Incoop: MapReduce for Incremental Computations

Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, Rafael Pasquini†
Max Planck Institute for Software Systems (MPI-SWS) and
†Faculdade de Computação - Universidade Federal de Uberlândia (FACOM/UFU)
{bhatotia, awieder, rodrigo, umut, pasquini}@mpi-sws.org

ABSTRACT

Many online data sets evolve over time as new entries are slowly added and existing entries are deleted or modified. Taking advantage of this, systems for incremental bulk data processing, such as Google's Percolator, can achieve efficient updates. To achieve this efficiency, however, these systems lose compatibility with the simple programming models offered by non-incremental systems, e.g., MapReduce, and more importantly, requires the programmer to implement application-specific dynamic algorithms, ultimately increasing algorithm and code complexity.

In this paper, we describe the architecture, implementation, and evaluation of Incoop, a generic MapReduce framework for incremental computations. Incoop detects changes to the input and automatically updates the output by employing an efficient, fine-grained result re-use mechanism. To achieve efficiency without sacrificing transparency, we adopt recent advances in the area of programming languages to identify the shortcomings of task-level memoization approaches, and to address these shortcomings by using several novel techniques: a storage system, a contraction phase for Reduce tasks, and a affinity-based scheduling algorithm. We have implemented Incoop by extending the Hadoop framework, and evaluated it by considering several applications and case studies. Our results show significant performance improvements without changing a single line of application code.

Categories and Subject Descriptors

H.3.4 [**Information Systems**]: Systems and Software—Batch processing systems, Distributed systems

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Self-adjusting computation, memoization, stability

SOCC'11, October 27–28, 2011, Cascais, Portugal.

1. INTRODUCTION

As organizations produce and collect increasing amounts of data, analyzing it becomes an integral part of improving their services and operation. The MapReduce paradigm offers the ability to process data using a simple programming model, which hides the complexity of the infrastructure required for parallelization, data transfer, scalability, fault tolerance and scheduling.

An important property of the workloads processed by MapReduce applications is that they are often incremental; i.e., MapReduce jobs often run repeatedly with small changes in their input. For instance, search engines will periodically crawl the Web and perform various computations on this input, such as computing a Web index or the PageRank metric, often with very small modifications, e.g., at a ratio of 10X to 1000X in the overall input [19, 14].

This incremental nature of workloads suggests that performing large-scale computations incrementally can improve efficiency dramatically. Broadly speaking there are two approaches to achieve such efficient incremental updates. The first approach is to devise systems that provide the programmer with facilities to store and use state across successive runs so that only sub-computations that are affected by the changes to the input need to be executed. This is precisely the strategy taken by major Internet companies who developed systems like Percolator [19] or CBP [14]. This approach, however, requires adopting a new programming model and a new API that differs from the one used by MapReduce. These new programming APIs also require the programmer to devise a way to process updates efficiently, which can increase algorithmic and software complexity. Research in the algorithms community on algorithms for processing incrementally changing data shows that such algorithms can be very complex even for problems that are relatively straightforward in the non-incremental case [6, 9].

The second approach would be to develop systems that can reuse the results of prior computations transparently. This approach would shift the complexity of incremental processing from the programmer to the processing system, essentially keeping the spirit of high-level models such as MapReduce. A few proposals have taken this approach. For example, in the context of the Dryad system, DryadInc [20] and Nectar [10] provide techniques for task-level or LINQ expression-level memoization.

In this paper we present a system called Incoop, which allows existing MapReduce programs, not designed for incremental processing, to execute transparently in an incremental manner. In Incoop, computations can respond auto-

matically and efficiently to modifications to their input data by reusing intermediate results from previous runs, and incrementally updating the output according to the changes in the input. To achieve efficiency, Incoop relies on memoization, but goes beyond the straightforward task-level application of this technique by performing a stable partitioning of the input and by reducing the granularity of tasks to maximize result re-use. These techniques were inspired by recent advances on self-adjusting computation (e.g., [2, 1, 12]). To further improve performance, Incoop also employs affinity-based scheduling techniques.

Our contributions include the following.

- **Incremental HDFS.** Instead of storing the input of MapReduce jobs on HDFS, we devise a file system called Inc-HDFS that provides mechanisms to identify similarities in the input data of consecutive job runs. The idea is to split the input into chunks whose boundaries depend on the file contents so that small changes to the input do not change all chunk boundaries. While preserving compatibility with HDFS (by offering the same interface and semantics), Inc-HDFS partitions the input to maximize the opportunities for reusing results from previous runs.
- **Contraction phase.** We propose techniques for controlling the granularity of tasks by dividing large tasks into smaller subtasks, which can be reused even when the large tasks cannot. This is particularly challenging for Reduce tasks, whose granularity depends solely on their input. To control granularity we propose a new Contraction phase that leverages Combiner functions, normally used to reduce network traffic by anticipating a small part of the processing done by Reduce tasks.
- **Memoization-aware scheduler.** To improve the effectiveness of memoization, we propose an affinity-based scheduler that uses a work stealing algorithm to minimize the amount of data movement across machines. Our new scheduler strikes a balance between exploiting the locality of previously computed results and executing tasks on available machines.
- **Use cases.** We employ Incoop to demonstrate two important use cases of incremental processing: *incremental log processing*, where we use Incoop to build a framework to incrementally process logs as more entries are added to them; and *incremental query processing*, where we layer the Pig framework on top of Incoop to enable relational query processing on continuously arriving data.

We implemented Incoop and evaluated it using five MapReduce applications and the two use cases. Our results show that we achieve significant performance gains, while incurring only a modest penalty during runs that cannot take advantage of memoizing previous results, namely the initial run of a particular job. The results also show the effectiveness of the individual techniques we propose.

The rest of this paper is organized as follows. Section 2 presents an overview of Incoop. The system design is detailed in Sections 3, 4, and 5. We present an analysis in Section 6, and an experimental evaluation in Section 7. Related work and conclusions are discussed in Section 8 and Section 9, respectively. Finally, we cover the two case studies in Appendix A and the proofs for the analytic performance study in Appendix B.

2. SYSTEM OVERVIEW

This section presents the goals, basic approach, and main challenges underlying the design of Incoop.

2.1 Goals

Our goal is to devise a system for large-scale data processing that is able to realize the performance benefits of incremental computations, while keeping the application complexity and development effort low. Specifically, this translates to the following two goals.

- **Transparency.** A transparent solution can be applied to existing bulk data processing applications without changing them. This (i) makes the approach automatically applicable to all existing applications while preserving the full generality, and (ii) requires no additional effort from the programmer to devise and implement an efficient incremental update algorithm.
- **Efficiency.** One lessons to take away from self-adjusting-computation work for incrementalization [2] is that this transparent incremental processing can be asymptotically more efficient (often by a linear factor) than complete, from-scratch re-computation. At the scale of the bulk data processing jobs that run in today's data centers, these asymptotic improvements can translate to huge speedups and cluster utilization savings. We aim to realize such speedups in practice.

Even though it would be possible to devise solutions that work with various types of data processing systems, in this paper, we target the MapReduce model, which has emerged as a *de facto* standard for programming bulk data processing jobs. The remainder of this section presents an overview of the challenges in the design of Incoop, an extension of Hadoop that provides transparent incremental computation of bulk data processing jobs.

Our design adapts the principles of self-adjusting computation (Section 8) to the MapReduce paradigm and the Hadoop framework. The idea of self-adjusting computations is to track dependencies between the inputs and outputs of different parts of a single-machine computation, and in subsequent runs, only rebuild the parts of the computation affected by changes in the input. A computation graph records the dependencies between data and (sub)computations. Nodes of the computation graph represent sub-computations and edges between nodes represent update-use relationships, i.e., dependencies, between computations: there is an edge from one node to another if the latter sub-computation uses some data generated by the former. To change the input dynamically, the programmer indicates which parts of the input were added, deleted, and modified (using a special-purpose interface) and the computation graph is used to determine the sub-computations (nodes) that are affected by these changes. The affected sub-computations are then executed, which can recursively affect other sub-computations. Results of sub-computations that remain unaffected are reused by a form of computation memoization.

Although the work on self-adjusting computation offers a general-purpose framework for developing computations that can perform incremental updates efficiently, it has not

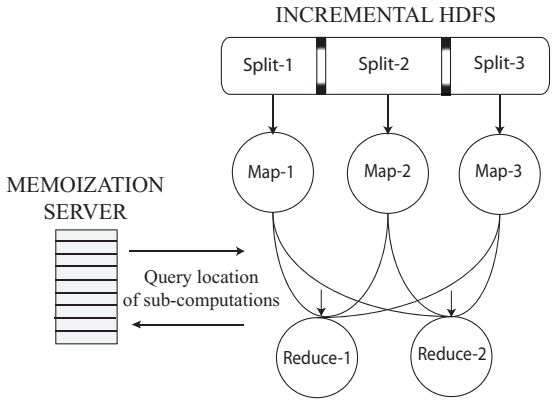


Figure 1: Basic design of Incoop

been applied to a distributed setting. In addition, the approach does not support transparency, and its efficiency critically depends on certain properties of the computation. In the rest of this section, we briefly outline these challenges and how we overcome them.

2.2 Challenges: Transparency

Self-adjusting computation traditionally requires the programmer to annotate programs with specific language primitives. These primitives help the compiler and the run-time system identify dependencies between data and computations. The interface for making changes to the input is also changed in a way that helps identify the "edits" to the input, i.e., how the input changes. To adapt this approach to the MapReduce model while maintaining transparency, we need to address some important questions.

Input changes. Our goal of transparency, coupled with the fact that the file system used to store inputs to MapReduce computations (HDFS) is an append-only file system, makes it impossible to convey generic modifications to an existing input. To maintain a backwards-compatible interface, but still allow for incremental processing of generic workloads, we allow for both the inputs and the outputs of consecutive runs to be stored in separate HDFS files. An alternative, which we consider in Appendix A.1, is to force new data to be only appended to the input, which is well-suited for our case study of periodically processing a log that grows throughout the system lifetime. Another alternative mechanism, which we do not explore in this paper, is to store the input in a structured data store such as HBase, which supports versioning of individual data items, and therefore allows for keeping track of which table cells have changed. Throughout the paper we will consider the first approach, since it is more general.

Programmer annotations. To eliminate programmer annotations that help identify dependencies, we exploit the structure of MapReduce computations. Specifically, we leverage the fact that, in the MapReduce paradigm, the data flow graph has a fixed structure, and the framework implicitly keeps track of this structure by maintaining the dependencies between the various tasks, which form a natural unit of sub-computation.

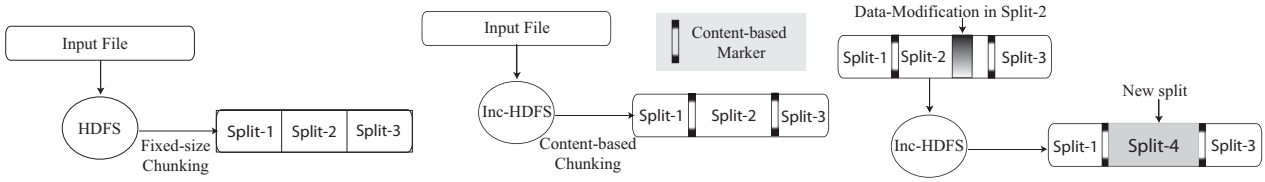
Based on these ideas, we arrive at a simple design that we use as a starting point, which is depicted in Figure 1, and can be described at a high level as follows. We add a *memoization server* that aids in locating the results of previous sub-computations. The role of this server is to store a mapping from the input of a previously run task to the location of the respective output. During a run, whenever a task completes, its output is stored persistently, and a mapping from the input to the location of the output is stored in the memoization server. Then, whenever a task runs, the memoization server is queried to check if the inputs to the task match those of a previous run of the computation. If so, we reuse the outputs that were kept from that previous run. This way only the parts of the computation affected by input changes are re-computed. Finally, we periodically identify and purge old entries from the memoization server, and the respective results from persistent storage, to ensure that storage does not grow without bounds.

2.3 Challenges: Efficiency

To achieve efficient dynamic updates, we must ensure that MapReduce computations remain stable under small changes to their input. Specifically, we define stability as follows. Consider performing MapReduce computations with inputs I and I' and consider the set of tasks that are executed, denoted T and T' respectively. We say that a task $t \in T'$ is not *matched* if $t \notin T$, i.e., the task that is performed with the second inputs I' is not performed with the first input. We say that a MapReduce computation is *stable* if the time required to execute the unmatched tasks is small, ideally, sub-linear in the size of the input. More informally, a MapReduce computation is stable if, when executed with similar inputs, the set of tasks that are executed are also similar, i.e., many tasks are repeated.

Achieving stability in MapReduce requires overcoming several important challenges: (a) making a small change to the input can change the input to many tasks, ultimately leading to a large number of unmatched tasks; (b) even if a small number of tasks is affected, the tasks themselves can require a long time to execute. To solve these problems, we propose techniques for (1) performing a stable partitioning of the input; (2) controlling the granularity and stability of the Map and Reduce tasks; and (3) finding efficient scheduling mechanisms taking into account the location of results that can be reused. We briefly summarize our design decisions below.

Stable input partitions. To see why the standard MapReduce approach to input partitioning leads to unstable computations, consider inserting a single record in the beginning of an input file. Since the input is partitioned into fixed-sized chunks by HDFS, this small change will shift each partition point by one record, effectively changing the input to each map task. In general, when the record is inserted at some position, all chunks that follow that position will have to shift by one, and thus on average nearly half of all tasks will be unmatched. The problem only gets more complicated as we allow more complex changes, where for example the order of records may be permuted; such changes can be common, for instance, if a crawler uses a depth-first strategy to crawl the web, and a single link changing can move an entire subtree's position in the input file. One possible solution to this problem would be to compute the differences between the two inputs files and somehow update the computation by using this difference directly. This would, however, require running a polynomial-time algorithm (e.g., an edit-distance algorithm) to find the difference.



(a) Fixed-size chunking in HDFS (b) Content chunking in Inc-HDFS (c) Example of stable partitioning

Figure 2: File chunking strategies in HDFS and Inc-HDFS

We use a stable partitioning technique that enables maximal overlap between the set of data chunks created with similar inputs. Maximizing the overlap between data-chunks in turn enables maximizing the number of matched Map tasks. To support stable partitioning we propose a file system, called Inc-HDFS, that we describe in Section 3.

Granularity control. We maximize the overlap between Map tasks by using a stable partitioning technique for creating the input data chunks. The input to the Reduce tasks, however, is directly determined by the outputs of the Map tasks, because the key-value pairs with the same key are processed together by the same Reduce task. This raises a problem if, for example, a single new key-value pair is added to a Reduce task that processes a large number of values, which forces the entire task to be recomputed. Furthermore, even if we found a way of dividing large Reduce tasks into multiple smaller tasks, this would not solve the problem if such tasks depended on each other, like what would happen if the input of each task depended on the output of the previous one. Thus, we need a way to reduce not only the task size but also eliminate potentially long (possibly linear-size) dependencies between parts of the Reduce tasks. In other words, we need to control granularity without increasing the number of unmatched tasks.

We solve this problem by performing an additional Contraction phase, where Reduce tasks are combined hierarchically in a tree-like fashion; this both controls the task size and ensures that no long dependencies between tasks arise, since all paths in the tree will be of logarithmic length. Section 4 describes our proposed approach.

Scheduling. To enable efficient reuse of matched sub-computations, it is important to schedule a task on the machine that stores the memoized results to be reused. We achieve this by extending the scheduling algorithm used by Hadoop with a notion of *affinity*. In this scheme the scheduler takes into account affinities between machines and tasks by keeping a record of which nodes have executed which tasks. This enables us to minimize the movement of memoized intermediate results, but at the cost of a potential degradation of job performance. This is because it increases the chances of introducing stragglers [22], since a strict affinity of tasks results in deterministic scheduling and prevents a lightly loaded node from stealing work from the task queue of a slow node. We therefore propose a hybrid scheduling policy that strikes a balance between work-stealing and affinity to the memoized results. Section 5 provides a detailed description of the modified scheduler.

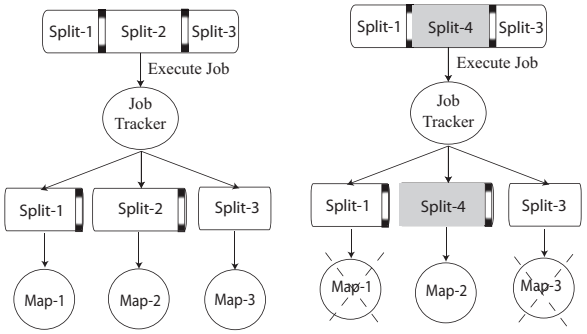
3. INCREMENTAL HDFS

We propose Incremental HDFS (Inc-HDFS), a distributed file system that assists Incoop in performing incremental computations efficiently. Inc-HDFS extends the Hadoop distributed file system (HDFS) to enable stable partitioning of the input via content-based chunking, which was introduced in LBFS [16] for data deduplication. At a high-level, content-based chunking defines chunk boundaries only based on the contents of input, instead of fixed-size chunks as provided by HDFS. As a result, insertions and deletions cause minimal changes to the set of chunks and hence the inputs to MapReduce tasks remain stable, i.e., similar to those of the previous run. Figure 2 illustrates a comparison of the chunking strategies of standard HDFS and Inc-HDFS.

To perform content-based chunking we scan the entire file using a fixed-width sliding window. For each file position, we read the window contents, and compute its Rabin fingerprint. Then if the fingerprint matches a certain pattern, which we term a *marker*, we place a chunk boundary at that position. A limitation of this approach is that it may create chunks that are too small or too large, given that markers will not be evenly spaced, and that the chunk size depends solely on the input (and only the average size can be controlled by the system). This variance in the chunk size for tasks may degrade overall job performance. To address this, we constrain the minimum and maximum chunk sizes. Thus, after we find a marker m_i at position p_i, we skip a fixed *offset* O in the input sequence and continue to search for a marker starting at position $p_i + O$. In addition, we bound the chunk length by setting a marker after M content bytes when no marker was found before that. Despite the possibility of affecting stability by either missing important markers due to skipping the initial offset, or consecutively using the maximum length in an unaligned manner, we found this scheme to work well in practice. This was because such occurrences were very rare and only had a minor impact on performance.

Chunking could be performed either during the creation of the input or when the input is read by the Map task. We chose the former approach for two main reasons. First, the additional cost for chunking can be amortized when the chunking and the actual generation of the input data can be performed in parallel. This is particularly advantageous when the process for input data generation is not limited by the storage throughput. The second reason is that when the input is first written to HDFS, it is already present in the main memory of the node that writes the input, and hence this node can perform the chunking without additional accesses to the data.

In order to leverage the common availability of multicores during the chunking process, we parallelized the search for markers in the input data. Our implementation uses multiple threads that each search for markers in the input starting at different positions. The markers that are found cannot be used immediately to define the chunk boundaries, since some



(a) First run (b) Incremental run

Figure 3: Incremental Map tasks

of them might have to be skipped due to the minimum chunk size requirement. Instead, we collect the markers in a list, and iterate over the list to determine the markers that are skipped and those that define the actual chunk boundaries. Our experimental evaluation (Section 7.4) highlights the importance of these optimizations in keeping the performance of Inc-HDFS very close to that of its HDFS counterpart.

4. INCREMENTAL MAPREDUCE

We describe the incremental MapReduce part of the infrastructure by separately discussing how Map and Reduce tasks handle incremental inputs.

Incremental Map. Given that Inc-HDFS already provides the necessary control over the alignment and granularity of the input chunks that are provided to Map tasks, the job of the incremental Map tasks becomes simplified, since they can implement task-level memoization without having to worry about finding opportunities for reusing previous results at a finer granularity, using another alignment, or at other locations of the input. Specifically, 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.

Instances of incremental Map tasks take advantage of previously stored results by querying the memoization server. If they find that the result has already been computed, they fetch the result from the node storing it, and conclude. Figure 3 illustrates this: part (a) describes the first run of an Incoop job and part (b) describes the subsequent run where split 2 is modified (hence replaced by split 4) and the Map tasks for splits 1 and 3 need not be re-executed.

Incremental Reduce. The Reduce function processes the output of the Map function grouped by the keys of the generated key-value pairs. For a subset of all keys, each Reduce retrieves the key-value pairs generated by all Map tasks and applies the Reduce function. For efficiency, we perform memoization of Reduce tasks at two levels: first as a coarse-grained memoization of entire Reduce tasks, and second as a fine-grained memoization of sub-computations of a novel Contraction phase as described below.

As with Map tasks, we remember the results of a Reduce task by persistently storing them and inserting a mapping from hash of the input to the location of the results in the memoization server. Since a Reduce task possibly receives input from the n Map tasks, the key of that mapping consists of collision-resistant hashes of the outputs from all n Map tasks that collectively form the input to the Reduce task. When executing a Reduce task, instead of immediately copying the output from the n Map tasks, the Reduce task retrieves the output hashes from these Map tasks to determine if the Reduce task has already been computed in a previous run. If so, the output is directly fetched from the location stored in the memoization server, which avoids the re-execution of that task.

This task-level memoization has a crucial limitation: small changes in the input can cause a full – and potentially large – Reduce task to be re-executed, which results in inefficient incremental updates. Furthermore, the larger the task, the more likely that its input will include some changed data, and thus the less likely that the task output will be re-used. Since each Reduce task processes all values that are produced for a given key and since the number of such values depends only on the computation and its input, we cannot control the size of the input to Reduce tasks under the current model. This ultimately hinders stability, and we therefore need a way to decrease the granularity of Reduce tasks. We must do so while avoiding creating a long dependence chain between the smaller tasks — such dependencies will force the execution of many subtasks, ultimately failing to achieve the initial goal.

To reduce the granularity of Reduce tasks effectively, we propose a new *Contraction Phase*, which is run by Reduce tasks. To this end, we take advantage of *Combiners*, a feature of the original MapReduce and Hadoop frameworks [8], which was designed for a completely different purpose. Combiners are meant to save bandwidth by offloading part of the computation performed by the Reduce task to the Map task. With this mechanism, the programmer specifies a separate Combiner function, which is executed on the machine that runs the Map task, and pre-processes various ⟨key,value⟩ pairs, merging them into a smaller number of pairs. The combiner function takes as input an argument of the same type as its output: a sequence of ⟨key,value⟩ pairs. Notably, Combiners and Reducers often perform very similar work.

We use Combiners to break up large Reduce tasks into many applications of the Combine function, which allows us to perform memoization at a much finer granularity. More precisely, we split the Reduce input into chunks, and apply the Combine function to each chunk. Then we again form chunks from the aggregate result of all the Combine invocations and recursively apply the Combine function to these new chunks. The data size gets smaller in each iteration, and finally, we apply the Reduce function to the output of the last level of Combiners. This approach enables us to memoize the results of the Combiners and therefore, when the input to the Contraction phase is changed, only a subset of the Combiners have to be re-executed rather than a full Reduce task.

This new usage of Combiners is syntactically compatible with the original Combiner interface, since both input and output of Combiners is a set of tuples that can be passed to the Reduce task. However, semantically, Combiners were only designed to run at most once, and therefore the correctness of the MapReduce computation is only required to be maintained across 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 usage of Combiner functions requires a slightly different property:

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

Even though it is theoretically possible to write a Combiner that meets the original requirement but not the new one, in practice, all of the Combiner functions we have analyzed obey the new requirement.

Stability of the Contraction phase. An important design question is how to partition the input to the Contraction phase into chunks that are processed by different Combiners. In this case, the same issue that arose at the Map phase needs to be handled: 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 undermine the stability of the dependence graph, since a small change could cause a large re-computation. This problem is illustrated in Figure 4, which shows two consecutive runs of the same Reduce task, where a new map task (#2) is added to the set of map tasks that produce values associated with the key being processed by this Reduce task. In this case, a simple partitioning of the input, e.g., into groups with a fixed number of input files, would cause all groups of files to become different from one run to the next, due to the insertion of one new file near the beginning of the sequence.

To solve this, we again rely on content-based chunking, and apply it 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 Mappers, we require chunk boundaries to be at file boundaries, i.e., chunking can only group entire files. This way we leverage the existing partitioning of the input, which simplifies the implementation and avoids re-processing this input: we use the hash of each input file to determine if a marker is present, i.e., if that input file should be the last of a set of files that is given to a single Combiner. In particular, we test if the hash modulo a pre-determined integer M is equal to a constant $k < M$. This way the input file contents do not need to be scanned to partition the input.

Figure 4 also illustrates how content-based chunking obviates the alignment problem. In this example, the content-based marker that delimits the boundaries between groups of input files is present in outputs #5, 7, and 14, but not the remaining ones. Therefore, inserting a new map output will change the first group of inputs but none of the remaining ones. In this figure we can also see how this change propagates to the final output. In particular, this change will lead to executing a new Combiner (labelled 1-2-3-5), and the final Reducer. The results for all of the remaining Combiners are reused without needing to re-execute them. This technique is then repeated across all levels of the tree.

5. MEMOIZATION-AWARE SCHEDULER

The Hadoop scheduler assigns Map and Reduce tasks to nodes for efficient execution, taking into account machine availability, cluster topology, and the locality of input data. The Hadoop scheduler, however, is not well-suited for incremental computations because it does not consider the locality of memoized results.

To enable efficient re-use of previously computed intermediate results, tasks should preferentially be scheduled on the node where some or all of the memoized results they use are stored. This is important, for instance, in case the Contraction phase needs to run using a combination of newly computed and memoized results, which happens when only a part of its inputs has changed. In addition to this design goal, the scheduler also has to 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 drastically delay the overall job completion [22].

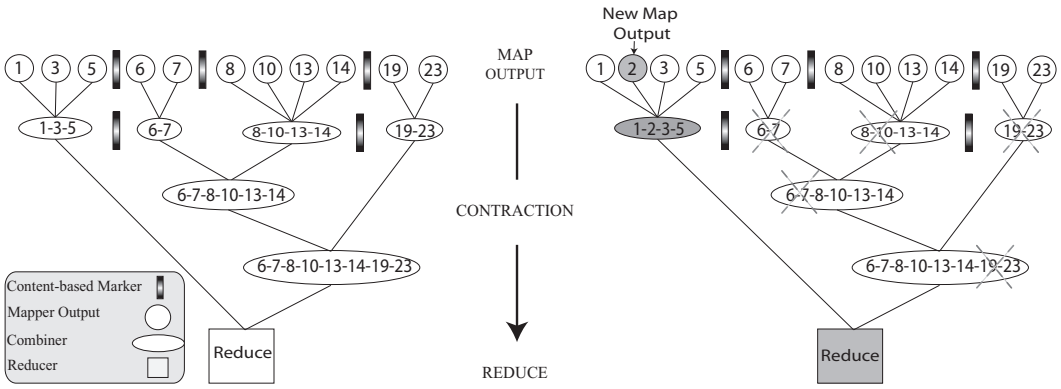
Based on these requirements, Incoop includes a new memoization-aware scheduler that strikes a balance between exploiting the locality of memoized results and incorporating some flexibility to minimize the straggler effect. The scheduler tries to implement a location-aware policy that prevents the unnecessary movement of data, but at the same time it implements a simple work-stealing algorithm to adapt to varying resource availability. The scheduler works by maintaining a separate task queue for each node in the cluster (instead of a single task queue for all nodes), where each queue contains the tasks that should run on that node in order to maximally 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 corresponding queue for the requesting node is empty, the scheduler tries to *steal* work from other task queues. The scheduling algorithm searches the task queues of other nodes, and steals a pending task from the task queue with maximum length. If there are multiple queues of maximum length, the scheduler steals the task that has the least amount of memoized intermediate results. Our scheduler thus takes the location of the memoized results into account, but falls back to a work-stealing approach to avoid stragglers and nodes running idle. Our experimental evaluation (Section 7.6) shows the effectiveness of our approach.

6. ANALYSIS OF INCOOP

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

For the initial run, we define the *overhead* as the slowdown of Incoop compared to a conventional implementation of MapReduce such as with Hadoop. We show that the overhead depends on communication costs and, if these are independent of the input size, which they often are, then it is also constant. Our experiments (Section 7) confirm that the overhead is relatively small. We show that dynamic updates are dominated by the time it takes to execute fresh tasks that are affected by the changes to the input data, which, for a certain class of computations and small changes, is logarithmic in the size of the input.

In the analysis, we use the following terminology to refer to the three different types of computational tasks that form an Incoop computation: Map tasks, Contraction tasks (applications of the Combiner function in the contraction



(a) First run (c) Incremental run

Figure 4: Stability of the Contraction phase

phase), and Reduce tasks. Since Incoop computes hashes to efficiently check if results can be reused, this computation needs to be factored in the total cost. We write t_h for the time it takes to hash data and t_m for the time it takes to send a short message (one that does not contain entire inputs and outputs of tasks). Our bounds depend on the total number of map tasks, written N_M and the total number of reduce tasks written N_R. We write n_i and n_O to denote the total size of the input and output respectively, n_m to denote the total number of key-value pairs output by the Map phase, and n_{mk} to denote the set of distinct keys emitted by the Map phase.

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

In this section we state only our final theorems, and the complete proofs are contained in Appendix B.

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

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

THEOREM 3 (DYNAMIC UPDATE: SPACE AND TIME). *In Incoop, a dynamic update with fresh tasks F requires time*

$$O\left(t_{memo}\left(N_M + N_C + N_R\right) + \sum_{a \in F} t(a)\right).$$

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

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

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

7. IMPLEMENTATION AND EVALUATION

We evaluate the effectiveness of Incoop for a variety of applications implemented in the traditional MapReduce programming model. In particular, we will answer the following questions:

- How does Incoop's Inc-HDFS performance compare to HDFS?(§7.4)
- What performance benefits does Incoop provide for incremental workloads compared to the unmodified Hadoop implementation? (§7.5)
- How effective are the optimizations we propose in improving the overall performance of Incoop? (§7.6)
- What overheads does the memoization in Incoop impose when tasks are executed for the first time? (§7.7)

7.1 Implementation

We built our prototype of Incoop based on Hadoop-0.20.2. We implemented Inc-HDFS by extending HDFS with stable input partitioning, and incremental MapReduce by extending Hadoop with a finer granuality control mechanism and the memoization-aware scheduler.

The Inc-HDFS file system provides the same semantics and interface for accessing all native HDFS calls. It employs a content-based chunking scheme which is computationally more expensive than the fixed-size chunking used by HDFS. As described in §3, the implementation minimizes the overhead using two optimizations: (i) we skip parts of the file contents when searching for chunk markers, in order to reduce the number of fingerprint computations and enforce a minimum chunk size; and (ii) we parallelize the search for

Application	Description
K-Means Clustering (`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.
k-NN Classifier (`KNN`)	k-nearest neighbors classifies objects based on the closest training examples in a feature space.
Co-occurrence Matrix (`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.
Bigram-Count (`BiCount`)	Bigram count measures the prevalence of each subsequence of two items within a given sequence.

Table 1: Applications used in the performance evaluation

markers across multiple cores. To implement these optimizations, the data uploader client skips a fixed number of bytes after the last marker is found, and then spawns multiple threads that each compute the Rabin fingerprints over a sliding window on different parts of the content. For our experiments, we set the number of bytes skipped to 40MB unless otherwise stated.

We implemented the memoization server using a wrapper around Memcached v1.4.5, which provides an in-memory key-value store. Memcached runs as a daemon process on the name node machine that acts as a directory server in Hadoop. Intermediate results memoized across runs are stored on Inc-HDFS with the replication factor set to 1, and, in case of data loss, the intermediate results are recomputed. A major issue with any implementation of memoization is determining which intermediate results to remember and which intermediate results to purge. As in self-adjusting computation approaches, our approach is to cache the *fresh* results from the "last run", i.e., those results that were generated or used by the last execution, and purge all the other *obsolete* results. This suffices to obtain the efficiency improvements shown in §7.7. We implement this strategy using a simple garbage collector that visits all cache entries and purges the obsolete results.

Finally, the Contraction phase is implemented by aggregating all keys that are processed by each node, instead of at a per-key granularity, since this is closer to the original Hadoop implementation.

7.2 Applications and Data Generation

For the experimental evaluation, we use a set of applications in the fields of machine learning, natural language processing, pattern recognition, and document analysis. Table 1 lists these applications. We chose these applications to demonstrate Incoop's ability to efficiently execute both data intensive (`WordCount, Co-Matrix, BiCount`), and computation intensive (`KNN` and `K-Means`) jobs. In all cases, we did not make any changes to the original code.

The three data-intensive applications take as input documents written in a natural language. In our benchmarks, we use a publicly available dataset with the contents of Wikipedia.[1] The computation-intensive applications take as input a set of points in a d-dimensional space. We generate this data synthetically by uniformly randomly selecting points from a 50-dimensional unit cube. To ensure reasonable running times, we chose all the input sizes such that the running time of each job would be around one hour.

7.3 Measurements

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

Work and (parallel) time. For comparing different runs, we consider two types of measures, work and time, which are standard measures for comparing efficiency in parallel applications. *Work* refers to the total amount of computation performed by all tasks and measured as the total running time of all tasks. *(Parallel) Time* refers to the amount of (end-to-end) time that it takes to complete a parallel computation. It is well-known that under certain assumptions a computation with W work can be executed on P processors in $\frac{W}{P}$ time plus some scheduling overheads; this is sometimes called the work-time principle. Improvements in total work often directly lead to improvements in time but also in the consumption of other resources, e.g., processors, power, etc. As we describe in our experiments, our approach reduces work by avoiding unnecessary computations, which translates to improvements in run-time (and use of other resources).

Initial run and dynamic update. The most important measurements we perform involve the comparison of the execution of a MapReduce job with Hadoop vs. with Incoop. For the Incoop measurements, we consider two different runs. The *initial run* refers to a run starting with an empty memoization server that has no memoized results. Such a run executes all tasks and populates the memoization server by storing the performed computations and the location of their results. The *dynamic update* refers to a run of the same job with a modified input, but that happens after the initial run, avoiding recomputation when possible.

Speedup. To assess the effectiveness of dynamic updates, we measure the work and time after modifying varying percentages of the input data and comparing them to those for performing the same computation with Hadoop. We refer to the ratio of the Hadoop run to the incremental run (Incoop dynamic update) as *speedup* (in work and in time). When modifying p% of the input data, we randomly chose p% of the input chunks and replaced them with new chunks of equal size and newly generated content.

Hardware. Our measurements were gathered using a cluster of 20 machines, running Linux with kernel 2.6.32 in 64-bit mode, connected with gigabit ethernet. The name node and the job tracker ran on a master machine which was equipped with a 12-core Intel Xeon processor and 12 GB of RAM. The data nodes and task trackers ran on the remaining 19 machines equipped with AMD Opteron-252 processors, 4GB of RAM, and 225GB drives. We configured the task trackers to use two Map and two Reduce slots per worker machine.

7.4 Incremental HDFS

To evaluate the overhead introduced by the content-based chunking in Inc-HDFS, we compare the throughput when

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

uploading a dataset of 3 GB for HDFS and Inc-HDFS. In HDFS the chunk size is fixed at 64 MB, while in Inc-HDFS, we vary the number of skipped bytes. The uploading client machine was co-located with the name node of the cluster, and we configured the parallel chunking code in Inc-HDFS to use 12 threads, i.e., one thread per core. The results of the experiments, shown in Table 2, illustrate the effectiveness of our performance optimizations. Compared to plain HDFS, Inc-HDFS introduces only a minor throughput reduction due to the fingerprint computation that is required for content-based chunking.

For the smallest skip offset of 20MB, Inc-HDFS introduces a more noticeable overhead because Rabin fingerprints are computed for a larger fraction of the data, which results in a reduction of overall throughput. When using the largest skip offset of 60MB, we again see a small throughput reduction, despite the smaller computational overhead for fingerprinting. This is due to the fact that a larger skip offset increases the average chunk size, resulting in a lower total number of chunks for an input file. As a consequence, less work can be done in parallel towards the end of the upload. For a skip offset of 40MB, however, the Inc-HDFS throughput is similar to HDFS because it strikes a balance between the computational overhead of fingerprint computations and opportunities for parallel processing of data blocks during the upload to the distributed file system.

7.5 Run-time Speedups

Figure 5 and Figure 6 show the work and time speedups, which are computed as ratio between the work and time of a dynamic run using Incoop and those of Hadoop. From these experimental results we can observe the following: (*i*) Incoop achieves substantial performance gains for all applications when there are incremental changes to the input data. In particular, work and time speedups vary between 3-fold and 1000-fold for incremental modifications ranging from 0% to 25% of data. (*ii*) We observe higher speedups for computation-intensive applications (`K-Means, KNN`) than for data-intensive applications (`WordCount, Co-Matrix, BiCount`). This is consistent with the approach (and the analysis in Appendix B), because for computation-intensive tasks the approach avoids unnecessary computations by re-using results. (*iii*) Both work and time speedups decrease as the size of the incremental change increases, because larger changes allow fewer computation results from previous runs to be re-used. With very small changes, however, speedups in total work are not fully translated into speedups in parallel time; this is expected because decreasing the total amount of work dramatically (e.g., by a factor 1000) reduces the amount of parallelism, causing the scheduling overheads to be larger. As the size of the incremental change increases, the gap between the work speedup and time speedup closes quickly.

The previous examples all consider fixed-size inputs. We

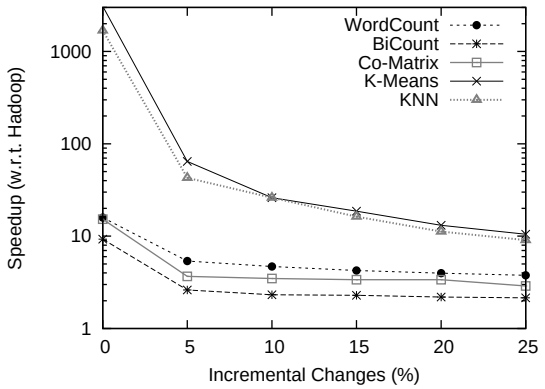


Figure 5: Work speedups versus change size.

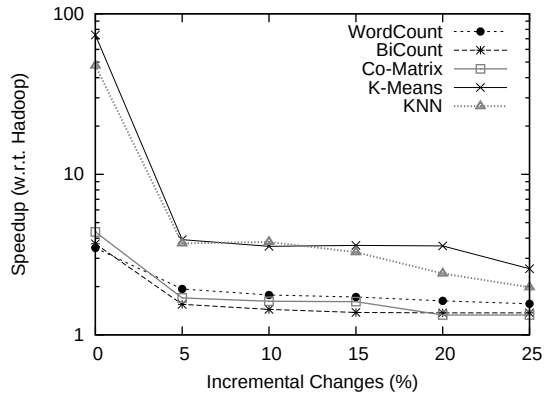


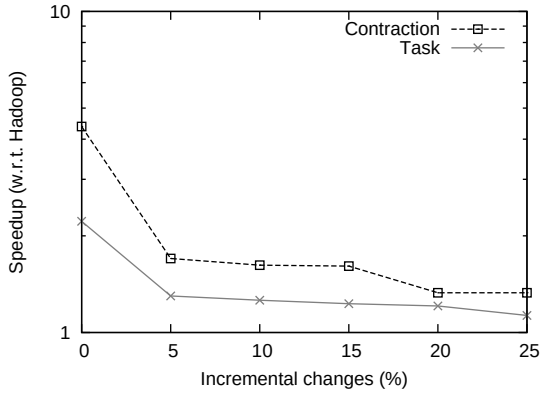
Figure 6: Time speedups versus change size.

experimented with other input sizes, and similar results hold. This is shown in Figure 7, which illustrates the time to run Incoop and Hadoop using the Co-Matrix application, and for a modification of a single chunk. This figure shows that the relative improvements hold for various different input sizes.

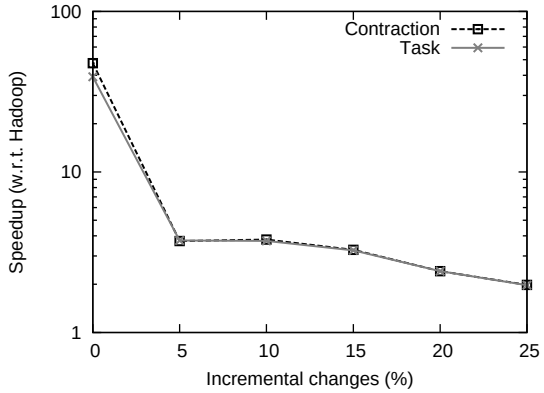
7.6 Effectiveness of Optimizations

We evaluate the effectiveness of the optimizations in improving the overall performance of Incoop by considering (*i*) the granularity control with the introduction of the Contraction phase; and (*ii*) the scheduler modifications to minimize unnecessary data movement.

Granularity control. To evaluate the effectiveness of the Contraction phase, we consider the two different levels of memoization in Incoop: (*i*) the coarse-grained, task-level memoization performed in the implementation, denoted as `Task`, and (*ii*) the fine-grained approach that adds the Contraction phase in the implementation, denoted as `Contraction`. Figure 8 shows our time measurements with `CoMatrix` as a data-intensive application and `KNN` as a computation-intensive application. The effect of the Contraction phase is negligible with `KNN` but significant in `CoMatrix`. The reason for negligible improvements with `KNN` is that in this application, Reduce tasks perform relatively inexpensive work and thus benefit little from the Contraction phase. Thus, even when not helpful, the Contraction phase does not degrade efficiency.



(a) Co-occurrence Matrix



(b) k-NN Classifier

Figure 8: Performance gains comparison between `Contraction` and `task` variants

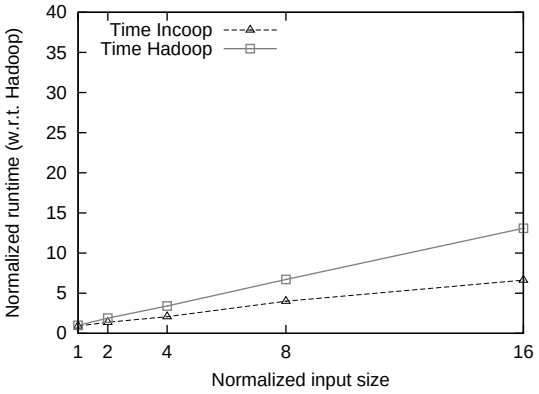


Figure 7: `Co-Matrix`: Time versus input size

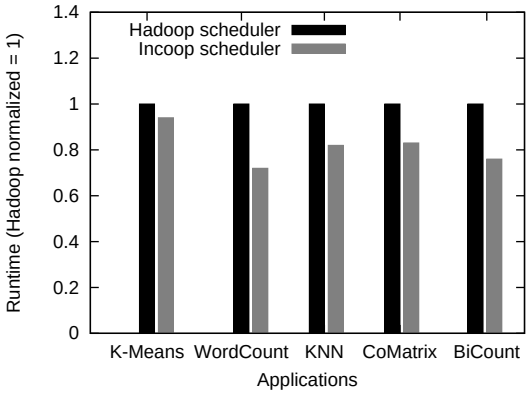


Figure 9: Effectiveness of scheduler optimizations.

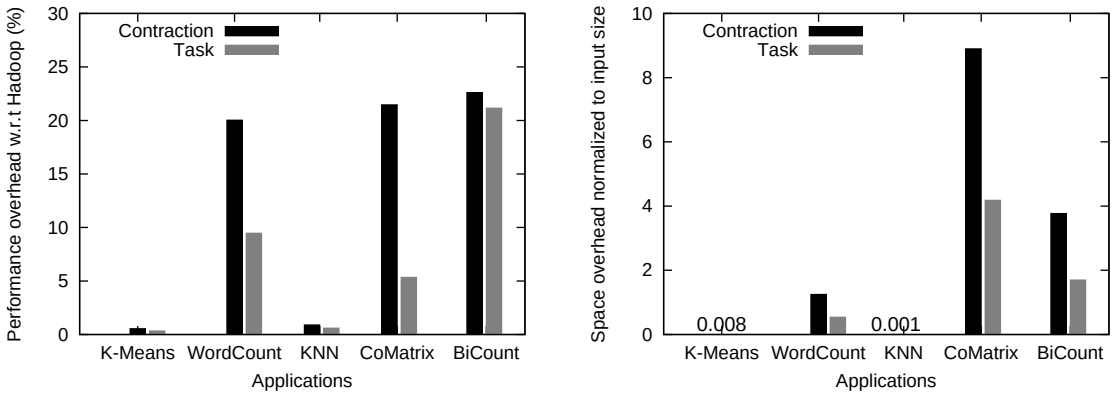
Scheduler modification. We now evaluate the effectiveness of the scheduler modifications in improving the performance of Incoop. The Incoop scheduler avoids unnecessary data movement by scheduling tasks on the nodes where intermediate results from previous runs are stored. Also, the scheduler employs a work stealing algorithm that allows some task scheduling flexibility to prevent nodes from running idle when runnable tasks are waiting. We show the performance comparison of the Hadoop scheduler with the Incoop scheduler in Figure 9, where the Y-axis shows runtime relative to the Hadoop scheduler. The Incoop scheduler saves around 30% of time for data-intensive applications, and almost 15% of time for compute-intensive applications, which supports the necessity and effectiveness of location-aware scheduling for memoization.

7.7 Overheads

The memoization performed in Incoop introduces runtime overheads for the initial run when no results from previous runs can be reused. Also, memoizing intermediate task results imposes an additional space usage. We measured both types, performance and space overhead, for each application and present the results in Figure 10.

Performance overhead. We measure the worst-case performance overhead by capturing the runtime for the initial run. Figure 10(a) depicts the performance penalty for both the `Task` and the `Contraction` memoization based approach. The overhead varies from $5\% - 22\%$, and, as expected, it is lower for computation intensive applications such as `K-Means` and `KNN`, since their run-time is dominated by the actual processing time rather than storing, retrieving and transferring data. For the data intensive applications such as `WordCount, Co-Matrix` and `BiCount`, the first run with `Task` level memoization is faster than `Contraction` memoization. This difference in performance can be attributed to the extra processing overheads for all levels of the tree formed in the Contraction phase. Importantly, this performance overhead is a one-time cost and the subsequent runs benefit from a high speedup.

Space overhead. We measure the space overhead by quantifying the space used for remembering the intermediate computation results. Figure 10(b) illustrates the space overhead as a factor of the input size with Task- and Contraction-level memoization. The results show that the Contraction-level memoization requires more space, which was expected because it stores results for all levels of the Contraction tree. Overall, space overhead varies substantially depending on the application, and can be as high as $9X$ (`CoMatrix` application). However, our approach for garbage collection prevents the storage utilization from growing over time.



(a) Performance overhead for the first job run

(b) Space overhead

Figure 10: Overheads imposed by `Incoop` in comparison to `Hadoop`

8. RELATED WORK

Our work builds on contributions from several different fields, which we briefly survey.

Dynamic algorithms. In the algorithms community, researchers designed dynamic algorithms that permit modifications or dynamic changes to their input, and efficiently update their output when such changes occur. Several surveys discuss the vast literature on dynamic algorithms [9] This research shows that dynamic algorithms can be asymptotically, often by a near-linear factor, more efficient than their conventional counterparts. In large-scale systems, this asymptotic difference can yield significant speedups. Dynamic algorithms can, however, be difficult to develop and implement even for simple problems; some problems took years of research to solve and many remain open.

Programming language-based approaches. In the programming languages community, researchers developed incremental computation techniques to achieve automatic incrementalization (e.g. [21]). This approach is automatic and hides the mechanism for incrementalization, and can thus dramatically simplify software development. Recent advances on self-adjusting computation made significant progress on the problem of incremental computation by proposing general-purpose techniques that can achieve optimal update times (e.g., [2, 1, 12]). Our work uses ideas and techniques from self-adjusting computation to ensure efficiency, e.g., the ideas of stable input partitions and stable contraction trees and the techniques for achieving these. Algorithms for parallel self-adjusting computation have also been proposed [11] but they have not been implemented; most work on self-adjusting computation assumes sequential execution.

Incremental database view maintenance. There is substantial work from the database community on incrementally updating a database view (i.e., a predetermined query on the database) as the database contents evolve. The techniques used by these systems can either directly operate on the database internals to perform these incremental updates, or rely on SQL queries that efficiently compute the modifications to the database view, and that are issued upon the execution of a database trigger [5]. Even though Incoop shares the same goals and principles as incremental view maintenance, it differs substantially in the techniques that are employed, since the latter exploits database-specific mechanisms and semantics.

Large-scale incremental parallel data processing. There are several systems for performing incremental parallel computations with large data sets. We broadly divide them into two categories: non-transparent and transparent approaches. Examples of non-transparent systems include Google's Percolator [19] which requires the programmer to write a program in an event-driven programming model based on observers. Observers are triggered by the system whenever user-specified data changes.Similarly, continuous bulk processing (CBP) [14] proposes a new data-parallel programming model,which offers primitives to store and reuse prior state for incremental processing. There are two drawbacks to these approaches, both of which are addressed by our proposal. The first is that they depart from the MapReduce programming paradigm and therefore require changes to the large existing base of MapReduce programs. The second, more fundamental problem is that they require that programmer to devise a dynamic algorithm in order to efficiently process data in an incremental manner.

Examples of transparent approaches include DryadInc [20], which extends Dryad to automatically identify redundant computations by caching previously executed tasks. One limitation of this basic approach is that it can only reuse common identical sub-DAGs of the original computation, which can be insufficient to achieve efficient updates. To improve efficiency the paper suggests the programmers specify additional merge functions. Another similar system called Nectar [10] caches prior results at the coarser granularity of entire LINQ sub-expressions. The technique used to achieve this is to automatically rewrite LINQ programs to facilitate caching. Finally, although not fully transparent, Haloop [4] provides task-level memoization techniques for memoization in the context of iterative data processing applications. The major difference between the aforementioned transparent approaches and our proposal is that we use a well-understood set of principles from related work to eliminate the cases where task-level memoization provides poor efficiency. To this end, we provide techniques for increasing the effectiveness of task-level memoization via stable input partitions and by using a more fine-grained memoization strategy than the granularity of Map and Reduce tasks. Our implemen-

tation does not support currently incremental execution for iterative computations; however, the framework can easily be extended to support iterative computations.

Our own short position paper [3] 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 [12], and does not offer anything close to a full-scale distributed design and implementation such as we describe here.

The Hadoop online prototype (HOP) [7] extends the Hadoop framework to support pipelining between the map and reduce tasks, so that reducers start processing data as soon as it is produced by mappers. This enables two new features in the framework. First, it can generate an approximate answer before the end of the computation (online aggregation) and second, it can support continuous queries, where jobs run continuously, and process new data as it arrives.

Stream processing systems. Comet [13] introduces the Batched Stream Processing (BSP), where input data is modeled as a stream, with queries being triggered upon bulk appends to the stream. The interface provided by Comet enables exploiting temporal and spatial correlations in recurring computations by defining the notion of a query series. Within a query series, the execution will automatically leverage the intermediate results of previous invocations of the same query on an overlapping window of the data, thereby exploiting temporal correlations. Further, by aligning multiple query series to execute together when new bulk updates occur, Comet exploits spatial correlations by removing redundant I/O or computation across queries. 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 [17] is a workflow manager recently proposed by Yahoo!, designed for the incremental execution of Pig programs upon continually-arriving data. NOVA introduces a new layer called the workflow manager on top of the Pig/Hadoop framework. Much like the work on incremental view maintenance, the workflow manager rewrites the computation to identify the parts of the computation affected by incremental changes and produce the necessary update function that 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. In our work, and particularly with our case study of incremental processing of Pig queries, we explore precisely this alternative design of adding lower-level support for reusing previous results. Furthermore, our work is broader in that it transparently benefits all MapReduce computations, and not only continuous Pig queries.

9. CONCLUSION

In this paper, we presented Incoop, a novel MapReduce framework for large-scale incremental computations. Incoop is based on several novel techniques to maximize the re-use of results from a previous computation. In particular, Incoop incorporates content-based chunking to the file system to detect incremental changes in the input file and to partition the data so as to maximize re-use; it adds a contraction phase to control the granularity of tasks in the reduce phase, and a new scheduler that takes the location of previously computed results into account. We implemented Incoop as an extension to Hadoop. Our performance evaluation shows that Incoop can improve efficiency in incremental runs (the common case), at a modest cost in the initial, first run (uncommon case) where no computations can be reused.

Acknowledgments

We appreciate the detailed and helpful feedback from Remzi Arpaci-Dusseau, Jens Dittrich, and the Sysnets group members at MPI-SWS. We would like to thank Daniel Porto for helping us to set up the PigMix experiments.

10. REFERENCES

[1] 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.

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

[3] 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)*.

[4] 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.

[5] 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.

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

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proc. 7th Symposium on Networked systems design and implementation (NSDI'10)*.

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

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

[10] 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)*.

[11] 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.

[12] 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.

[13] 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)*.

[14] 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)*.

[15] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ mapreduce for log processing. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, 2011.

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

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

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[19] 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)*.

[20] 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)*.

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

[22] 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.

APPENDIX

A. CASE STUDIES

The success of MapReduce paradigm enables our approach to transparently benefit an enormous variety of bulk data processing workflows. In particular, and aside from the large number of existing MapReduce programs, MapReduce is also being used as an execution engine for other systems. In this case, Incoop will also transparently benefit programs written for these systems.

In this section, we showcase two workflows where we use Incoop to transparently benefit systems from efficient incremental processing in their context, namely *incremental log processing* and *incremental query processing*.

A.1 Incremental Log Processing

Log processing is an essential workflow in Internet companies, where various logs are often analyzed in multiple ways on a daily basis [15]. For example, in the area of click log analysis, traces collected from various web server logs are aggregated in a single repository and then processed for various purposes, from simple statistics like counting clicks per user, or more complex analyses like click sessionization.

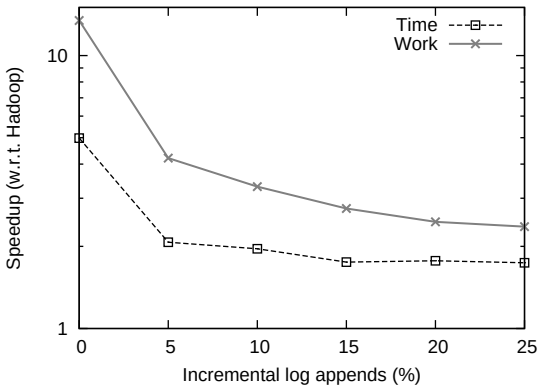


Figure 11: Speedup for incremental log processing

To perform incremental log processing, we integrated Incoop with Apache Flume [2] – a distributed and reliable service for efficiently collecting, aggregating, and moving large amounts of log data. In our setup, Flume aggregates the data and dumps it into the Inc-HDFS repository. Then, Incoop performs the analytic processing incrementally, leveraging previously computed intermediate results.

We evaluate the performance of using Flume in conjunction with Incoop for incremental log processing by comparing its runtime with the corresponding runtime when using Hadoop. For this experiment, we perform document analysis on an initial set of logs, and then append new log entries to the input, after which we process the resulting larger collection of logs incrementally. In Figure 11, we depict the speedup for running Incoop as a function of the size of the new logs that are appended after the first run. Incoop achieves a speedup of a factor of 4 to 2.5 with respect to Hadoop when processing incremental log appends of a size of 5% to 25% of the initial input size, respectively.

A.2 Incremental Query Processing

We showcase incremental query processing as another workflow that exemplifies the potential benefits of Incoop. Incremental query processing is an important workflow in Internet companies, where the same query is processed frequently for an incrementally changing input data set [17]. We integrated Incoop with Pig to evaluate the feasibility of incremental query processing. Pig [18] is a platform to analyze large data sets built upon Hadoop. Pig provides Pig Latin, an easy-to-use high-level query language similar to SQL. The ease of programming and scalability of Pig made the system very popular for very large data analysis tasks, which are conducted by major Internet companies today.

Since Pig programs are compiled down to multi-staged MapReduce jobs, the integration of Incoop with Pig was seamless, just by using Incoop as the underlying execution engine for incrementally executing the multi-staged MapReduce jobs. We evaluate two Pig applications, word count and the PigMix[3] scalability benchmark, to measure the effectiveness of Incoop. We observe a runtime overhead of around

[2]Apache Flume: `https://github.com/cloudera/flume`
[3]Apache PigMix: `http://wiki.apache.org/pig/PigMix`

15% for first run, and a speedup of a factor of around 3 for an incremental run with unmodified input. The detailed result breakdown is shown in Table 3.

Application	Features	M/R stages	Overhead	Speedup
Word Count	Group_by, Order_by, Filter	3	15.65 %	2.84
PigMix scalabilty benchmark	Group_by, Filter	1	14.5 %	3.33

Table 3: Results for incremental query processing

B. ANALYSIS OF INCOOP (PROOFS)

THEOREM 5 (INITIAL RUN:TIME AND OVERHEAD). *Assuming that Map, Combine, and Reduce functions take time asymptotically linear in their input size and that Combine functions are monotonic, total time for performing an incremental MapReduce computation in Incoop with an input of size n_i, where $n_m$key-value pairs are emitted by the Map phase is $O(t_{memo} \cdot (N_M + N_R + N_C)) = O(t_{memo} \cdot (n_i + n_m))$. This results in an overhead of $O(t_{memo}) = O(t_h + t_m)$ over conventional MapReduce.*

PROOF. Memoizing a task requires 1) computing the hash of each input, and 2) sending a message to the memoization server containing the triple consisting of the task id, the input hash and the location of the computed result. Given the time for hashing an input chunk t_h and the time for sending a message t_m, this requires $t_{memo} = t_h + t_m \in O(t_h + t_m)$ time for each task of the job. Memoization therefore causes $O(t_h + t_m)$ per task slowdown. To compute the total slowdown we bound the number of tasks.

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

THEOREM 6 (INITIAL RUN: SPACE). *Total storage space for performing an Incoop computation with an input of size n_i, where $n_m$key-value pairs are emitted by the Map phase, and where Combine is monotonic is $O(n_i + n_m + n_O)$.*

PROOF. In addition to the input and the output, Incremental MapReduce requires additionally storing the output of the map, contraction, and reduce tasks. Since Incoop only keeps data from the most recent run (initial or dynamic run), we use storage for remembering only the task output from the most recent run. The output size of the map tasks is bounded by n_m. With monotonic Combine functions, the size of the output of Combine tasks is bounded by $O(n_m)$. Finally, the storage needed for reduce tasks is bounded by the size of the output. □

THEOREM 7 (DYNAMIC UPDATE: SPACE AND TIME). *In Incoop, a dynamic update requires time*

$$O\left(t_{memo}\left(N_M + N_C + N_R\right) + \sum_{a \in F} t(a)\right).$$

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

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

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

THEOREM 8 (NUMBER OF FRESH TASKS). *If the Map function generates k key-value pairs from a single input record, and the Combine function is monotonic, then the number of fresh tasks is at most $O(k \log n_m + k)$.*

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