read the new input or to write the new output). Finally, we note that the speedups decrease with the fraction of changed data, as expected. However, with very small changes, speedups in work are not translated into speedups in time, because of the effect of there being less parallelism available when only a few tasks need to be rerun.

0.7.6 Individual design features

Next we evaluate the some of the design features individually, namely the Contraction phase and the new scheduler.

Contraction phase. To evaluate the Contraction phase, we run two versions of Incoop, a version that only memoizes the output of an entire Reduce task, and the full design that includes the Contraction phase. We identify these two versions as `Task` and `Contraction`. Figure 6 compares the work and time speedup of the two versions using an application of each class (`CoMatrix` is data-intensive and `KNN` is CPU-intensive). The Contraction phase does not change the performance of `KNN` but significantly improves the performance of `CoMatrix`. This is related to the fact that the Reduce phase in `KNN` performs a simple computation, and thus has little to gain from the Contraction phase. Given this fact, it is noteworthy that the



(a) Co-occurrence Matrix (b) k-NN Classifier

Figure 6: Performance gains comparison between `Contraction` and `task` variants

Contraction phase did not add significant overhead.

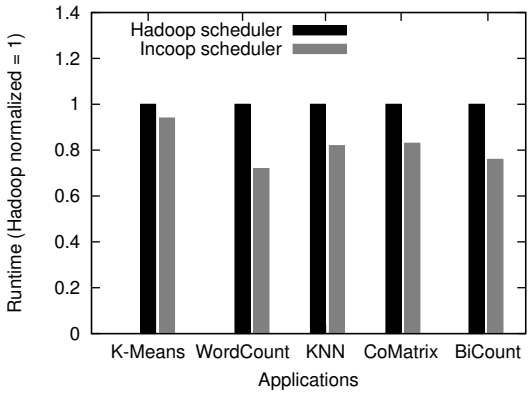
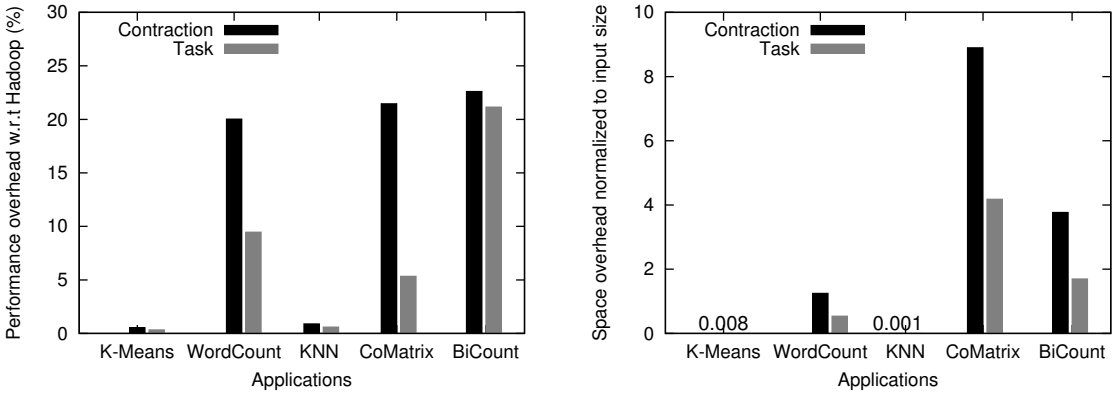


Figure 7: Effectiveness of scheduler optimizations.

Scheduler modification. We now evaluate the effectiveness of the memoization-aware scheduler. In Figure 7 we compare the time to run the various applications in Incoop using the new and the original Hadoop scheduler. The Y-axis presents the total running time normalized to the time using the Hadoop scheduler. The memoization-aware scheduler cuts the running time by 30% for data-intensive applications, and almost 15% for CPU-intensive applications. This highlights the importance of this design aspect.



(a) Performance overhead for the first job run

(b) Space overhead

Figure 8: Overheads imposed by `Incoop` in comparison to `Hadoop`

0.7.7 Overheads

Next we evaluate the price that is paid for the gains we showed in the previous section, namely the overheads introduced by Incoop during the initial run, and the space requirements for storing memoized results. The results are shown in Figure 8.

Performance overhead. Figure 8(a) depicts the performance overhead for the first run for the `Task` and the `Contraction` variants as described before. We stress that these overheads are a one-time cost that can lead to substantial gains in subsequent runs. The overhead varies from 5 to 22%, and is lower for CPU-intensive applications (`K-Means` and `KNN`), since the time to compute over the data dominates the time to transfer this data to be stored. For data-intensive applications (`WordCount, Co-Matrix`, and `BiCount`), the overheads using the `Contraction` approach are higher, due to the overheads for processing all levels of the tree of the Contraction phase.

Space overhead. Figure 8(b) shows the the space overhead for storing memoized results as a fraction of the input size. The results show that the Contraction approach requires more space to store the results for all the levels of the tree, as expected. Overall, space overheads can reach up to $9X$ (`CoMatrix`), highlighting the trade-off between space in time that enabled by this approach.

0.8 Related Work

Several fields and research communities have looked into related problems. In this section, we present an overview of this body of related work.

Dynamic algorithms. There are several algorithms for a variety of problems that are designed for a dynamic input [3, 16, 21, 35, 20]. This body of work shows that dynamic algorithms can be more efficient than their conventional counterparts. However, dynamic algorithms can be difficult to develop and implement: some problems took years of research to solve and many remain open.

Programming language approaches. This body of work proposes programming language support to achieve automatic incrementalization (e.g. [35]). Recent advances on self-adjusting computation propose general-purpose techniques that can achieve optimal update times (e.g., [6, 5, 24]). Self-adjusting computation offers abstractions for automatic incrementalization, allowing programs written in a conventional style to be compiled to programs that can respond to changes in their data inputs automatically. While earlier proposals [4] required program annotations for efficiency, recent results show that some of these annotations can be inferred by type systems [15]. Stability, one of the key concepts that we use in this paper was developed initially as an algorithmic analysis technique [3] and later as part of a cost semantics [27]. Self-adjusting computation has been shown to be effective for a wide variety of problems and has led to progress on and sometimes solutions to open problems in several domains including computational geometry, machine learning, and checking of data structural invariants (e.g., [36, 8, 5, 37, 9]). Some earlier [7, 23] and more recent work [13] realized that many parallel algorithms are amenable to self-adjusting computation and developed techniques for taking advantage of both simultaneously.

Incremental database view maintenance. The database community proposed several techniques for incrementally updating a database view (a predetermined query) as the database state is updated. This can be implemented either directly by modifying the database engine internals or using database triggers that update the view whenever the data changes [14]. Even though we share the same goals as incremental view maintenance, the

techniques used by this class of systems are very specific to database semantics.

Large-scale incremental processing. We split the prior work on incremental parallel computations for large data sets into two categories: non-transparent and transparent proposals.

Non-transparent approaches are exemplified by Google's Percolator [33], which provides the programmer with the ability to write a handler (called an observer) that is triggered upon data changes. Observers in turn can modify other data forming a dependence chain that implements the incremental data processing. Similarly, continuous bulk processing (CBP) [28] offers a new programming model with primitives to store and reuse prior state for incremental processing upon the arrival of new data. Naiad [30] is another proposal for incremental processing that offers differential dataflow. These proposals have two drawbacks, which are addressed by Incoop. The first is that they depart from existing models and therefore do not allow for reusing the large existing base of MapReduce programs. The second is that the programmer must devise a dynamic algorithm, which, as mentioned, can be difficult to design with efficiency for many problems.

Examples of transparent approaches include DryadInc [34], which like Incoop, caches task results, and Nectar [22], which caches prior results of entire LINQ sub-expressions. Also, although not fully transparent, Haloop [12] provides task-level memoization techniques for iterative computations. Incoop improves on these proposals by using a set of principles from related work to identify and overcome the situations where task-level memoization is inefficient, such as the stable input partitioning or the Contraction phase.

Our own short position paper [10] makes the case for applying techniques inspired by self-adjusting computation to large-scale data processing in general, and uses MapReduce as an example. This position paper, however, models MapReduce in a sequential, single-machine implementation of self-adjusting computation called CEAL [24], and does not offer a full-scale distributed design and implementation such as the system we presented.

Finally, the idea of breaking up the work of a Reduce task using Combiner functions was also adopted for the purpose of minimizing network bandwidth overheads [29, 17].

Stream processing systems. Comet [25] proposed the Batched Stream Processing (BSP) model, where queries are triggered upon appending a chunk of data to an input stream. In contrast to Comet, we are compatible with the MapReduce model and focus on several issues like controlling task granularity or input partitioning that do not arise in Comet's model.

NOVA [32] allow for the incremental execution of Pig programs as new data continuously arrives. Much like the work on incremental view maintenance, NOVA introduces a workflow manager that rewrites the computation to identify the parts of the computation affected by incremental changes and produce the necessary update function, which in turn runs on top of the existing Pig/Hadoop framework. However, as noted by the authors of NOVA, an alternative, more efficient design would be to modify the underlying Hadoop system to support this functionality. This is precisely that path we took in Incoop. Furthermore, our work is more general since it can be applied to any MapReduce program, and not only the programs produced by the Pig framework.

Other systems such as Storm [2] or S4 [1] work at a finer granularity by triggering a processing routine each time a (possibly small) input arrives. Deduce [26] presents a hybrid solution for real-time data analytics using a combination of techniques from batch processing in MapReduce and streaming processing in IBM's System S. In contrast to these proposals, the focus of Incoop is to provide incremental processing in batch processing workloads.

0.9 Conclusion

This chapter presents a set of principles and techniques for performing incremental MapReduce computations. We build on prior work from several fields, most notably contributions from the programming languages community on self-adjusting computation, a technique for incremental computations, which we extend to large-scale parallel computations. The resulting system combines the efficiency of recomputing a small subset of sub-computations affected by each input change with the ability to transparently execute existing MapReduce computations. This work thus has the potential to bring substantial performance improvements to MapReduce computations when these computations are repeated on evolving data,

We achieve the efficiency improvements by reorganizing MapReduce computations to be executed slightly differently in a way that is consistent with the principles of self-adjusting computation so that the benefits of incremental computation can be delivered transparently. Specifically, Incoop organizes a MapReduce computation such that it remains stable even as the input data set changes—that is, computations performed on similar datasets themselves remain similar. To ensure the stability of the Map phase, Incoop incorporates content-based chunking to the file system to detect incremental changes in the input file and to partition the data; this maximizes reuse of the results of the Map phase. To ensure the stability of the Reduce phase, Incoop performs an additional Contraction phase, where results from smaller reductions are combined to form larger reductions in a balanced-tree fashion; this enables the reuse of results from the intermediate states of an otherwise large reduction. Incoop proposes a memoization-aware scheduler that improves efficiency by taking the location of previously computed results into account.

We view this work as a first step in taking advantage of the benefits of transparent incremental computation in parallel and distributed systems. We envision that the work will open many avenues of research for applying similar incrementalization techniques to other distributed systems.

Acknowledgement

This experimental evaluation is supported by AWS in Education Grant award. The research of R. Rodrigues has received funding from the European Research Council under an ERC starting grant.

References

[1] Apache s4 (http://incubator.apache.org/s4/). May, 2013.
[2] Storm (http://storm-project.net/). May, 2013.
[3] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[4] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. A Library for Self-Adjusting Computation. *Electronic Notes in Theoretical Computer Science*, 148(2), 2006.

[5] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Programming Languages and Systems*, 32(1):1–53, 2009.

[6] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Programming Languages and Systems*, 28(6):990–1034, 2006.

[7] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and M. Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 2004.

[8] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust Kinetic Convex Hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, September 2008.

[9] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *Proc. 26th Symp. Computational Geometry (SCG'10)*, 2010.

[10] P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'11)*.

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

[12] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *36th International Conference on Very Large Data Bases*, Singapore, September 14–16, 2010.

[13] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: A model for parallel and incremental computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

[14] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 577–589, 1991.

[15] Y. Chen, J. Dunfield, and U. A. Acar. Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2012.

[16] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

[17] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: exploiting in-network aggregation for big data applications. In *NSDI*, 2012.

[18] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*.

[19] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*. CRC, 2005.

[21] 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, in Computer and Information Science, 2005.

[22] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in data centers. In *Proc. 9th Symp. Operating Systems Design and Implementation (OSDI'10)*.

[23] M. Hammer, U. A. Acar, M. Rajagopalan, and A. Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Proceedings of the first workshop on Declarative Aspects of Multicore Programming*, 2007.

[24] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: A C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[25] 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 *Proc. 1st Symposium on Cloud computing (SoCC'10)*.

[26] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. Deduce: at the intersection of mapreduce and stream processing. In *EDBT*, 2010.

[27] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

[28] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *1st Symp. on Cloud computing(SoCC'10)*.

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

[30] F. McSherry, R. I. Isaacs, M. Isard, and D. G. Murray. Composable incremental and iterative data-parallel computation with naiad. *MSR-TR-2012-105*, 2012.

[31] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. 18th Symp. on Operating systems principles (SOSP'01)*.

[32] C. Olston and et.al. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD, 2011.

[33] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. 9th Symposium on Operating Systems Design and Implementation (OSDI'10)*.

[34] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *Worksh. on Hot Topics in Cloud Computing (HotCloud'09)*.

[35] G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *Proc. 20th Symp. Princ. of Progr. Languages (POPL'93)*.

[36] A. Shankar and R. Bodik. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming language Design and Implementation*, 2007.

[37] O. Sümer, U. A. Acar, A. Ihler, and R. Mettu. Adaptive exact inference in graphical models. *Journal of Machine Learning*, 8:180–186, 2011.

[38] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, 2008.