# Adaptive MapReduce using Situation-Aware Mappers

Rares Vernica*
Hewlett-Packard Laboratories
Palo Alto, CA, USA
rares.vernica@hp.com

Andrey Balmin    Kevin S. Beyer    Vuk Ercegovac
IBM Almaden Research Center
San Jose, CA, USA
{abalmin,kbeyer,vercego}@us.ibm.com

## ABSTRACT

We propose new adaptive runtime techniques for MapReduce that improve performance and simplify job tuning. We implement these techniques by breaking a key assumption of MapReduce that mappers run in isolation. Instead, our mappers communicate through a distributed meta-data store and are aware of the global state of the job. However, we still preserve the fault-tolerance, scalability, and programming API of MapReduce. We utilize these "situation-aware mappers" to develop a set of techniques that make MapReduce more dynamic: (a) Adaptive Mappers dynamically take multiple data partitions (splits) to amortize mapper start-up costs; (b) Adaptive Combiners improve local aggregation by maintaining a cache of partial aggregates for the frequent keys; (c) Adaptive Sampling and Partitioning sample the mapper outputs and use the obtained statistics to produce balanced partitions for the reducers. Our experimental evaluation shows that adaptive techniques provide up to 3× performance improvement, in some cases, and dramatically improve performance stability across the board.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

## General Terms

Algorithms, Performance

## 1. INTRODUCTION

The MapReduce parallel data-processing framework, pioneered by Google, is quickly gaining popularity in industry [1, 2, 3, 4, 5] as well as in academia [6, 7, 8, 9, 10]. Hadoop is the dominant open-source MapReduce implementation backed by Yahoo!, Facebook, and others.

In order to provide a simple programming environment for the users, MapReduce offers a limited choice of execution strategies, which can adversely affect performance. For example, mappers checkpoint after every split, all map outputs are sorted and written to file, and reducers read statically determined partitions. To gain more flexibility, new MapReduce-inspired massive data processing platforms have emerged: Dryad [11], Hyracks [12], Spark [13], Nephele [14], Ciel [15] - all include elements of MapReduce, but have more choices in runtime query execution. In contrast to these projects, we chose to enhance MapReduce, to leverage existing investment in the Hadoop framework and in the query processing systems built on top of it, such as Jaql [16], Pig [4], and Hive [5].

While adding new runtime options to Hadoop, we made them adaptive to the runtime environment, to avoid making performance tuning any harder than it already is [17]. Adaptive algorithms demonstrate superior performance stability, as they are robust to tuning errors and changing runtime conditions, such as other jobs running on the same cluster. Furthermore, adaptive techniques do not rely on cardinality and cost estimation, which in turn requires accurate analytical modeling of job execution. Such modeling is extremely difficult for large scale MapReduce environments, mainly for two reasons. First, the scale and interference from other software running in the same cluster. Parallel databases, for instance, rarely scale to thousands of nodes and always assume full control of the cluster. Second, MapReduce is a programming environment where much of the processing is done by black-box user code. Even in higher-level query processing systems like Jaql, Pig, or Hive, complex queries typically rely on user-defined functions written in Java. That is why all these systems offer various query "hint" mechanisms instead of traditional cost-based optimizers. In this environment, the use of adaptive run-time algorithms is a logical choice.

We show that it is possible to make MapReduce more flexible and adaptive by breaking a key assumption of the programming model that mappers are completely independent. We introduce an asynchronous communication channel between mappers, by using a transactional, distributed meta-data store (DMDS). This enables the mappers to post some metadata about their state and see state of all other mappers. Such "situation-aware mappers" (SAMs) can get an aggregate view of the job state and make globally coordinated optimization decisions. In particular, our SAM tasks are able to alter their execution, at runtime, depending on the global state. To the best of our knowledge, no prior

*Work done at IBM Almaden Research Center.

*EDBT 2012*, March 26–30, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-0790-1/12/03 ...$10.00

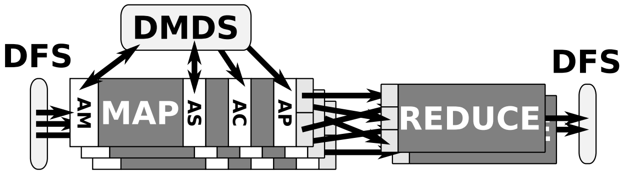


**Figure 1: Adaptive techniques in SAMs and their communication with DMDS.**

system supports such general *intra-task* adaptivity.

The independent mappers assumption allows MapReduce to flexibly partition the inputs, arbitrarily order their processing, and safely reprocess them in case of failures. While implementing SAMs we had to be careful to satisfy key MapReduce assumptions about scalability and fault tolerance, and not introduce noticeable performance overhead. For instance, SAMs can be executed in any order and reexecuted at any time. We also avoid synchronization barriers and pay special attention to recovery from task failures in the critical path.

We utilize SAMs in a number of adaptive techniques: Adaptive Mappers dynamically control the checkpoint interval, Adaptive Combiners use best-effort hash-based aggregation of map outputs, Adaptive Sampling uses some early map outputs to produce a global sample of their keys, and Adaptive Partitioning dynamically partitions map outputs based on the sample. Figure 1 shows the adaptive techniques in the SAMs and their communication with DMDS.

**Adaptive Mappers (AMs)** dynamically change the granularity of checkpoints to trade off system performance, load balancing, and fault tolerance. MapReduce (and its Hadoop implementation) support intra-job fault tolerance by running a separate map task and checkpointing the output for every input data partition, called a *split*. This makes split size an important tuning parameter because having too few splits results in poor load balancing and decreased performance under faults, while with too many splits the overhead of starting and checkpointing the tasks may dominate the job's running time. In contrast, AMs make a decision after every split to either checkpoint or take another split and "stitch" it to already processed one(s). As a result we get the best of both worlds: minimum task startup overhead and dynamic load balancing.

**Adaptive Combiners (ACs)** perform local aggregation in a fixed-size hash table kept in a mapper, as does Hive and as was suggested in [18]. However, unlike these systems, once the hash table is filled up, it is not flushed. An AC keeps the hash table and starts using it as a cache. Before outputting a result, the mapper probes the table and calls the local aggregation function (i.e. *combiner*) in case of a cache hit. The behavior in case of a cache miss is determined by a plugable replacement policy. We study two replacement policies: No Replacement (NR) and Least Recently Used (LRU). Our experimental evaluation shows that NR is very efficient - its performance overhead is barely noticeable even when map outputs are nearly unique. LRU overhead is substantially higher, however it can outperform NR, if map output keys are very skewed or clustered, due to a better cache hit ratio.

While ACs use an adaptive algorithm, their decisions are local and thus do not utilize SAMs. However, we employ SAM-based Adaptive Sampling technique, described next, to predict if AC will benefit query execution and decide which replacement policy and cache size to use. Also, AMs further improve performance benefit of ACs as they increase the amount of data that gets combined in the same hash table.

**Adaptive Sampling (AS)** collects a sample of map output keys and aggregates them into a global histogram. During its initial "sampling" phase, every AM writes a subset of the output keys to a separate sample file, and continuously updates the DMDS with whatever information is needed to determine if a sufficient sample has been accumulated. For example, if the AS stopping condition is to generate $k$ samples, every mapper will increment a sample size counter in the DMDS every time it appends its sample file with $k/100$ output keys. The first mapper that detects that the stopping condition has been satisfied, becomes the leader, collects all the sample files, and aggregates them into one histogram. AS utilizes AMs to take the input splits in random order, thus the histogram is equivalent to what a coarse block-level sampling would produce.

AS has two main advantages over static, pre-determined, sampling runs that have been proposed in the parallel database literature [19, 20] and used in Pig [4]. First, AS is more efficient since the map outputs used in the sample do not need to be produced again by the main query run. However, the main advantage is the ability of AS to determine when to stop sampling at runtime, based on a global condition, e.g. the total number of keys output by *all* mappers. For complex queries, e.g. with black-box predicates, this avoids sample runs that are either too big or too small, often by a very large margin.

The histogram produced by AS has many uses. One that we already mentioned is tuning parameters of AC. Another important one is **Adaptive Partitioning (AP)**, which allows us to change the partitioning of map outputs among the reducers based on the histogram produced by AS. In particular, AP can produce equal-sized range partitions to support parallel sort of map outputs, or reduce skew in joins. In the future we plan to use AS outputs in other adaptive optimization decisions, both within a Hadoop job (e.g. join methods) and between the jobs (e.g. join order).

Hadoop's flexible programming environment allowed us to implement SAMs and use them in adaptive techniques without any changes to Hadoop itself. Instead, the adaptive techniques are packaged as a library that can be used by Hadoop programmers through a simple API. Notice that the original programming API of MapReduce remains completely unchanged. In order to make the adaptive techniques completely transparent to the user, we also implemented them inside the Jaql [16] query processor. Our distributed meta-data store utilizes Apache ZooKeeper [21, 22], a transactional, distributed coordination service.

Our adaptive techniques are implemented in the context of Hadoop and Jaql, and the rest of the paper describes them in this context. However, the general ideas of SAMs and adaptive techniques are applicable more broadly. It should be relatively easy to adopt these idea to other systems that include elements of MapReduce, such as Dryad [11], Hyracks [12], Spark [13], Nephele [14], not to mention other Hadoop-based query processing systems like Hive and Pig.

The main contributions of this paper are:

- We propose SAMs that use a transactional meta-data store to exchange information about their state and collaboratively make optimization decisions.
- We employ SAMs to build a number of adaptive opti-

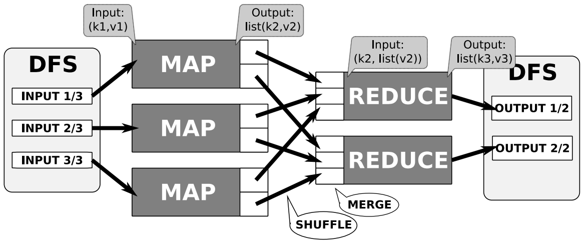


**Figure 2: Data flow in a MapReduce computation.**

mization techniques that preserve fault tolerance, scalability, and programmability of MapReduce.

- An experimental evaluation demonstrates up to 3× performance improvements over the existing state-of-the-art techniques, as well as superior performance stability.

The rest of the paper is organized as follows. We start with some background information on MapReduce and the distributed meta-data store we use, in Section 2. We describe SAMs and the adaptive techniques in Section 3. Section 4 contains experimental evaluation of all the techniques. We discuss related work in Section 5 and conclude in Section 6.

# 2. PRELIMINARIES

## 2.1 MapReduce and Hadoop

MapReduce [3] is a popular paradigm for data-intensive parallel computation in shared-nothing clusters. Example applications for the MapReduce paradigm include processing crawled documents, Web request logs, etc. In the open-source community, Hadoop [1] is a poplar implementation of this paradigm. In MapReduce, data is initially partitioned across the nodes of a cluster and stored in a distributed file system (DFS). Data is represented as `(key, value)` pairs. The computation is expressed using two functions:

```
map    (k1,v1)       → list(k2,v2);
reduce (k2,list(v2)) → list(k3,v3).
```

Figure 2 shows the data flow in a MapReduce computation. The computation starts with a map phase in which the `map` functions are applied in parallel on different partitions of the input data, called splits. A map task, or mapper, is started for every split, and it iterates over all the input `(key, value)` pairs applying the map function. The `(key, value)` pairs output by each mapper are hash-partitioned on the `key`. The pairs are sorted in a fixed-size memory buffer. Once the buffer is filled up, the sorted run, called a *spill* is written to the local disk. At the end of the mapper execution all the spills are merged into a single sorted file. At each receiving node, a reduce task, or reducer, fetches all of its sorted partitions during the shuffle phase, and merges them into a single sorted stream. All the pair values that share a certain key are passed to a single reduce call. The output of each `reduce` function is written to a distributed file in the DFS.

Besides the `map` and `reduce` functions, the framework also allows the user to provide a `combine` function that is executed on the same nodes as mappers right after the `map` functions have finished. This function acts as a local reducer, operating on the local `(key, value)` pairs. This function allows the user to decrease the amount of data sent through the network. The signature of the `combine` function is:

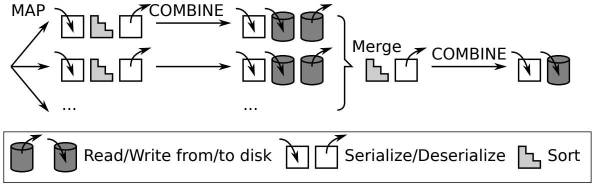


**Figure 3: Serialize, deserialize, disk read, disk write and sort operations performed to apply the `combine` function.**

```
combine (k2,list(v2)) → list(k2,v2).
```

In Hadoop, the `combine` function is applied once the outputs have been sorted in the memory, just before they are spilled to disk. At the end of the map execution, when all the spills are merged into a single output file, the combiner function is applied again on the merged results. Figure 3 shows the sequence of operations necessary for applying combiners.

Each node in the cluster has a fixed number of “slots” for executing map and reduce tasks. A node will never have more map or reduce tasks running concurrently than the corresponding number of slots. If the number of mappers of a job exceeds the number of available map slots, the job runs with multiple “waves” of mappers. Similarly, the reducers could also run in multiple waves.

## 2.2 Distributed Meta-data Store

One of the main components of our SAM-based techniques is a distributed meta-data store (DMDS). The store has to perform efficient distributed read and writes of small amounts of data in a transactional manner. We use Apache ZooKeeper [21, 22], an open-source distributed coordination service. The service is highly available, if configured with three or more servers, and fault tolerant. Data is organized in a hierarchical structure similar to a file system, except that each node can contain both data and sub-nodes. A node’s content is a sequence of bytes and has a version number attached to it. A ZooKeeper server keeps the entire structure and the associated data cached in memory. Reads are extremely fast, but writes are slightly slower because the data needs to be serialized to disk and agreed upon by the majority of the servers. Transactions are supported by versioning the data. The service provides a basic set of primitives, like `create`, `delete`, `exists`, `get` and `set`, which can be easily used to build more complex services such as synchronization and leader election. Clients can connect to any of the servers and, in case the server fails, they can reconnect to any other server while sequential consistency is preserved. Moreover, clients can set watches on certain ZooKeeper nodes and they get a notification if there are any changes to those nodes. Please note that the techniques described in this paper are not limited to ZooKeeper.

For the rest of the paper we refer to DMDS as ZooKeeper. However, Other transactional stores, like Memcached, or even an RDBMS could also be used in place of ZooKeeper, though probably not with the same levels of performance.

# 3. ADAPTIVE MAPREDUCE

In this section we describe a set of techniques for making the MapReduce framework more adaptive to the input data and runtime conditions. These techniques affect different parts of a MapReduce job, yet all of them are implemented

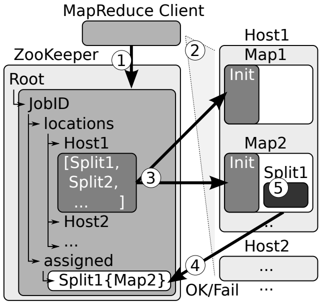


**Figure 4: Local split assignment in AMs.**

inside the map tasks and they all rely on DMDS for global communication, i.e., they utilize SAMs.

The advantages of our SAM-based techniques are two-fold. First, we dynamically alter job execution based on the map input, output, and the environment. This allows us to relieve the user, or the high-level-language compiler, of making the right choices before each job starts. Second, we decentralize the decision making for such changes. Note that all the decisions are made by SAMs as DMDS provides only an API for data access and does not execute any user code. This reduces the load on the coordinator, and prevents it from becoming a bottleneck. It also makes the decision process more flexible as decisions that affect only the local scope are made individually by each SAM.

## 3.1 Adaptive Mappers

In MapReduce, there is a one-to-one correspondence of map tasks and partitions of input data, called *splits*, that they process. To balance the workload across the cluster, a job can have many *waves* of mappers, i.e., more map tasks than map *slots* available to execute them. However, having more mappers increases task scheduling and starting overhead. The startup overhead may include running user code to perform job-specific setup tasks, such as loading reference data, which can become a significant portion of the running time. At the same time, having smaller splits tends to reduce the benefit from applying a combiner, since MapReduce checkpoints results of every mapper.

An AM dynamically "stitches" a set of splits into a single *virtual* split assigned to a mapper, thus changing the checkpoint interval as the mapper is running. AMs decouple the number of splits from the number of mappers and obtain the best of both worlds. That is, load balancing, reduced scheduling and starting overhead, and combiner benefit.

The decoupling is achieved as follows. The split location information is stored in DMDS. A fixed number of mappers are started and they compete for splits. Every time an AM finishes processing a split, it makes a decision to stop or to take a new split from DMDS and concatenate it to the existing one, transparently to the map function. The split assignment conflicts are resolved using the transaction capabilities of DMDS.

Figure 4 shows the main flow of execution and the data structures of our ZooKeeper-based AM implementation.In the first step (marked with "1" in the figure), the MapReduce client creates the AM data structure in ZooKeeper. For each job, we create a `locations` and an `assigned` node. The `locations` node contains metadata for all input splits, organized by hosts where these splits are available. On multi-rack clusters, hosts are further organized by rack and datacenter. The `assigned` node will contain information about which split got processed by which map.

In step 2, the client uses *virtual* splits to start mappers. When the virtual splits are initialized, they connect to ZooKeeper and retrieve a list of the real splits, which are local to the current host (step 3). In step 4, AM picks a random split from the list and tries to lock it. The randomness helps us minimize the number of locking collisions between AMs working on the same host. For example, `Map2` tries to lock `Split1` for processing, by creating a node for `Split1` under the `assigned` node. The created node contains the ID of the map. If the node creation succeeded the map has locked the split and can process it (step 5). After all the data from the chosen split is processed, the map tries to pick a new split. If it succeeds, mapper processing continues unaffected. The split switch is transparent to the `map` function unless it explicitly asks for the location of the split. When mappers finish processing local splits, they start processing any unprocessed remote splits. They do this by selecting a random unfinished host and subtracting the list of `assigned` splits from the list of available splits at that host. If no more splits are available in ZooKeeper, mappers end their execution.

As mappers process splits and accumulate output data, we have to keep in mind that the output data has to be sorted and shuffled, and in case of mapper failure, the data has to be reprocessed. After each split they process, AMs decide if they should pick another split or stop execution, based on how long they've been running, how many splits they processed, and how much output they produced. For instance, an AM stops execution if the size of its output exceeds a pre-determined threshold. The intuition is that if map output size is significant, it will be advantageous to start the shuffle early and overlap it with the mappers. Once an AM stops, the reducers can start copying its output.

Additionally, a user may choose to specify a time limit $t$ to mitigate performance impact of failures. An AM will not take any more splits if it had been running longer than $t$. Given a mean time between failures (MTBF), it is possible to automatically optimize $t$ by doing a simple cost-benefit analysis. The cost of taking an additional split is paid for during failures when more work must be redone, but the benefit eliminates per-split start-up time during normal processing. Every AM observes how long it's been running, and remembers how long the map start-up took.

If mappers in the earlier waves decide to stop, mappers in the later stages get to process the rest of the data. The AMs of the last wave do not stop until all splits are processed.

### 3.1.1 Hadoop Implementation Details

We implemented adaptive mappers in Hadoop by creating a new input format and a new record reader. The new input format and record reader wrap the job's original input format and record reader. When Hadoop asks the input format for the splits, the real splits are stored in ZooKeeper and a number of virtual splits are created and returned. When Hadoop asks the record reader to read a record from a virtual split, a real split is fetched and the record is read from it instead.

To facilitate multiple waves of mappers, we start, by default, four times more AMs than slots in the cluster. From our experience during experimental evaluation, four waves of mappers provided most of the benefit from overlapping the map and shuffle phases, without introducing too much overhead. If advanced users specify a AM time limit, they should also manually adjust the number of waves based on their expectation of the map running time. A better alternative to starting a fixed number of waves would be to modify Hadoop framework to allow new virtual splits to be created during job execution, as needed. However, that is outside the scope of this work. The wave notion is logical, but Hadoop runtime allocates sequential IDs to the map tasks, starting from 0. Given $n$ total map tasks and $k$ map slots, we consider map tasks with IDs $n - k - 1$ to $n - 1$ as the *last wave*. In general, is this Hadoop feature was not available we could utilize DMDS to designate the last $k$ AMs to start as last wave.

We currently do not support speculative execution of AMs. Though we could implement it, we have not observed any need for it. Speculative execution is only beneficial for jobs with long-running, unbalanced tasks. AMs avoid this problem by using, by default, a relatively small real split size, e.g. a fraction of a DFS data block, since the split switch overhead is minimal.

### *3.1.2 Fault-tolerance*

An important aspect of the MapReduce framework is its fault-tolerance. Because AMs manage the splits themselves, special care needs to be taken to handle task failures. Essentially, as a mapper gets splits, those splits are not processed by other mappers. If a mapper fails we need to make sure that the splits that it tried to process are eventually processed (possibly by other mappers).

AM failure resolution relies on the fact that MapReduce automatically restarts failed mappers. A restarted AM scans the `assigned` node and remove all the entries assigned to the virtual split of this AM, which were locked by the previous execution attempt of the same task. Thus, the splits assigned to the previous attempt become available for reassignment. In order for the other mappers to learn about the newly available splits, they read the `assigned` node when they run out of splits. In this way other mappers can help balance the workload that needs to be redone.

### *3.1.3 Scheduling Support*

AMs take special care to cooperate with the MapReduce scheduling algorithms. The default Hadoop scheduler is based on a FIFO queue and it always assigns all the map tasks of a job to the available slots before taking any map tasks of the next job in the queue. Thus, the FIFO scheduler operates the same way with regular mappers and with AMs.

In contrast, the FAIR scheduler [23], that has gained popularity in large shared clusters, divides slots between multiple jobs and schedules tasks of all the jobs at the same time. FAIR avoids starvation of a smaller job that arrives in the system after a large job by reducing the number of slots allocated to the large job. Now, as some map task of a large job finishes, its slot can be used to run the tasks of a smaller job. Thus, the large job gets throttled down to let the small job finish first. FAIR policy typically results in much better average response time for batches of jobs than FIFO. Notice, that FAIR relies on large jobs having many waves of mappers to throttle them down. This is usually the case for normal mappers. However, AMs may finish the entire job in one wave, in which case FAIR performance regresses to that of FIFO.

To support FAIR, AMs include a mechanism to respect slot allocations and shut down some mappers if the allocation shrinks. To achieve this, we store the number of slots allocated to every job in ZooKeeper. We introduced a small modification to FAIR scheduler code, so that every time this number gets changed by the scheduler it is also updated in ZooKeeper. We also maintain in ZooKeeper a number of currently running AM tasks for each job. Every time a non-last-wave AM tries to take a new split, it reads these two counters for its job, and if the number of running AMs exceeds the current allocation, the AM terminates. Last wave AMs do not do this check to guarantee job completion. Note that if FAIR increases slot allocation for an adaptive job, new AMs will be started by Hadoop.

For more advanced schedulers that need to understand performance characteristics of a job, other changes may be needed to support AMs. For example the FLEX scheduler [24] would have to be updated to read from ZooKeeper in order to understand how much progress a job has made (as measured by real splits).

## 3.2 Adaptive Combiners

MapReduce supports local aggregation of map outputs using *combiner* functions to reduce the amount of data that needs to be shuffled and merged in the reducers. Hadoop combiners require all map outputs to be serialized, sorted, and possibly written to disk (see Figure 3). However, it is well know from the database literature that hash-based aggregation often performs better than sort-based aggregation.

In this section we describe how we leverage hash-based aggregation for combining map outputs with frequent keys, while keeping the sort-based aggregation as a fallback alternative for non-frequent ones.

The ACs preserve the benefit of shuffling and merging less data in the reducers, while eliminating some of the overhead required to apply combiners. We replace sort with hashing for the frequently occurring map output keys, by maintaining a fixed size cache of partial aggregates (implemented as a hash-map). For each map output, $R$, we probe the cache with $R$'s key. On a cache hit, we apply the combine function for the output value and the cached value and store the result back into the cache. On a cache miss, if the cache is not full we create a new entry for the output pair. If the cache reached its size limit a pair has to be output. Depending on the cache replacement policy, we either directly output the current pair (*No-Replacement* policy), or insert the current pair into the cache and remove and output the least-recently-used (LRU) pair from the cache (*LRU* policy). The NR policy assumes that frequent keys will be inserted into the cache before it gets full. If the key distribution is uniform or normal and in no particular order, than, on average, the first set of keys that could fit into the cache are as good as any set of keys. Moreover, this cache policy has very small overhead as no deletions are performed. In LRU we insert the current pair in the cache and remove and output the pair with the least-recently-used key from the cache. The main idea of this policy is to keep in the cache the current popular keys and maximize the aggregation opportunities. For instance, LRU is an optimal policy if data is sorted on the output key, whereas NR may perform very badly in this case, depending on the order. Other policies can also be implemented. Finally, when there is no more input for the map, we scan the cache and write all the pairs to the output.

Notice that ACs are best-effort aggregators. They might not perform all the possible aggregations, but they will never spill to disk. In fact, the regular combiners are still enabled and they will be able to perform the aggregations missed by the cache. Also worth noting is that the ACs operate on deserialized records, as they reduce the amount of data that regular combiners have to serialize and sort. Moreover, ACs benefit increases when they are used with AMs since multiple splits can be processed by an AC without draining and rebuilding the cache.

**Multi-core Optimizations:** To make use of the multi-core machines available to a Hadoop cluster, usually, multiple mappers are scheduled to run in parallel in different processes, on a single node. Using ACs, each mapper will have its own cache. One such cache can only use a fraction of the memory and do a fraction of the possible aggregations. One way to overcome this is to have a single map process with multiple mapper threads and a single shared cache. ACs could be adapted to work in this multi-threaded setup, using ideas that have been explored for hardware caching of hash tables on multi-core machines in [25].

## 3.3 Adaptive Sampling and Partitioning

The partitioning function of a MapReduce job decides what data goes to which reducer. In Hadoop, by default, the partitioning is done by hashing, though a custom partitioning function may be used (e.g., range partitioning for global sorting, or for performance reasons.) Custom or not, the partitioning function is statically decided before the job starts. In cases when good partitioning depends on the input data, a separate sampling job is often used. The sampling could be expensive as it is not clear how much input data the mappers need to process to produce sufficient output between all of them. Also, the sampling effort is wasted when all the data are reprocessed by the main job.

In contrast, our AS technique piggybacks on the main job, and dynamically decides when to stop sampling, based on a global sampling condition. Thus, AS eliminates the need for a sampling stage in which work is wasted, balances the sampling across cluster, and avoids sampling too much or too little.

AS produces a global histogram of map output keys early in the map stage. This histogram has many applications, for example, setting AC parameters, but a particularly important one is AP, which dynamically decides the partitioning function while the job is running.

### 3.3.1 Adaptive Sampling

The goal of this technique is to obtain a good sample of the mapper output and balance the sampling workload across the cluster. The main idea is to have mappers independently sample their output data while coordinating to meet a global sampling requirement. After the sampling requirements are met, a leader mapper is elected to aggregate the samples. The coordination between mappers is achieved using ZooKeeper.

AS depends on AMs for two reasons. First, AMs randomly chose input splits to ensure a random block-sampling. Second, AMs guarantee that enough map outputs are generated in the first wave, by not stopping execution until the histogram is produced.

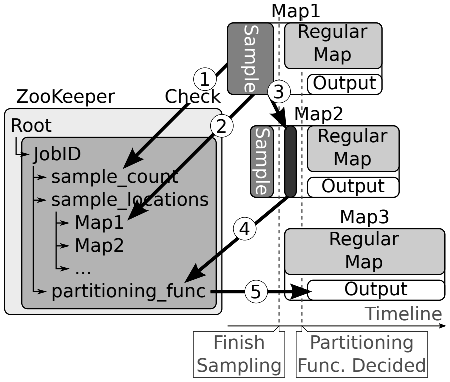


**Figure 5: Communication between mappers and ZooKeeper in AS and AP for performing a global sort using a fixed number of samples (`samples_count`).**

AS has two phases, a Sample-Collection phase and a Sample-Aggregation phase. In Sample-Collection, the status of the global sampling requirements is stored in a predetermined place in ZooKeeper. An example of a global sampling requirement is a fixed number of samples and, in this case, ZooKeeper stores the current sample count. Figure 5, shows the communication between mappers and ZooKeeper during AS (steps 1 to 3). When mappers start they check ZooKeeper to see if the global sampling requirement is met (step 1 in the figure). If the requirement is not met, the mappers start sampling their outputs. Once a mapper accumulates a small fraction of the required samples (e.g., in the case the fixed number of samples, a good fraction is 1%), it updates the sample count in ZooKeeper, writes the sample to the local disk and publishes the location in ZooKeeper (step 2 in the figure). Once a mapper observes that the global requirement is met, it stops sampling and "applies" for leader election, using ZooKeeper. Once a lead mapper is elected, the leader queries ZooKeeper for the sample locations and retrieves and aggregates all the samples (step 3 in the figure). The overhead introduced by the leader election and the sample aggregation is negligible in practice (1-2 seconds). The non-leader mappers continue their regular map processing.

Depending on why sampling is necessary, the mappers could directly output the processed data or they might have to buffer it until the sampling process has finished. If the sampled data is necessary for partitioning, the mappers cannot output any data until the leader aggregates the samples and the partitioning map is computed (see AP description bellow). In this case, mappers allocate a buffer for storing processed data until it can be output. In case the output buffer becomes full the mappers have the option of discarding and reprocessing the data after the sampling is completed, writing the processed data to disk, or stalling.

The sampling process is balanced across the cluster due to the global coordination between the mappers. Mappers might end up sampling different amounts of data depending on when they are scheduled. Early mappers sample more, while later mappers might not sample at all.

**Implementation Details:** The status of the global sampling requirements are stored as the data of a predetermined node in ZooKeeper. After computing a sample fraction, each mapper reads the current state of the global requirements, updates them locally, and writes them back to ZooKeeper. Note that for performance reasons we do not perform locking of the ZooKeeper node and state updates might be overridden and more than the required number of samples might be collected. This is not a problem if the samples are easy to produce. In the case where samples are expensive to produce, the chance of multiple mappers updating the ZooKeeper node in the same time decreases, and so the chance of producing more samples than required decreases. For implementing the leader election we use ZooKeeper Sequential nodes as suggested in the ZooKeeper documentation [21]. Sampled data is transferred between nodes using the same mechanism as for transferring data between mappers and reducers.

Various global sampling requirements can be easily implemented. Some of them could include obtaining a certain coverage of the space. For example, if certain ranges have high frequency variations, more samples could be requested in that range, if some information about input data partitioning is available.

**Fault Tolerance:** As described above, ZooKeeper contains sufficient information about the state of the Adaptive Samplers so that mappers could recover their state in case of restarts. Still, because of the global coordination requirements, the failure of a mapper could slow down other mappers. In the following, we describe two changes to the coordination algorithm which improves recovery for the most probable types of failures, that is, of a node that samples. If there is a software failure, the samples already computed are still accessible to the leader and they need not be recomputed. If there is a hardware failure, the sampled data is lost and recomputing it might be time consuming. To avoid this inefficiency, we change the sampling process so that mappers keep sampling after the global sampling requirement has been met, until the leader mapper finishes aggregating the samples. In this way the lost set of samples is replaced with a new set computed by other mappers in parallel. A less probable type of failure is leader failure. To deal with this situation, we change the leader election process so that non-leader mappers watch for leader failures using ZooKeeper. If such a failure is detected, one of the non-leader mappers becomes the new leader and the sample aggregation process is restarted.

### *3.3.2 Adaptive Partitioning*

AP determines the partitioning function while the job is evaluated. The main idea is for mappers to start processing data, but not produce any output. In parallel, mappers coordinate and the partitioning function is decided by one of them based on the data seen so far. As soon as the partitioning function is decided, the mappers can start outputting data.

The AP piggybacks on AS, which already aggregated seen map outputs into a single histogram at a leader mapper. Based on this histogram, the same leader computes a partitioning function, and publishes it in ZooKeeper (step 4 in Figure 5). For example, if range partitioning is needed to perform global sort, AP will split the histogram into contiguous key ranges with approximately the same number of total occurrences. As soon as the partitioning function becomes available in ZooKeeper the mappers start outputting data, which triggers the start of their partitioners. Each partitioner, upon start-up, loads the partitioning map from ZooKeeper (step 5) and the job continues normally.

## 3.4 Uses of Adaptive Sampling and Partitioning

In this section, we look into more detail on how Adaptive Sampling and Adaptive Partitioning could be used as primitives to obtain more optimization opportunities for MapReduce jobs.

**Global Sorting:** To perform a global sort Adaptive Sampling is used to sample the data and Adaptive Partitioning is used to decide the range partitioning points to balance the data across the cluster.

**Joins:** Adaptive Sampling and Partitioning is used to perform the following optimization in the case of a redistribution equi-join. In this join algorithm provenance labels are added to the each `(key, value)` pair, for example "`R`" for pairs coming from the first dataset and "`S`" for the pairs coming from the second dataset. In reducers, for each unique key, all the "`R`" pairs are read first and buffered in memory, then the "`S`" pairs are streamed by. Using Adaptive Sampling, we detect which of the two datasets has a smaller set of values for each unique key and assign the first label, "`R`", to it. This optimization reduces the memory utilization in the reducer, and is especially useful for foreign-key joins.

In future we plan to support further join optimization, such as dynamically switch between different join algorithms [26]. Also, in case of skewed joins, we could also use Adaptive Sampling and Partitioning to better balance the workload among reducers, just as in global sorting.

**Number of Reducers:** Besides balancing the data across the cluster, Adaptive Partitioning could be used to intentionally unbalance the data. One example where this could be useful is when the mappers are very selective and, as a result, the amount of data which need to be processed by the reducers is very small. In this case the pre-allocated reducers waste most of the time in startup and shutdown overheads. Instead, we detect such situations in the Adaptive Partitioner and direct all the data to a single reducer. The rest of the pre-allocated reducers will terminate immediately after the mappers end.

**Adaptive Combiner Tuning:** Adaptive Sampling is also used for tuning ACs. More exactly we exploit the global sampling mechanism to chose the right configuration parameter for the AC cache. That is, by looking at the global distribution of the data we decide upon the following: (1) whether to use cache or not and (2) which cache policy to use. We make this decision based on the following heuristic. First, if the number of distinct keys in the sample is over 75% of the sample, we disable AC. If the frequency of the $10^{th}$ most frequent key is over 0.1% of the sample size, or we detect that the keys are ordered, we turn on AC with the LRU replacement policy. Otherwise, AC with NR policy is used. We set cache size based on the average size of the map output record, which we measure in AS. Our experimental evaluation supports this heuristic.

**Monitoring:** While a job is running, AS is used to monitor or debug the progress of the mappers. As sample locations get published in ZooKeeper, the user may chose to inspect the samples though a web UI, to ensure that the job is executing correctly from the very early stages.

## 4. EXPERIMENTS

In this section we describe the performance evaluation of our adaptive techniques. We focused our experiments on the

following tasks:
**(A)** *Set-Similarity Join*: Set-similarity joins pair data items from two collections that are deemed to be similar to each other according to a similarity function. Work done in [27] performed an extensive evaluation of algorithms for computing such joins using the MapReduce framework. The join computation requires three stages of MapReduce jobs, as follows: Token Ordering, RID-Pair Generation, and Record Join. We do not explain the details of the join algorithm here. Instead we refer the reader to the original source of the work. We used the same two datasets as in [27], namely **DBLP**[1] with 1.2M records for a total of 310MB and **CITESEERX**[2] 1.3M records or 1,750MB. For our tests we scaled up the datasets by 10× or 100× as described in the original paper. In summary the two datasets contain publication information, including publication title, authors, date, and journal. The complete details of the datasets can be found in the original paper [27]. For this task we stored the datasets in HDFS using text files. We used the Set-Similarity Join source-code from the original paper[3] unchanged.
**(B)** *GROUP-BY*: The GROUP-BY task operated on a single dataset and grouped records by one, two, or three columns and applied an aggregation function for each group. For this task we used a synthetically generated dataset, called **TWL**, where records contain four integers, corresponding to three dimensions (A1, A2, A3), and one fact. The dataset had 10 billion records with approximately 12 bytes per record (using variable length encoding for integers) and a total size of 120GB. We stored the dataset in HDFS using Hadoop Sequence Files. The distributions for the four fields were as follows: A1 and A2 had a normal distribution 0 and 300, with a mean of 150 and a standard deviation of 37.5, and A3 had a normal distribution between 0 and 100, with a mean of 50 and a standard deviation of 12.5.

We used Jaql as a high-level query language to generate the corresponding MapReduce jobs for this type of queries. Since the records are small, we used the `batch` statement from Jaql to group the input data in 100-record batches, to mitigate per-record overhead of HDFS and Sequence Files.

The schema, data value distributions, and the queries for this task, were based on that of a real customer workload.
**(C)** *JOIN and JOIN-ORDER-BY* The JOIN and JOIN-ORDER-BY queries were performed using a single dataset ("fact" table) and a function which assigned a fan-out coefficient to every record (similar to a "dimension" table). We opted for this approach instead of a regular R-S join in order to better control the fan-out and isolate certain performance artifacts. We used an average join fan-out of 1:30. That is, for each input record, we generated from 0 to 128 output records, with an average of 30 and standard deviation of 44. For these queries we used the Sort Benchmark[4] data generator available as part of the Hadoop [1] code-base. We call this dataset **TERASORT**. Each record is a sequence of 100 bytes, where the first 10 bytes constitute the key. We used the key to determine fan-out factor and the order of the records. We stored the dataset in HDFS using the custom file format provided by Hadoop for this dataset [28].
**Hardware** We ran experiments on a 42-note IBM System x iDataPlex dx340. Each node had two quad-core Intel Xeon E5540 64-bit 2.83GHz processors, 32GB RAM, and four SATA disks. Thus the cluster consisted of 336 cores and 168 disks. The nodes were connected using a 1Gbs Ethernet connection. We used one node for running the master daemons to manage the Hadoop jobs and the Hadoop distributed file system and 40 for running the slave daemons. On each slave node we allocated four map and four reduce slots, so, in total, we had 160 map and 160 reduce slots.
**Software** On each node we installed the Ubuntu Linux operating system with kernel version 2.6.32-24 64-bit server edition, Java version 1.6 64-bit server edition, Hadoop version 0.20.2, and ZooKeeper version 3.3.1. To maximize parallelism and minimize the running time, we made the following changes to the default Hadoop configuration: 512MB sort buffer with 25% allocated for bookkeeping information in the mappers, 200MB merge buffer in the reducers, 300 merge factor in the mappers and the reducers, 128MB DFS block size, 128K I/O buffer size, replication factor of one, disable speculative execution, start reducers when mappers start, reuse JVM, and 4GB JVM heap space. We started three ZooKeeper servers each on a different node. We ran each experiment three times and we report the average running time.

## 4.1 Adaptive Mappers Performance

First, we analyzed the overhead introduced by using AMs with ZooKeeper by running a map-only job that slept 1 second for each input record. In order to exclude HDFS performance overhead, we used minimum-size records (1 byte). Figure 6(a) shows the total running time of the job on 20 slots, using both regular mappers and AMs, while varying the number of splits from 20 to 200 to 2000. The input data had 2000 records; thus, for every slot, the job slept for 100 seconds, on average. In case of regular mappers, the job ran with 1, 10, or 100 waves of mappers, respectively. AMs always ran in one wave.

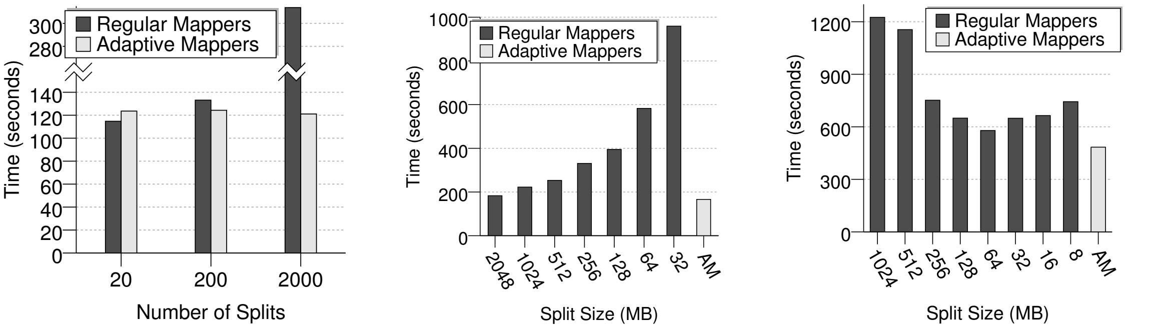
The time difference between the regular mapper tests is due to the overhead incurred by Hadoop for communication, scheduling, and starting map tasks. AMs did not incur this overhead. Notice that JVMs were reused. As we increase the number of splits, the AMs overhead actually decreases because the AMs have more splits to chose from and the probability of a collision when locking a split decreases (see Section 3.1 for details). From the system logs we saw that on average it took around 10ms to lock a split once an AM was initialized.

Next, we evaluated the performance of the AMs on One-Phase Record Join (OPRJ) stage of the set-similarity join [27]. This algorithm uses one MapReduce job, which broadcasts the list of joining RID-pairs to every mapper and streams the original input data, to produce complete join results. We computed the join between the DBLP and CITESEERX datasets scaled up 10×.

Figure 6(b) shows the running time of OPRJ using regular mappers with different split sizes and adaptive mappers. For this experiment we used only 5 cluster nodes (with 40 map slots) to ensure that every node got a non-trivial amount of data. For regular mappers, the 2048MB split size generated a single wave of mappers. At each of the following steps we divide the split size by two and the number of map waves doubles. AMs used a single wave of mappers.[5] The

[1] http://dblp.uni-trier.de/xml/dblp.xml.gz
[2] http://citeseerx.ist.psu.edu/about/metadata
[3] http://asterix.ics.uci.edu/fuzzyjoin/
[4] http://sortbenchmark.org/

[5] We forced the number of waves to be one instead of the



(a) Map-only sleep job.

(b) Third stage of Set-Similarity Join of DBLP 10× and CITESEERX 10×.

(c) JOIN query on 1,000M TERASORT records.

**Figure 6: Comparison between regular mappers and Adaptive Mappers performance.**

poor performance of the regular mappers as the split size decreases is due to two reasons: (1) the time needed to load the list of RID-pairs in memory each time a mapper starts and (2) the Hadoop overhead for scheduling and starting the map tasks. Even when we used a 2048MB split size, regular mappers were still slower than AMs because AMs balance the workload better across the cluster (CITESEERX records are longer than the DBLP records and so they need more processing time). We can see that AMs improve performance 3× compared to the default Hadoop split size (64MB).

Finally, we ran a map-only job that performed a broadcast-join on 1 billion TERASORT records. Figure 6(c) shows the running time of the job using regular mappers with different split sizes and adaptive mappers. For regular mappers we varied the split size between 8MB and 1024MB. The 1024MB splits case used a single wave of mappers. AMs also used a single wave of mappers. We can see that regular mappers perform the worst when we used the largest block size. This is because the input data was sorted by the join key, which was correlated to join fan-out. As a consequence, some input splits had many keys with large fan-out while others had many keys with very small fan-out. This created an imbalance between the mappers when we used large split size. As we decreased the split size, the workload balance improved. AMs are even faster than the fastest regular mappers setting (64MB splits). This is because the regular mappers setting used 20 waves of mappers which introduced significant scheduling and startup overhead.

We conclude that AMs significantly improve the performance and robustness of MapReduce jobs across workloads. They minimize the `map` startup overhead and balance the workload across the cluster. Moreover, as we will see next, AMs improve the performance of MapReduce jobs even further when used in conjunction with ACs.

## 4.2 Adaptive Combiners Performance

We evaluated AC performance on two workloads. First, we focused on the first MapReduce job of the Basic Token Ordering (BTO) algorithm of the Set-Similarity Join [27]. This job tokenizes the join attribute values from each dataset, normalizes the tokens, and computes token occurrence frequencies. Since dataset sizes are typically large, relative to the dictionary size, the use of combiners to compute partial counts for each token provides a very important performance boost.

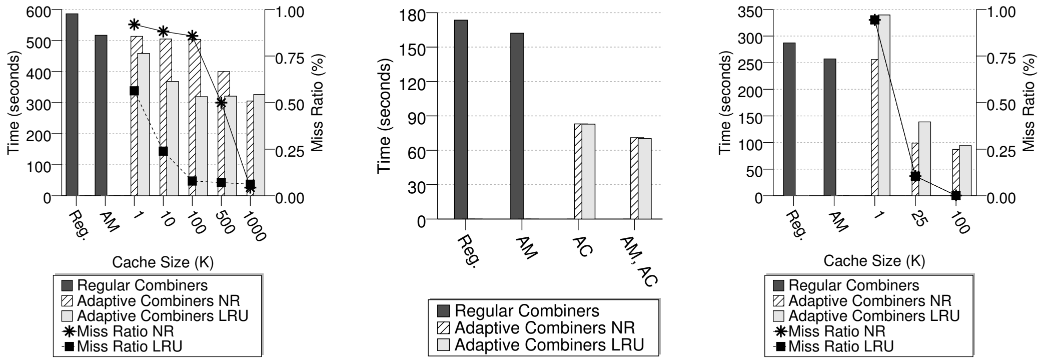
For this experiment, we computed a self-join on the DBLP dataset scaled 100×, to 32GB of text. The number of unique tokens was scaled 10× to around 5 million, while maintaining the original long tail distribution of occurrences.

Figure 7(a) shows the running time for the first MapReduce job of the BTO algorithm for the following cases: (1) regular mappers and combiners ("Reg."), (2) AMs and regular combiners ("AM"), and (3) AMs and ACs with different cache sizes and cache replacement policies, *No-Replacement* (NR) and *Least-Recently-Used* (LRU). We varied the cache size from 1K to 1,000K records, which is the default in our implementation. The bars represent the running time of the job, while the lines represent the cache miss ratio (indicated on the right-hand-side axis). We can see how LRU yields better performance than NR on small cache sizes. This is due to a very skewed distribution of the tokens, as the frequent tokens are better captured by the LRU policy. However, LRU performance did not improve when cache size grew above 100K, as the cache maintenance costs offset the gains from extra cache hits.

Performance overhead of NR was negligible, even when it had a very poor miss ratio due to small cache sizes. Using AMs and ACs with a cache size of 1,000K we obtained about 2× improvement over regular MapReduce job. Note that the input data is processed at a rate of about 6MB/s. This slow processing speed is mainly due to user code in the mapper which has to remove punctuation, normalize case, and tokenize each input record. This leads to the creation of many Java objects and limits the speedup. The speedup would be much larger if the mapper was I/O-bound. Also worth noting is the fact that the largest cache size could only fit one fifth of the unique tokens.

The second set of experiments for evaluating ACs performance was on the GROUP-BY task. The aggregation function was `count`. Figure 7(b) shows the running time of the query for grouping on the A1 dimension for the same four cases as in the previous experiment. For the settings which used ACs we used a cache size of 1K. The A1 dimension had 300 unique grouping keys and so they all fit in the cache. We can see that, with regular mappers, ACs improved performance over regular combiners by a factor greater than 2×. Adding AMs improved the performance even further. Since all the keys fit in the cache, we had 0% miss ratio

default of four in order to clearly isolate the performance characteristics of the AMs.



(a) First stage of Set-Similarity Join, Basic Token Ordering, of DBLP 100× with 10× unique tokens.

(b) GROUP-BY A1 on the TWL dataset.

(c) GROUP-BY A1 and A2 on the TWL dataset.

**Figure 7: Comparison between regular combiners and Adaptive Combiners performance.**

and there was little to no difference between the two cache policies.

Figure 7(c) shows the running time for the same query, but grouping on dimensions A1 and A2. Over the two dimensions we had 90,000 unique grouping keys. The figure also shows the miss ratio for each cache size. We can see that for the largest cache size, which could accommodate all the keys, the ACs bring a 3× performance improvement over regular combiners. For the small cache size case, the LRU policy becomes very expensive. This is due to the more complex data-structures which have to be maintained and to the replacements which take place on every miss. All this cost is paid without improving the miss ratio significantly. Still, the NR cache policy performs significantly better for the small cache sizes and is at most as expensive as the using a regular combiner. Due to lack of space we do not include evaluation of the query grouping on all three dimensions A1, A2, and A3. Its behavior was very similar to that of the previous query, and the default AC (NR policy, 1M records cache) provided around 2× improvement.

Overall, we conclude that group-by jobs can significantly benefit from using ACs, and, when used in conjunction with AMs, the performance improves even further. The main benefit of ACs over regular combiners is that ACs do not have to serialize, sort, and deserialize the data in order to apply the `combine` function. By stitching splits together AMs allow for the cache to be reused. In general the NR cache policy brings significant improvements, but for cases where the cached keys follow a power-law distribution and we can only afford a small cache size the LRU policy performs better than NR.

## 4.3 Performance of Adaptive Sampling and Partitioning

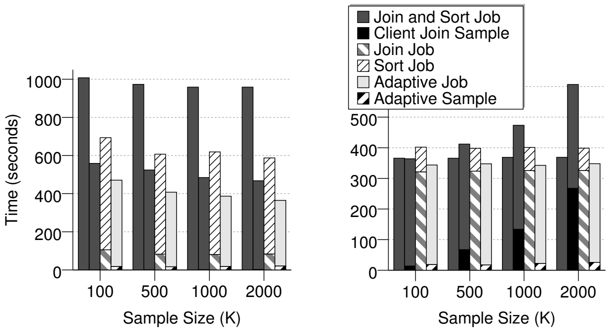
In this section we analyze the performance of the AS and AP using the same broadcast-join query on TERASORT records that we used in Section 4.1, followed by a global sort of the results.

To produce the global order, first, a range-partitioning map has to be computed, based on a sample. We used the following four methods for answering this query. (1) "Client Input Sample" followed by "Join and Sort Job": sequentially sample *input data* in the Hadoop client (before the job is launched, same as the Terasort package in Hadoop) then run a job that does the join and sort operations. (2) "Client Join Sample" followed by "Join and Sort Job": use the `map` function in the client and sequentially sample *join results,* then run the join-and-sort job. (3) "Join Job" followed by "Client Input Sample" and "Sort Job": use two MapReduce jobs where the first one computes the join while the second one sorts the data. We sample the sort input in the client before starting the sort job. (4) "Adaptive Job": using AMs (four waves), AS, and AP. Another alternative would be to use a sampling job followed by a join-and-sort job. This would be similar to the techniques available in Pig [29]. We believe that the join-job-sort-job alternative covers this case as it also includes the overhead of using an additional job, and it would also require extra processing for computing the sample.

Notice, that sampling the input data is not feasible if the sort key comes from the reference data, or is computed from both join inputs. We purposely chose to sort on the fact key, to compare against client-input-sampling jobs. Also, "Client Join Sample" strategy assumes that client machine has enough resources to run the map function. E.g. it has enough memory to load the reference data. In our case, we ran the client on the spare node in the same cluster that Hadoop was using.

Due to the high volume of data being shuffled over the network and the large number of simultaneous readers of map outputs, we noticed that for these experiments some reducers reported network errors, resulting in very unstable performance. To avoid this problem we decreased the number of reduce slots by half.

Figure 8(a) shows the running time of the entire query including the time spent in the client for different methods while varying the amount of collected samples from 100K to 2,000K. We used 100 million TERASORT records as input. Each of the four methods is represented by a vertical bar (broken down by parts, if applicable). The time to sample input data is very small compared to the overall job time and it is not visible in the figure. Notice that for the adaptive method, the processing done while sampling is not lost. We can clearly see the benefit of sampling join results (2) versus sampling input data (1), as the input data do not capture the skew of the join result which leads poor performance due to



(a) 100M records, cheap join predicate

(b) 10M records, expensive join predicate

**Figure 8: Comparison between regular and Adaptive Sampling and Partitioning on a JOIN-ORDER-BY query on the TERASORT dataset.**

imbalance among the reducers. Even if sampling join results in the client does not take a significant amount of time, the Adaptive job (4) achieves better performance than the client-join-sample job (2) due to the use of AMs. Because of the join fan-out, the map output increases 30× and AMs use all four waves of mappers (versus one used by the regular job) to overlap map time with shuffle time. The method using a join job and a sort job (3) does not perform very well because of the time spent on scheduling and starting an additional job. System logs for AS showed that for the 100K samples case, around 60 mappers produced samples (out of 160 running in parallel), and it took around one second to elect the leader, collect and aggregate the samples and propagate the partitioning map. For the 2,000K samples case, the entire process took around three seconds.

Figure 8(b) demonstrates the danger of doing open-ended sampling in the client. In this experiment we used 10 million input records, and to emulate an expensive join we introduce a look-up cost ($1ms$ per input record) and a match cost ($0.1ms$ per output pair). In this case the job sampling client input data (1) was no longer significantly affected by data skew as its running time was dominated by the time spent in the map phase. The time spent in the client for computing join samples (2) increased with the sample size and becomes a significant part of the entire job. Again, the method using two jobs (3) did not perform too well due to the cost of running two MapReduce jobs. The adaptive method was the fastest of the four.

We conclude that none of the client-sample methods, nor the multi-job method are a good overall solution and that choosing the wrong method can be very expensive. On the other hand, using the adaptive techniques is a robust solution that has the best performance. Overall, the adaptive job is always the fastest approach because it does not waste time in the client, it achieves a better balance in the reducers as it samples join results, not input data, and it overlaps map and shuffle time.

## 5. RELATED WORK

The MapReduce paradigm [3] has gained a lot of attention in academia [30, 17, 7, 8, 9, 10, 31] and industry [4, 16, 32, 5]. A comparison of the MapReduce paradigm with parallel DBMS has been done in [31]. A number of higher-level languages have been proposed on top of MapReduce, including Hive [5], Jaql [16], and Pig [4]. These systems include various techniques for dealing with global ordering, map side joins, join reordering, skewed joins, and hash-based partial aggregation [4, 29, 5]. Nevertheless, all these techniques need to be turned on explicitly and configured by the user before the job starts. Moreover some techniques, like sampling and skewed joins, require running an additional MapReduce job. On the other hand, our adaptive techniques are always turned on, tune themselves based on the data seen so far, and do not require additional MapReduce jobs.

In [17], the authors did an extensive evaluation of how different configuration parameters affect job performance in Hadoop. Our techniques try to mitigate these effects and make job configuration less prone to expensive tuning errors. A number of techniques have been proposed to improve the performance of MapReduce jobs [7, 8, 9, 10]. In [9], the authors propose a set of general low-level optimizations which include improving I/O speed for local data, exploiting indexes, using different decoding schemes when deserializing the data, using fingerprinting for faster key comparisons, and block size tuning. The authors of [8] observed that collocating data blocks in the distributed file system and adding an index to each block substantially helps the performance of join algorithms. The study in [10] focused on grouping MapReduce jobs that perform common computations and evaluating each group as a single job. In [7], the authors modified the MapReduce architecture to allow pipelining of the intermediate data between operators. All these four studies are complementary to our study and can be used in our framework to improve the performance even further. More recently, in [33], the authors studied the problem of using MapReduce for one-pass analytics. Similar to our Adaptive Combiner ideas, they propose using hash-based techniques for grouping. They utilize a different replacement strategy, and in their setting, data is serialized before partial aggregates are computed and the partial results are stored in serialized format. As a consequence, they require an extra serialization/deserialization step before the aggregate function can be applied, unless the partial aggregation function can be done on the serialized objects. Nevertheless, our Adaptive Mappers, Adaptive Sampling and Adaptive Partitioning techniques can be employed to boost their performance even further.

MapReduce framework is already dynamic at a task level, i.e. at runtime it decides where and when to execute each task. However, its set of tasks is static. Dryad [11] goes one step further by allowing modification of the dataflow graph once a task is finished. The recently developed Ciel [15] framework adds additional flexibility by allowing user tasks to spawn new tasks, in a controlled way, while maintaining transparent fault tolerance. Still, SAMs are adaptive at a lower granularity, as individual tasks can alter their execution depending on the global state. This allows us to efficiently support techniques such as Adaptive Mappers that are able to minimize the number of task start-ups and balance the workload at the same time. It is not clear how to achieve the same effect using Ciel's task spawning mechanism.

Dryad [11], as well as Hadoop, has considered techniques to direct multiple input partitions to a single task, however, all of these techniques need to be setup statically before the job starts, hence, in the general case, they cannot balance the workload as efficiently as Adaptive Mappers.

Adaptive aggregation algorithms have been studied in parallel shared-nothing architectures [18] as well as in multi-core architectures [25]. In [18], the authors propose a set of parallel aggregation algorithms that dynamically adapt at run-time based on the observed selectivities of the data. From this work we borrow ideas like in-memory hash-tables and local and global decision making, and adapt them to the MapReduce framework, while addressing the problems of synchronization and fault-tolerance. Moreover, we only use the hash-table as a cache and rely on the framework for spilling. A relevant study in [25] focuses on leveraging local and global cache for efficient aggregation processing using multi-cores.

Parallel sort and join algorithms for large datasets have been widely studied since the early 1980's (e.g., [19, 34, 20]). In particular, various techniques have been proposed for parallel sampling [19, 20] and handling data skew in joins in [35, 36]. Though non-adaptive, these techniques are complimentary to our approach. In particular, AP could apply same skew handling techniques if needed.

## 6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a number of adaptive optimization techniques for the MapReduce framework, that dramatically improve its performance and especially performance stability. These adaptive techniques utilize "Situation-Aware Mappers" that are able to cooperatively make global optimization decision. From the experimental evaluation, we observed that the adaptive techniques can bring significant performance improvements. The adaptive techniques never hurt the performance of the MapReduce jobs and configure themselves.

We expect SAMs to become an important extension point for MapReduce framework. Advanced users can implement essentially system-level enhancements using this mechanism, just as we implemented the adaptive techniques described in this paper.

Our future work direction includes adding features to the existing adaptive techniques to allow finer user control. For example, to allow a job to specify a partial order among the splits that AMs will maintain. We are also interested in building more adaptive optimizations for Hadoop and Jaql, especially those utilizing the Adaptive Sampling outputs.

## 7. REFERENCES

[1] Apache Hadoop, http://hadoop.apache.org.

[2] B. F. Cooper et al., "Building a cloud for Yahoo!" *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 36–43, 2009.

[3] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[4] A. Gates et al., "Building a highlevel dataflow system on top of MapReduce: the Pig experience," *PVLDB*, vol. 2, no. 2, pp. 1414–1425, 2009.

[5] A. Thusoo et al., "Hive - a petabyte scale data warehouse using Hadoop," in *ICDE*, 2010, pp. 996–1005.

[6] Y. Bu and et al., "Haloop: Efficient iterative data processing on large clusters," *PVLDB*, vol. 3, no. 1, pp. 285–296, 2010.

[7] T. Condie et al., "MapReduce online," in *NSDI*, 2010, pp. 313–328.

[8] J. Dittrich et al., "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *PVLDB*, vol. 3, no. 1, pp. 518–529, 2010.

[9] D. Jiang et al., "The performance of MapReduce: an in-depth study," *PVLDB*, vol. 3, no. 1, pp. 472–483, 2010.

[10] T. Nykiel et al., "MRShare: sharing across multiple queries in MapReduce," *PVLDB*, vol. 3, no. 1, pp. 494–505, 2010.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007, pp. 59–72.

[12] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *ICDE*, 2011.

[13] M. Zaharia et al., "Spark: cluster computing with working sets," in *HotCloud, USENIX workshop on Hot topics in cloud computing*, 2010.

[14] D. Battré et al., "Nephele/PACTs: a programming model and execution framework for web-scale analytical processing," in *SoCC*, 2010, pp. 119–130.

[15] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: a universal execution engine for distributed data-flow computing," in *NSDI*, 2011.

[16] Jaql, http://www.jaql.org.

[17] S. Babu, "Towards automatic optimization of MapReduce programs," in *SoCC*, 2010, pp. 137–142.

[18] A. Shatdal et al., "Adaptive parallel aggregation algorithms," in *SIGMOD Conf.*, 1995, pp. 104–114.

[19] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "Parallel sorting on a shared-nothing architecture using probabilistic splitting," in *PDIS*, 1991, pp. 280–291.

[20] D. A. Schneider et al., "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *SIGMOD Conf.*, 1989, pp. 110–121.

[21] Apache ZooKeeper http://hadoop.apache.org/zookeeper.

[22] P. Hunt et al., "ZooKeeper: wait-free coordination for internet-scale systems," in *USENIX Conf.*, 2010.

[23] M. Zaharia et al., "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010, pp. 265–278.

[24] J. Wolf et al., "FLEX: a slot allocation scheduling optimizer for MapReduce workloads," in *Middleware, to appear*, 2010.

[25] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *VLDB*, 2007, pp. 339–350.

[26] S. Blanas et al., "A comparison of join algorithms for log processing in MapReduce," in *SIGMOD Conf.*, 2010, pp. 975–986.

[27] R. Vernica et al., "Efficient parallel set-similarity joins using MapReduce," in *SIGMOD Conf.*, 2010.

[28] O. O'Malley and A. C. Murthy, "Winning a 60 second dash with a yellow elephant," Yahoo! Inc., Tech. Rep., 2009.

[29] "Skewed join in Pig," http://wiki.apache.org/pig/PigSkewedJoinSpec.

[30] A. Abouzeid et al., "HadoopDB: an architectural hybrid of MapReduce and dbms technologies for analytical workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.

[31] A. Pavlo et al., "A comparison of approaches to large-scale data analysis," in *SIGMOD Conf.*, 2009, pp. 165–178.

[32] C. Olston et al., "Automatic optimization of parallel dataflow programs," in *USENIX Conf.*, 2008, pp. 267–273.

[33] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using mapreduce," sIGMOD 2011.

[34] M. Kitsuregawa et al., "Application of hash to data base machine and its architecture," *New Generation Comput.*, vol. 1, no. 1, pp. 63–74, 1983.

[35] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *VLDB*, 1992, pp. 27–40.

[36] J. L. Wolf et al., "New algorithms for parallelizing relational database joins in the presence of data skew," *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 6, pp. 990–997, 1994.