Anti-Combining for MapReduce

Alper Okcan
Northeastern University, Boston, USA
okcan@ccs.neu.edu

Mirek Riedewald
Northeastern University, Boston, USA
mirek@ccs.neu.edu

ABSTRACT

We propose Anti-Combining, a novel optimization for MapReduce programs to decrease the amount of data transferred from mappers to reducers. In contrast to Combiners, which decrease data transfer by performing reduce work on the mappers, Anti-Combining shifts mapper work to the reducers. It is also conceptually different from traditional compression techniques. While the latter are applied "outside" the MapReduce framework by compressing map output and then decompressing it before the data is fed into the reducer, Anti-Combining is integrated into mapping and reducing functionality itself. This enables lightweight algorithms and data reduction even for cases where the Map output data shows no redundancy that could be exploited by traditional compression techniques. Anti-Combining can be enabled automatically for any given MapReduce program through purely syntactic transformations. In some cases, in particular for certain non-deterministic Map and Partition functions, only a weaker version can be applied. At runtime the Anti-Combining enabled MapReduce program will dynamically and adaptively decrease data transfer by making fine-grained local decisions. Our experiments show that Anti-Combining can achieve data transfer reduction similar to or better than traditional compression techniques, while also reducing CPU and local I/O cost. It can even be applied in combination with them to greater effect.

Categories and Subject Descriptors

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

Keywords

MapReduce, Anti-Combining, throughput optimization

1. INTRODUCTION

MapReduce, especially its open-source Hadoop implementation, has become one of the leading approaches for parallel Big Data processing. Users can work with "plain" MapReduce, where programs are expressed directly in terms of Map and Reduce functions in a programming language such as Java. Alternatively, they can choose from a variety of high-level languages including PigLatin [20] and HiveQL [23], which come with compilers for translating queries into plain MapReduce code.

Often the shuffle-and-sort phase, when data is transferred from mappers to reducers, represents the bottleneck of a MapReduce job execution. There are several reasons for this. (1) During the shuffle-and-sort phase, large amounts of data are grouped, sorted, and moved across the network [8, 26, 29]. (2) This data transfer is inherent to enable parallel execution. (3) While shared-nothing environments (on which MapReduce tends to be executed) make it easy to increase CPU, memory, and disk resources by adding more machines, this is difficult for the network. Network links and switches are in fact shared resources in the sense that the same link or switch is on the path between many pairs of machines. Hence reducing network load is essential for increasing *throughput* in highly utilized environments.

Network load can be reduced through the use of a *Combiner* as proposed in the original MapReduce paper [6]. A Combiner attempts to decrease mapper-to-reducer data transfer by applying some of the reducers' work on the mappers, replacing individual records with more compact aggregate data. Unfortunately, combining essentially is limited to applications that compute distributive or algebraic [9] aggregates. And even if combining is possible, it will only be effective if many Map output records in the same map task have the same key. The same applies to the in-mapper combining design pattern [16].

Other than a Combiner, the programmer could choose one of the *compression* techniques that usually come with MapReduce implementations. While they differ in computational cost and compression rate, they all follow the same pattern: Mapper output is compressed on the mapper machine. Then the compressed data chunks are sent to the appropriate reducers, where they are decompressed before merging and processing. While conceptually simple, this use of general-purpose compression "outside" the mapping and reducing functionality can add a significant overhead for compression and decompression of large data sets.

Our approach is based on the following observation that trivially holds for any MapReduce program: Consider an input record i. For this input record the Map function might produce zero or more output records $o_1, o_2, \ldots$ Some of these output records will be assigned to the same reduce task,

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610499.

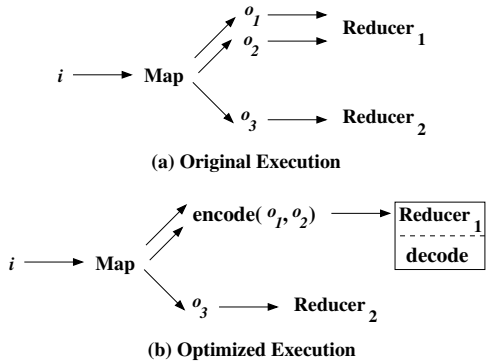


Figure 1: Anti-Combining Intuition

others will end up in different reduce tasks. Figure 1 shows an example. Whenever multiple or large Map output records are assigned to the same task, e.g., o_1 and o_2 to Reducer 1, there is an opportunity for data reduction. Starting with this observation, we chose the following two design goals:

Simple encoding/decoding functions. We want to keep the overhead for data reduction low by only using encode and decode functions (see Figure 1) with low CPU cost. Furthermore, to avoid the need for large buffer space, each encode (and corresponding decode) function call will only be applied to output records of the *same* Map call.

Fine-grained adaptive optimization. To achieve good compression, the choice of encode (and corresponding decode) should be driven by the data. In the example in Figure 1, we might choose a different encoding for $\{o_1, o_2\}$ than for o_3. In particular, for o_3 it might be best to simply leave the record alone and transmit it unaltered. Hence the encoding decision has to be made adaptively at runtime and it might be different for the output of different Map calls.

To meet these design goals, we propose **Anti-Combining**. As will become clear later, the encode/decode functions we propose for Anti-Combining reduce the total data transfer from mappers to reducers by shifting some of the mapper-side processing to the reducers. In that sense it does the opposite of a Combiner, which performs reducer-side work on the mappers.

Anti-Combining can be enabled in any given "plain" MapReduce program through purely *syntactic* program transformations. This makes it possible to enable it *automatically* even in programs written in expressive, usually Turing-complete, languages such as Java or C++. Hence it can be also applied to **compiler-generated MapReduce code** produced by systems such as Pig and Hive, or to other statically optimized MapReduce programs, e.g., those produced by database-style scan-sharing and multi-query optimization [23, 4, 8, 15, 18].

Even though it can be enabled for any MapReduce program, Anti-Combining (like Combiners) will not always result in significant cost savings. Fortunately, in our experience there is a large and diverse spectrum of applications that can significantly benefit from it. **Join processing** for instance, relies on input replication in the map phase in order to compute multi-way joins [3], complex join predicates [19], similarity joins [2, 24], and k-nearest neighbor joins [17, 28]. For many **graph algorithms** including PageRank [21], Hyperlink-Induced Topic Search [14], and social network analysis, the Map function processes a node by emitting output records for each outgoing edge in the node's adjacency list. As graphs tend to be very skewed, Anti-Combining's adaptive approach can significantly reduce cost for nodes with high out-degree, while leaving those with low out-degree alone. Furthermore, all previously proposed **multi-query optimization** techniques such as scan-sharing [4, 8, 18, 25] are a perfect target for Anti-Combining because a single record produced by the shared operator might have to be duplicated many times in order to forward it to the downstream operators of the queries involved.

We make the following main contributions:

1. We identify opportunities for lightweight adaptive runtime optimization of MapReduce programs based on the input-output behavior of the Map function. These opportunities are general in nature and can enable significant reduction of the amount of data transferred between mappers and reducers.

2. We propose Anti-Combining based on encoding and decoding techniques that exploit these sharing opportunities. *EagerSH* is a "safe" optimization for Map output records with the same value component. *LazySH* can be applied even when Map output records have different keys and values, i.e., when traditional compression would not be effective. Both approaches can be used in combination with traditional compression. We also develop a framework that allows the various encodings to co-exist, enabling very fine-grained adaptive optimizations.

3. We propose a syntactic program transformation to enable Anti-Combining. It works for plain MapReduce programs (written in Turing-complete languages such as Java) and does not need to understand program semantics. Furthermore, our approach can be implemented without modifying the MapReduce environment itself.

2. MAPREDUCE OVERVIEW AND QUERY-SUGGESTION EXAMPLE

Consider a typical commercial search engine, which returns the best matching Web pages for a given search query. To aid users in composing a query, most search engines propose possible query completions as the user is typing. For example, after entering "sig" the search engine might suggest "sigmod", "sigmod 2014", and "sigmod acceptance rate". For realtime suggestions, these expansions of a given prefix have to be pre-computed. While the algorithm for selecting suggested expansions is more complex (and usually a trade secret), one of its crucial inputs is the popularity of queries starting with the prefix typed by the user so far. For illustration purposes, we will therefore consider the following version of the Query-Suggestion problem: *We are given a log of search queries. For any string P that occurred as a prefix of some query in the log, pre-compute the five most frequent queries in the log starting with prefix P.*

This is a comparably simple query involving grouping, aggregation, and top-k selection. Hence Query-Suggestion is perfectly suitable for parallel computation using MapReduce. A MapReduce program consists of two major primitives, Map and Reduce. The Map function is executed for each input record, emitting a set of intermediate key/value pairs. The MapReduce environment automatically groups Map output records by key. The Partition function assigns intermediate keys to reduce tasks. When processing a reduce

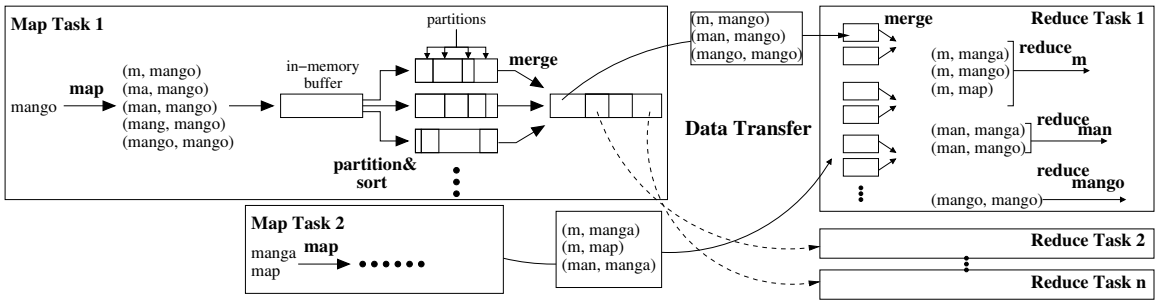


Figure 2: MapReduce Overview with Query Suggestion Example

task, the Reduce function is called for each intermediate key; it works on the list of *all* values with this key.

The natural way of implementing Query-Suggestion in MapReduce is illustrated in Figure 2. For a query Q, Map emits intermediate key/value pairs (P, Q) for each prefix P of Q. By using the prefix as the key, the MapReduce environment guarantees that the Reduce call for prefix P will have all queries with prefix P in its input list. It can then easily determine the most frequent queries for P.

As Figure 2 illustrates, a single query string will result in multiple Map output records, so that this query will be taken into account for each of its prefixes. The Map output is collected in a buffer which is spilled to disk when it fills up. Before writing them to disk, the intermediate key/value pairs are assigned to partitions corresponding to different reduce tasks by the Partition function. Records in each partition are sorted by key. Before the map task is finalized, the spill files are merged on disk, preserving the sort order for each partition. Then each partition is transferred, usually over the network, to the machine responsible for the corresponding reduce task. In the example, keys "m", "man" and "mango" are assigned to reduce task 1. This task processes the records in increasing key order, calling the reduce function once for each key and determining the top-5 queries for it. Notice that in practice each query comes with additional *features*, e.g., on which search result the user clicked. These features can be included in the value component of a record, but are omitted here for simplicity.

For a search query of length n, the Map function will generate n output records. Since each output record contains the query itself, each Map function call's output is *quadratic* in its input size. This results in high cost of the shuffle-and-sort phase. Query-Suggestion's aggregate function admits the use of a Combiner. In particular, the Combiner could replace m occurrences of the same pair (key, value) in the output of a map task by "aggregate" record (key, (value, m)). Unfortunately, our experiments show that the Combiner approach is not very effective for Query-Suggestion due to the large number of distinct query strings in each map task input batch (see Section 7.3). For other problems, a Combiner might not be applicable at all.

We next propose data encoding strategies that exploit data sharing opportunities based on the output produced by a single Map function call.

3. EAGER SHARING STRATEGY

Let (k_1, v') and (k_2, v'), $k_1 \neq k_2$, be two intermediate records emitted by Map for some input record (k, v). As

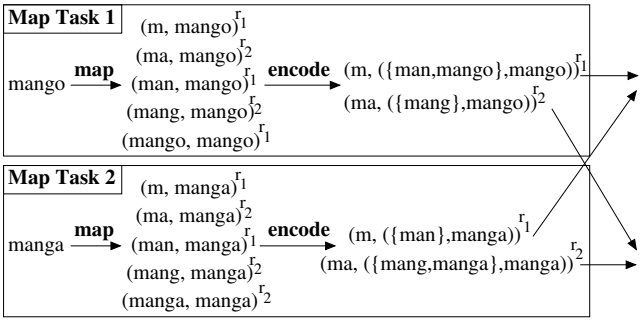


Figure 3: *EagerSH* Map Phase for Query-Suggestion

both output records have the same value field, they present a data reduction opportunity since they can be encoded more compactly as $(k_1, (\{k_2\}, v'))$. Unfortunately, sharing across records with *different* keys is challenging, because intermediate key/value pairs with different keys are processed by different Reduce calls. Hence even if two Map output records (k_1, v') and (k_2, v') share the same value v', the reducer will need these records separately: one for the input list for the Reduce call for k_1 and the other for the input list for the Reduce call for k_2.

We propose *EagerSH*, which enables data reduction based on the shared value component *as long as both keys are assigned to the same reduce task.* (This key-to-task assignment happens in the Partitioner.) Consider the following example:

$$(k_{in},\ v_{in}) \xrightarrow{map} \begin{matrix}(k_1, v_1)^{r_1} \\ (k_2, v_1)^{r_2} \\ (k_3, v_2)^{r_2} \\ (k_4, v_2)^{r_2} \\ (k_5, v_2)^{r_2}\end{matrix} \xrightarrow{encode} \begin{matrix}(k_1, (\{\}, v_1))^{r_1} \\ (k_2, (\{\}, v_1))^{r_2} \\ \\ (k_3, (\{k_4, k_5\}, v_2))^{r_2} \\ \end{matrix}$$

For records (k_1, v_1) and (k_2, v_1), *EagerSH* cannot exploit the common value v_1, because they are sent to different reduce tasks r_1 and r_2, respectively. Since the other three Map output records have the same value v_2 and keys k_3, k_4, and k_5 are assigned to the same reduce task r_2, *EagerSH* would transmit only a single encoded record for value v_2 to reduce task r_2. We then have to ensure that the encoded record is properly decoded in the reduce task so that the Reduce calls for k_4 and k_5 see value v_2.

3.1 EagerSH Map Phase

We present *EagerSH* using the query suggestion example. Recall that for queries "mango" and "manga", the original

Algorithm 1 : *EagerSH*'s Map Function

Input: input tuple I

```
1: MapOutput = O-map(I) /* Original map */
2: result = SELECT MIN(O.key) AS key,
   (setOfOtherKeysInGroup(), O.value) AS value
   FROM MapOutput O
   GROUP BY getPartition(O.key), O.value
3: for all r ∈ result do
4:    Emit(r.key, r.value)
```

Algorithm 2 : *EagerSH*'s Reduce Function

Input: $<$ key$_k$, KVAL = listOf(key set K, value)$>$

```
1: repeat
2:    altKey = Shared.peekMinKey()
3:    if altKey < key_k then
4:       O-reduce(altKey, Shared.popMinKeyValues()) /*
         Original reduce on values with smaller key altKey
         */
5: until altKey ≥ key_k
6: for all (K, value) in KVAL do
7:    for all key in K do
8:       Shared.add(key, value) /* Store for later reduce
         calls */
9: Values = KVAL.getValues()
10: if altKey = key_k then
11:    Values = Values ∪ Shared.popMinKeyValues() /* Ap-
       pend values with same key in Shared */
12: O-reduce(key_k, Values) /* Original reduce */
```

map function generates five key/value pairs per query, each with a different prefix as the key and the same input query as the value. The Partition function assigns keys "m", "man", and "mango" to reduce task 1 and "ma", "mang" and "manga" to reduce task 2.

Figure 3 shows the encoding of the Map function output for two different calls, one for input "mango" in map task 1 and one for "manga" in another map task 2. Consider the Map call for "mango". Instead of sending (m, mango), (man, mango), and (mango, mango) separately to reduce task 1, *EagerSH* sends the more compact (m, ({man,mango},mango)), thus eliminating the value duplication. To generate the encoded record, *EagerSH*'s map function first executes the original Map on the given input record. It then groups the original Map's output by value and partition number (as assigned by the Partitioner). For each group, a single record is emitted. Its key is the *smallest* key in the group; all other keys are added to the value component.

Algorithm 1 shows the pseudo-code. Notice that Map produces key/value pairs, hence MapOutput has a key and a value attribute. The Partition function used by the MapReduce job, getPartition, returns the reduce task to which a key is assigned. And setOfOtherKeysInGroup is a function that returns the set of all keys except for the minimal one in the group. The minimal key is chosen as the "representative" key for the encoded record, because all Reduce calls in a reduce task happen in ascending key order [6]. This way the other keys can be decoded **before** their Reduce calls are executed.

3.2 EagerSH Reduce Phase

The encoded records generated by the mappers have to be decoded on the reducers. In particular, for each of the keys that were transmitted with the value component of an encoded record, the corresponding key/value pair has to be made available before the key's Reduce call. To avoid modifications to the underlying MapReduce system, we rely on a data structure called **Shared**. (It is discussed in more detail in Section 5.) Structure Shared is defined at the level of a reduce *task*, i.e., is visible to all Reduce function calls in the same task. Hadoop provides a method called `setup` for initializing such data structures before the first Reduce call in a task, and a `cleanup` method that is executed after the last Reduce call in that task completed.

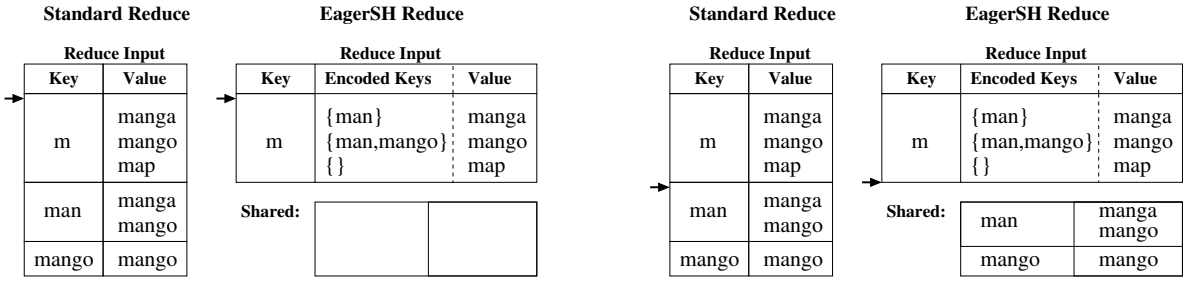
Figure 4 illustrates the difference between the original Reduce and *EagerSH*'s Reduce. In the original MapReduce execution, reduce task 1 receives all the records with the keys assigned to it by the Partitioner, in key order. It then calls Reduce three times, first for key "m", followed by "man", and finally "mango" in the example.

EagerSH's Reduce only receives three encoded records, in this case all those with key "m". Before executing the original Reduce call for key "m", *EagerSH*'s Reduce scans through all records with that key and inserts into Shared the corresponding key/value combinations for all keys encoded in the value component. In the example, records (man, manga), (man, mango), and (mango, mango) are added to Shared. Then the original Reduce is called for key "m". The other keys are processed the same way: first the input list is scanned to decode and insert into Shared, then the original Reduce function is executed. For correctness, *EagerSH*'s Reduce conceptually has to work with the merged list consisting of both the "normal" reduce task input buffer and Shared. We achieve this with a merge-sort style approach that reads from normal input buffer and Shared in lockstep.

Algorithm 2 shows the pseudo-code. KVAL is a list of encoded records, each consisting of a list of keys and the value shared by them. KVAL.getValues returns all values in list KVAL; and Shared.getValues returns all values for a given key. Assume the MapReduce environment just started executing Reduce for key$_k$, i.e., key$_k$ is the smallest key in the reduce task's input buffer for which Reduce has not yet been executed. Since previous Reduce calls might have inserted records with smaller keys into Shared, *EagerSH*'s reduce has to make sure that their Reduce calls are processed first. This is done by the repeat-until loop.

The following for-all nested loop scans the list of encoded records with key key$_k$ in the reduce task input and inserts the decoded key/value pairs into Shared. (Since encoding used the smallest key as the representative, it is guaranteed that all newly inserted records have keys greater than or equal to key$_k$!) Finally the original Reduce function is executed on the union of all records with key key$_k$ from both KVAL (i.e., the "normal" reduce task input) and Shared.

Decoding has to deal with yet another subtle problem. Consider again the example in Figure 4. With *EagerSH*, the input buffer for reduce task 1 only contains records with key "m". Hence the MapReduce system will only call *EagerSH*'s Reduce function for key "m". The other two keys—"man" and "mango"—only appear in Shared. Hence *EagerSH*'s Reduce would not be called for them by the MapReduce system. To make sure the remaining records in Shared are processed after the last "regular" Reduce call completed, the re-



(a) Initial state of reduce tasks.

(b) State after first reduce call execution on values with key m.

Figure 4: Original Reduce vs. *EagerSH*'s Reduce for Query-Suggestion

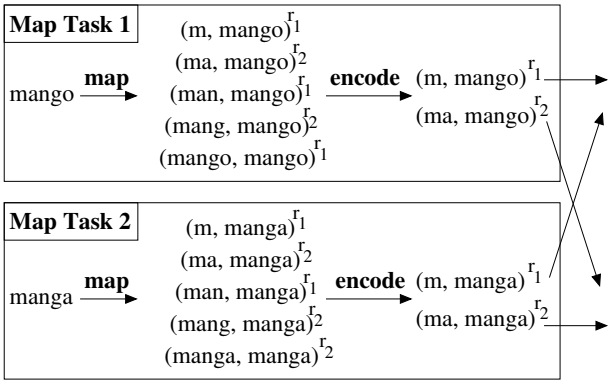


Figure 5: *LazySH* Map Phase for Query-Suggestion

duce task's clean-up function also has to use a similar repeat-until loop to process all remaining records in Shared. (Recall that `cleanup` is called automatically by the MapReduce system after all Reduce calls of the task have completed.)

4. LAZY SHARING STRATEGY

To decrease mapper output size, a Combiner requires records with the same key, *EagerSH* requires records with the same value, and traditional compression techniques require some form of redundancy among the keys and values. In contrast, our second strategy, *LazySH*, might achieve significant data reduction **even if all keys and values in the output of a Map call are unique**. It is able to do this because instead of sending Map *output* from mappers to reducers, *LazySH* simply transfers the Map *input* record to all reduce tasks that would have received some of the Map output for this record, e.g.:

$$(k_{in},\, v_{in}) \xrightarrow{map} \begin{matrix}(k_1, v_1)^{r_1}\\ (k_2, v_2)^{r_1}\\ (k_3, v_3)^{r_2}\\ (k_4, v_4)^{r_2}\\ (k_5, v_5)^{r_2}\end{matrix} \xrightarrow{encode} \begin{matrix}(k_1, (k_{in}, v_{in}))^{r_1}\\ (k_3, (k_{in}, v_{in}))^{r_2}\end{matrix}$$

Since reduce tasks r_1 and r_2 would have received Map output records for input (k_{in}, v_{in}), those and exactly those two reduce tasks will receive (k_{in}, v_{in}). Since Reduce ultimately needs the Map output records, these have to be generated **lazily** on the Reducer by re-executing Map there.

Algorithm 3 : *LazySH*'s Map Function

Input: input tuple I

```
1: MapOutput = O-map(I) /* Original map */
2: result = SELECT MIN(O.key) AS key
     FROM MapOutput O
     GROUP BY getPartition(O.key)
3: for all r ∈ result do
4:     Emit(r.key, I)
```

Depending on the application, *LazySH* could achieve *asymptotic* data reduction. Consider again the Query-Suggestion problem for a query string Q of length n. The original map function would generate a pair (P, Q), for each prefix P of Q. Hence it generates a total of $(1+n)+(2+n)+\cdots+(n+n) = n(n+1)/2+n^2$ data for Q. In the best case for Anti-Combining all prefixes P are assigned to the same reduce task. For this case, *EagerSH* would produce a single output record containing all prefixes P and a single copy of Q for a total data size of $n(n+1)/2+n$, which is still quadratic in n. For that same scenario, *LazySH* would simply send the input record Q of size n.

4.1 LazySH Map Phase

Figure 5 illustrates *LazySH* for our running example. For input query "mango", instead of sending (m, ({man, mango},mango)) to reduce task 1, *LazySH* transfers only (m, mango). Algorithm 3 shows the pseudo-code. It first computes the output of the original Map call for input record I, then finds the minimal key for each reduce task (i.e., partition) that would have received some of that output. Finally record I is emitted for each of these minimal keys.

The SQL statement in Algorithm 3 highlights the difference to *EagerSH*'s Map in Algorithm 1. Since *LazySH*'s Map function groups the original Map output only by partition (and not also by value), there are more data reduction opportunities. And by using Map input I as the value, it does not need to transmit all the other keys. This is crucial for asymptotic (in Map input size) data reduction for the Query-Suggestion problem.

To make the differences between the original MapReduce program, *EagerSH*, and *LazySH* more tangible, consider the following example of the real query "watch how i met your mother online". If the Partitioner assigns all its prefixes to the same reduce task, then *EagerSH* would transmit $34 \cdot 35/2 + 34 = 629$ characters, significantly improving over

Algorithm 4 : *LazySH*'s Reduce Function

Input: < key_k, VAL = listOf(map input I)>

```
1: repeat
2:    altKey = Shared.peekMinKey()
3:    if altKey < key_k then
4:       O-reduce(altKey, Shared.popMinKeyValues()) /*
         Original reduce on values with smaller key altKey
         */
5: until altKey ≥ key_k
6: for all I in VAL do
7:    MapOutput = O-map(I) /* Original map */
8:    for all (key, value) in MapOutput do
9:       if getPartition(key) = this.partitionNumber then
10:         Shared.add(key, value)
11: O-reduce(key_k, Shared.popMinKeyValues()) /* Original reduce */
```

the original program's output of size $34 \cdot 35/2 + 34 \cdot 34 = 1751$. However, *LazySH* would do even better, requiring only $1 + 34 = 35$ characters to be transmitted, using "w" as the key and the complete query as the value component.

4.2 LazySH Reduce Phase

The reduce tasks of *LazySH* receive Map input, not output, therefore decoding in the reducer requires re-execution of the original Map function. Decoded records are stored in a reduce-task level data structure Shared to allow data transfer between individual Reduce calls, as discussed for *EagerSH*'s Reduce. Since not all outputs of a given Map call might be assigned to the current reduce task, the Partition function has to be used to determine those that are. Algorithm 4 shows the pseudo-code for *LazySH*'s Reduce. It essentially is identical to *EagerSH*'s Reduce, except for the decoding process that calls the original Map and getPartition functions.

For simplicity, we illustrate the algorithm with an example in Figure 6. Similar to *EagerSH*'s reduce, all values with minimal key "m" are present in the input buffer of reduce task 1. When the MapReduce system calls *LazySH*'s Reduce for prefix "m", the original Map function is applied to generate all original Map output pairs. For each output record the Partition function is applied to identify those records that belong to reduce task 1. E.g., when input record (m, manga) is processed in this Reduce call, only (m, manga) and (man, manga) are inserted into Shared.

5. THE SHARED DATA STRUCTURE

The Shared data structure used in the reduce phase is designed to efficiently manage decoded key-value pairs and return all records that have the minimal key. It maintains the minimal key using a min-heap, and an in-memory hash-table that maps keys to their corresponding values. As the memory reserved for Shared fills, the data is spilled to local disk in sorted key order. This is done by repeatedly removing the minimal key from the min-hash and then removing the list of values for it from the hash-table, writing it sequentially to disk. This mirrors what happens during the Map phase of the original MapReduce program. For each spill, the minimal key is recorded in Shared. If the number of spill files exceed the merge threshold, they are merged, again mirroring the standard map phase processing [26].

```
// Original Mapper class
Class Mapper {
   map(K, V, context) {...}
   setup(...) {...}
   cleanup(...) {...}
}

// Extended Context class
Class AntiContext {
   mapOutput

   write(K key, V val) {
      mapOutput.insert(key, val)
   }

   getOutput() {
      return mapOutput
   }
}

// Adaptive Mapper for Anti−Combining
Class AntiMapper {
   Mapper o_mapper; AntiContext a_context

   setup(...) { call o_mapper.setup() }
   cleanup(...) { call o_mapper.cleanup() }

   map(K key, V val, context) {
      o_mapper.map(key, val, a_context) //Call original map, measure cost
      mapOutput = a_context.getOutput()
      mapOutput.partition(Partitioner)     //Call Partitioner, measure cost
      if ((cost of map + cost of partition call) * number of partitions > T)
         for all partitions P in mapOutput do context.write(EagerSH−encoded P
      else
         for all partitions P in mapOutput
            if (size of EagerSH−encoded P < size of input record (key, val))
               use EagerSH to encode P
            else
               use LazySH to encode P
            context.write(encoded partition P)
   }
```

Figure 7: Syntactic rewrite of original mapper class to enable Anti-Combining

As shown in Algorithms 2 and 4, Shared provides three functions: peekMinKey(), popMinKeyValues(), and add(key, value). As detailed in Section 3.2, Anti-Combining **only** reads key-value pairs stored in Shared having minimal key. The min-heap is used to find the minimal key at constant cost when peekMinKey() is called. All values associated with the minimal key are retrieved using popMinKeyValues(). If the values are in the in-memory hash-table, they are found through a lookup using the minimal key. If values with the minimal key are spilled to disk, since the spill files are sorted by key, Shared performs a buffered sequential read on the relevant spill files, never needing random accesses. Add(key, value) is performed on the in-memory hash-table in constant amortized running time. Inserting the key into the min-heap requires logarithmic time.

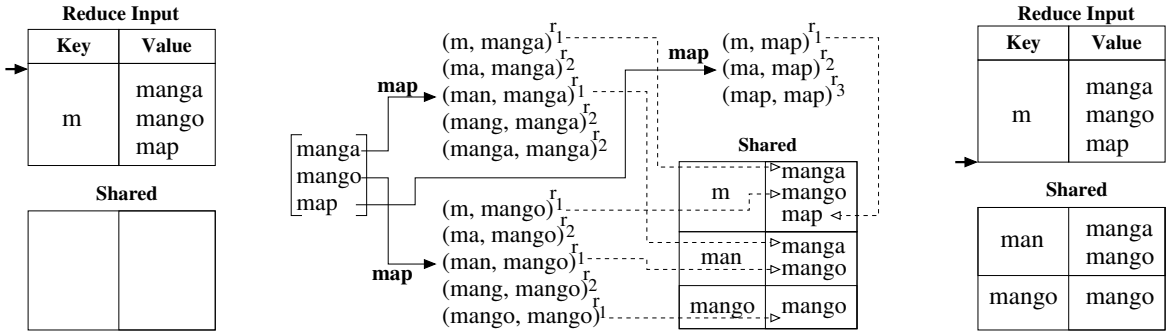
Using Combine in the Reduce Phase. For MapReduce programs that admit the use of a Combiner during the map phase, the provided Combine function can also be leveraged during the reduce phase of *EagerSH* and *LazySH*. The Combine function summarizes a set of Map output records that have the same key. Hence it can be applied to reduce the amount of data managed in Shared. Instead of adding each decoded key-value pair separately, Shared can immediately Combine values and maintain a single record for each unique key. Our experiments show that Combine reduces the size of Shared significantly, sometimes allowing in-memory processing without spilling to disk. Combining during the Reduce phase is highly effective, because records with the same key end up in the same reduce task. Hence even for applications where Combining is not effective in the Map phase, leveraging the Combine function can be highly effective for reducing the size of Shared in the reduce phase.

6. ENABLING ADAPTIVE RUNTIME OPTIMIZATION

We discuss how to enable Anti-Combining through syntactic transformations of a given MapReduce program.

6.1 Program Transformation

The original mapper class is replaced by a new mapper class called AntiMapper as shown in Figure 7. This is done by modifying the class name in the statement that sets the



(a) Initial state of reduce task.

(b) Decoding key/value pairs into Shared during reduce call execution on values with key m.

(c) State after first reduce call.

Figure 6: *LazySH*'s Reduce Phase for Query-Suggestion

```
// Original Reducer class
Class Reducer {
   reduce(K, iter<V>, context) {...}
   setup(...) {...}
   cleanup(...) {...}
}
```

```
Class Shared<K, V> {
   minHeap<K>
   hashMap<K, V>
   keyComparator
   groupingComparator

   add(K, V) {...}
   popMinKeyValues() {...}
}
```

```
// Adaptive Reducer for Anti−Combining
Class AntiReducer {
   Shared
   Reducer o_reducer; Mapper o_mapper

   setup(...) { initialize Shared; call o_reducer.setup() }
   cleanup(...) { cleanup Shared; call o_reducer.cleanup() }

   reduce(K key, iter<adaptiveV> values, context) {
      repeat {
         altKey = Shared.peekMinKey()
         if (altKey < key)
            o_reducer.reduce(altKey, Shared.popMinKeyValues(), context)
      } until altKey >= key
      for all val in values {
         determine encoding used for val
         decode val and insert key−value pairs into Shared
         // Decoding for LazySH calls o_mapper.map and the
         // original Partitioner, which is available through the context
      }
      o_reducer.reduce(key, Shared.popMinKeyValues(), context) }
   }
}
```

Figure 8: Syntactic rewrite of original reducer class to enable Anti-Combining

mapper class for the program. Notice that we do not need to modify the original class. Instead, AntiMapper contains an object of the original mapper class and an extended context object. The former enables the use of the original Map functionality. The latter extends Hadoop's context class and is needed because in Hadoop mappers emit their output to the context object. The extended context intercepts the original Map output and replaces it by the encoded version. Notice that MapReduce implementations other than Hadoop would have to rely on a similar mechanism for collecting Map output, enabling a similar interception approach.

For each individual Map call, the AntiMapper has to adaptively choose between *EagerSH* and *LazySH* to pick the encoding strategy that decreases data transfer the most. In fact, the encoding decision is made independently for each partition (i.e., reduce task) the output records of the Map call are assigned to. There are two reasons for this: First, since different tasks cannot share data, the encoding decision for one partition does not affect the choices for the others. Second, the greater flexibility enables greater data reduction compared to enforcing the same decision for all partitions. Notice that the original program's *unencoded* output is a special case of *EagerSH* when the set of keys included in the value component is empty. For proper decoding in the reducers, a flag is added to the encoded record's value component to indicate which strategy was used.

As discussed for *EagerSH* and *LazySH*, AntiMapper's Map function first executes the original program's Map function on input record (key, val) (through the o_mapper object) and then partitions the output, which was intercepted by the extended context object, using the original program's Partitioner (accessed through the context, as usual). After these steps, the **exact execution cost of the original map and getPartition for this input record (key, val) are known**. (We currently measure cost in terms of CPU time, but one could similarly use measures that include I/O cost.) We also know to how many reduce tasks (key, val)'s output records will be sent. From this we can compute the total cost of re-executing o_mapper.map and getPartition if *LazySH* was used for encoding. To deal with expensive map and getPartition cases, Anti-Combining uses a cost threshold T that disables *LazySH* if total re-execution cost exceeds the threshold (see Figure 7). The "ideal" setting for T depends on user preference. Smaller T limits the overhead from "duplicate" Map and getPartition executions, but limits Anti-Combining's encoding choices. Hence larger T enables greater decrease of network transfer at the cost of higher CPU load. In the extreme, if T is set to ∞, then Anti-Combining chooses freely between *EagerSH* and *LazySH* the one that minimizes data transfer cost. Setting $T = 0$ forces Anti-Combining to only use *EagerSH*, i.e., completely avoid any duplicate Map and getPartition calls.

Like for the mapper class, we also replace the original reducer class with our AntiReducer in the program statement that sets the reducer class. AntiReducer, as shown in Figure 8, can be generated through syntactic rewrites from the given original reducer class. It essentially performs the decoding work as introduced in Alg. 2 and 4. There are some non-trivial technical challenges to enable reading from the union of the regular reducer input and from Shared, which contains decoded records. In particular, for the current minimal key, there could be *EagerSH*- and *LazySH*-encoded records in the regular input buffer, in addition to already decoded records (from previous Reduce calls) in Shared. To correctly deal with such cases, AntiReducer's Reduce function first iterates over all values in the reducer's input in order to decode them. For *EagerSH*-encoded records such as (key1, (key2,..., val)), it also inserts (key1, val) into Shared. Hence after the "for all val in values" loop, Shared contains

all key-value pairs that the Reduce call in the original program would have received for that key.

Shared.popMinKeyValues removes the key from Shared's min-heap and all its associated values from the hash-table, passing an iterator for the removed values to the o_reducer.reduce call. The grouping comparator is used to determine key equality, ensuring that Shared's behavior is consistent with the original MapReduce program when the user provides a grouping comparator that is different from the regular key comparator, e.g., for secondary sort [26]. (Since records are removed from Shared in key order, the values passed to o_reducer.reduce are in key order.)

Notice that a Combiner is defined as a reducer class. Hence we apply the same syntactic transformation to it. Like for mapper and reducer, in the given program, the statement setting the Combiner class then can be changed to select the Anti-Combining enabled version. Since the Combiner is optional, Anti-Combining has a second parameter (in addition to T), a flag C that lets the user disable the Combiner in the map phase by setting $C = 0$.

6.2 Anti-Combining in Practice

Anti-Combining can be enabled for any MapReduce program, because it treats the original units of functionality (mapper, reducer, Combiner, Partitioner) as blackboxes. However, there are cases that require more analysis:

Non-determinism. Non-determinism affects *LazySH* because the re-executed Map and getPartition function might return different results. Whenever non-determinism in Map or getPartition can affect the Map output keys or their assignment to reduce tasks, *LazySH* must be disabled. The user disables it by setting threshold $T = 0$ when she suspects such effects of non-determinism. In practice we have not encountered examples for this type of application.

Programs without Combiner. If a program has no Combiner, then Anti-Combining can be safely enabled. *EagerSH* in the worst case, i.e., when there are no sharing opportunities, would add an insignificant overhead due to the additional bits needed to indicate the encoding of a record. All critical steps—write Map output to local disk, sort/merge it locally, transfer it to reducers, merge it on reducer, read it in Reduce call—become less costly thanks to the smaller encoded Map output. The additional cost on the reducers for decoding and managing Shared is the same or less compared to the reduction in cost on the mappers for writing sorted Map output to local disk and merging it there. Our experiments support this analysis. For *LazySH* the analysis is similar, except that expensive Map and getPartition calls can result in significantly higher CPU cost. As our experiments show, the user can effectively control the tradeoff between data size reduction and CPU cost increase through parameter T.

Combiner on or off. For programs with Combiner, the user can turn it off by setting Anti-Combining flag C accordingly. (This will only turn it off in the map phase, but still use it in the reduce phase.) If the Combiner results in small data reduction in the original program, e.g., less than 20%, then it should be turned off. The reason is that it decodes the Anti-Combining encoded records, i.e., undoes Anti-Combining, without delivering significant data reduction. Somewhat surprisingly, if a Combiner is highly effective, e.g., reduces data transfer by a factor of 10, then it will also benefit from Anti-Combining. Our experiments show this for Word Count. Intuitively the reason is that Map writes smaller encoded output (compared to the original program) and the Combiner reads these smaller data. The Shared data structure remains small because the effective Combiner reduces data size as it decodes. In general the decision about turning the Combiner off can be made by running the program with and without Combiner on a sample of input file splits, choosing the winner based on this sample run.

Partitioner. Careful design of a Partitioner can increase the impact of Anti-Combining by assigning records with commonalities to the same reduce task. Defining an appropriate Partitioner is the responsibility of the programmer who needs to analyze the statistical properties of the Map output. This is part of the MapReduce program design process already, because the effectiveness and scalability of parallel programs in general depend on a good partitioner to distribute records over reduce tasks. Automatically finding a good partitioner is beyond the scope of this paper since different data properties and applications lead to different sophisticated partitioning techniques [29, 19, 10]. In our experiments we demonstrate that even easy-to-design Partition functions already lead to significant cost savings.

Total cost versus running time. Anti-Combining reduces network transfer to lower the stress on this crucial shared resource. As a side-effect, writing and reading the smaller encoded data locally on mappers and reducers often also decreases local I/O and CPU cost. Hence we often see improvements not only in total cost, but also in running time. The main exception are cases where Anti-Combining introduces additional skew because of *LazySH* encoding. In particular, a reducer dealing with many *LazySH* encoded records might receive a large share of additional CPU and local I/O cost (for spilling Shared to disk). This skew is not a concern when the goal is to optimize for throughput. While the overloaded machine delays job completion, other machines finish early and can be used by other jobs in the cluster. Also, by choosing a smaller threshold T, the user can control how aggressively she wants to optimize for lower cost (and hence higher throughput) at the cost of potentially longer job completion time.

7. EXPERIMENTAL EVALUATION

All experiments were performed on a 12-machine cluster running Hadoop 1.0.3 [1]. One machine served as the head node, while the other 11 were worker nodes. Each machine has a single quad-core Xeon 2.4GHz processor, 8MB cache, 8GB RAM, and two 250 GB 7.2K RPM SATA hard disks. In total, the cluster therefore has 44 worker cores with 2GB memory per core available for map and reduce tasks. Unless stated otherwise, the number of reduce tasks is set to 44 to finish the reduce phase in one wave. Distributed file system block size is 64 MB and all machines participate as storage nodes.

All machines are directly connected to the same Gigabit network switch. Notice that this configuration of comparably few machines connected to a fast network and single switch is a challenging setup for Anti-Combining, which focuses on reducing network cost. In larger data centers with more machines and multi-hop communication between them, Anti-Combining will deliver even more benefits.

We compare the performance of the original program written by an experienced MapReduce programmer (*Original*)

against the transformed version with Anti-Combining. For Anti-Combining, we compare a version that only uses *EagerSH*, another that only uses *LazySH*, and the adaptive version *AdaptiveSH* (Section 6). The following data sets were used:

`QLog` contains 140 million real queries issued to a commercial search engine between 03/01/2011 and 04/30/2011. Each input record consists of an anonymous user identifier, the search query, and two query features (total number of occurrences of the query in search logs, total number of resulting links users browsed). The average search query string consists of 19.07 characters and the total data size is 4.3GB.

`ClueWeb09` is a real data set containing the first English segment of a web crawl by Carnegie Mellon University. There are around 50 million documents and 1.4 billion links, resulting in a total data size of 7GB.

`Cloud` is a real data set containing extended cloud reports from ships and land stations [11]. There are 382 million records, each with 28 attributes, resulting in a total data size of 28.8GB.

`RandomText` is a synthetic data set containing randomly generated text records with around 360GB total size.

7.1 Anti-Combining Overhead Analysis

We measure the overhead of Anti-Combining for workloads where it is ineffective. This is done for the Hadoop Sort program on `RandomText`. Sorting emits a single Map output record for each input record, therefore Anti-Combining is not beneficial. In all cases our adaptive algorithm automatically chooses *EagerSH* encoding without any shared keys, which degenerates to the original record plus a few bits that are needed to flag the type of encoding used.

AdaptiveSH results in only 0.2% more total disk read/write than the *Original* program due to the extra bits used for the encoding type flag. This also results in 0.15% more total data transfer cost. The total CPU time spent using *AdaptiveSH* is 7.8% more than the original program which is the overhead of looking for sharing opportunities among Map output records. The total runtime increased by 1.7% when *AdaptiveSH* is used. This supports our claim that Anti-Combining incurs little overhead even when it is applied to a job that does not benefit from it.

7.2 Query-Suggestion

We study Anti-Combining for the Query-Suggestion problem on `QLog` and explore the effect of the choice of Partitioner by comparing three alternatives. The first corresponds to an inexperienced programmer who simply relies on the standard hash Partitioner (*Hash*). The second assumes a programmer who understands the behavior of Map and realizes that maximal sharing is possible when all keys with the same first letter are sent to the same partition (*Prefix-1*). (Figuring this out is actually not that difficult, because sharing opportunities are determined by the Map and Partition functions and do not require complex program analysis.) Finally, Partitioner *Prefix-5* uses the first five characters. It addresses the concern that *Prefix-1* might generate only a small number of reduce tasks, equal to the number of distinct first characters of search queries in `QLog`. Note that none of these functions are specially designed based on input data statistics, which may be easily collected by a programmer. More careful data analysis may even lead to better partition functions that increase sharing opportunities.

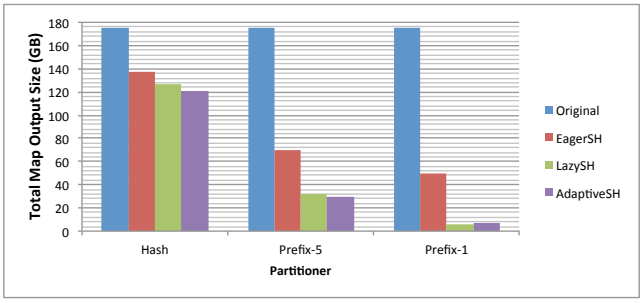


Figure 9: Total Map Output Size for Query-Suggestion

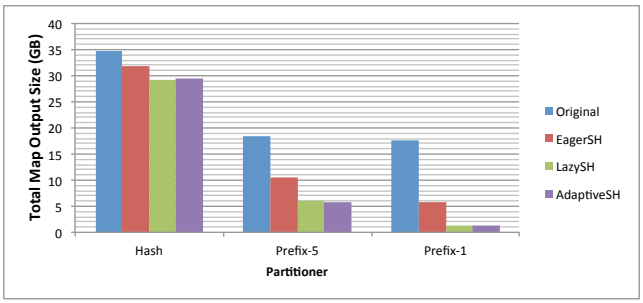


Figure 10: Total Map Output Size for Query-Suggestion using Combiner and Compression

Figure 9 shows the total Map output size for each strategy and partition function. Note that *Original* produces the same output size with all partition functions since it does not exploit sharing opportunities. On the other hand, *EagerSH* and *LazySH* effectively reduce the amount of data transferred for **all** partition functions, achieving reduction up to a factor of 27. Our adaptive strategy, *AdaptiveSH*, achieves the best result in all cases (except for *Prefix-1*) as expected since it adaptively picks the most suitable encoding type per record. For *Prefix-1*, *AdaptiveSH* encodes all map output records using *LazySH* encoding but produces slightly larger output than pure *LazySH* due to the extra bits used for identifying the encoding type used.

7.3 Query-Suggestion With Combiner

We repeated the same experiments on `QLog`, but now for the case that the original program came with a Combiner. The Combiner was not effective, reducing the total Map output size of *Original* by only about 12% compared to the no-Combiner result in Figure 9. Since the Combiner was not effective, we set $C = 0$, i.e., turned it off for Anti-Combining (see Section 6.2). Hence the Map output size did not change for any of the Anti-Combining strategies compared to Figure 9. The Combiner was, however, used in the Anti-Combining reduce phase and turned out to be highly effective in reducing the size of Shared so that virtually no spilling of Shared to disk occurred (see Section 5).

7.4 Query-Suggestion With Compression

We provide results for the same set of experiments, but now with both Combiner and Map output compression enabled. **We tried all of Hadoop's standard compression algorithms** and report the results for gzip, which achieved lower total CPU time with top- or near-top compression rate

Table 1: Total Cost Breakdown for Prefix-5, using different Compression Techniques

	Deflate	Gzip	Bzip2	Snappy	*AdaptiveSH* with Gzip
Total Disk Read(GB)	65	65	56	105	15
Total Disk Write(GB)	82	82	70	133	21
Total Map Output Size(GB)	18	18	15	30	6
Total CPU Time(1000 sec)	126.9	125.2	332.4	77.4	27.9

Table 2: Total Cost Breakdown of Query-Suggestion

Algorithm	Total CPU Time (1000 sec)	Disk Read (GB)	Disk Write (GB)
Original	168.8	566.1	741.5
Original-CB	172.9	510.4	664.6
Original-CP	125.2	64.5	82.3
AdaptiveSH	30.8	150.8	179.9
AdaptiveSH-CB	20.8	61.9	84.9
AdaptiveSH-CP	27.9	15	20.6

for *Original*. As Figure 10 shows, compression significantly reduces the total amount of data transferred for all strategies. Anti-Combining still performs better than *Original* for all partition functions, showing that it works well in combination with compression. As Table 1 highlights, compared to traditional compression alone, Anti-Combining's lightweight but effective data encoding not only lowers data transfer, but also local disk and CPU cost.

7.5 Effect on Disk I/O and CPU

The cost breakdown for total disk read/write and total CPU time of the Query Suggestion experiments using *Prefix-5* are shown in Table 2. Suffixes "-CB" and "-CP" refer to the experiments when the algorithms are executed using Combiner and compression, respectively.

AdaptiveSH effectively reduces the total amount of disk I/O, achieving reduction up to a factor of 3.8 and 4.1 for total amount of reads and writes respectively. Although Anti-Combining performs extra Map calls in the Reduce phase, it also reduces the total CPU time by a factor of 5.5. We believe that *Original* suffers from low CPU utilization because of the large amount of disk I/O and network transfer. Note that total CPU time measured for *Original-CP* is also lower than *Original* although CPU intensive compression is performed on the Map output buffer.

The Shared data structure in *AdaptiveSH* spills the data to disk 1575 times. On the other hand, *AdaptiveSH-CB* effectively applies Combine on the Shared data structure in the Reduce phase and manages to keep all unique keys and aggregated values in-memory. This reduces the total amount of disk reads and writes by a factor of 2.4 and 2.1, respectively.

7.6 CPU Intensive Workloads

We analyze the performance of Anti-Combining for CPU intensive workloads by adding extra CPU intensive calls to the Map function. As explained in Section 6.1, *AdaptiveSH* performs runtime cost-based optimization and switches to *EagerSH* encoding in order to avoid re-executing expensive Map calls in the reduce phase. We measure the performance of Anti-Combining using two extreme runtime thresholds: *Adaptive-0*, with runtime threshold $T = 0$ that results in *EagerSH* encoding for all Map output records and *Adaptive-*∞ with $T = \infty$ that does not restrict the choice of eager vs. lazy. We also provide results for *Adaptive-*α which uses a 400 microseconds runtime threshold. We modified the Map function of Query-Suggestion in order to add extra CPU intensive work by computing Fibonacci numbers.

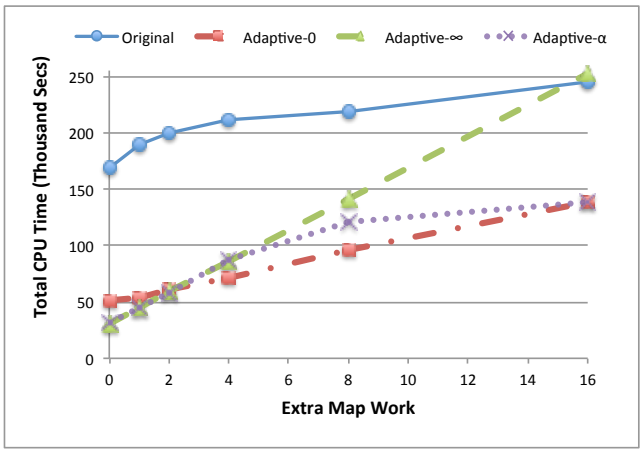


Figure 11: Total CPU Time using Runtime Cost-Based Optimization

Figure 11 shows the effect of increasing Map function call cost on the total CPU time for all algorithms. The x-axis represents the amount of busy work added. When x_i extra work is added, each map call computes the first $25000 \times x_i$ Fibonacci numbers. For low Map call cost, *Adaptive-*∞ achieves lower total CPU time as expected by optimizing for Map output size. As the Map call cost increases, executing the Map function in the Reduce phase for *LazySH* encoded records increases as well, resulting in higher total CPU time increase for *Adaptive-*∞. The area between the *Adaptive-0* and *Adaptive-*∞ plots represents the space of runtime thresholds that could be assigned by an optimizer. As expected, *Adaptive-*α uses lazy encoding where beneficial for low cost Map calls and converges to *Adaptive-0* as the Map function gets more expensive. The experiments show that Anti-Combining can be optimized for the dominating cost of a MapReduce job. In the next section, we also show effectiveness of Anti-Combining on two different CPU intensive applications.

7.7 Anti-Combining on Diverse Workloads

We evaluate Anti-Combining on a diverse set of workloads: WordCount, PageRank, and Join Processing.

7.7.1 Word Count

We study Anti-Combining for **WordCount** on `RandomText`. Map emits $(\text{word}, 1)$ for each word processed. Before the data is sent over the network, Combine computes the partial sum for each word in the map task. Reduce aggregates the partial counts per word. Note that Combine effectively reduces the total amount of data transferred to 92 MB (input size is 360 GB). Although it may seem that Anti-Combining is not necessary, the WordCount workload has high CPU utilization during the map stage [12]. Therefore, we explore the effect of Anti-Combining on the total CPU time spent, and total amount of disk read/write.

AdaptiveSH effectively reduces the total amount of disk read/write, achieving reduction by a factor of 9.1 and 6.3 for total disk reads and writes, respectively. Moreover,

AdaptiveSH encoding reduces the number of Map output records (before Combine is called) by a factor of 7. Therefore, the reduction in total disk I/O and number of records to be sorted locally results in reduced total CPU time spent by a factor of 1.7. *AdaptiveSH* also reduces the total runtime by a factor of 1.44. The total amount of data shuffled across the network is only 8MB larger than *Original* which is due to the encoding type flags used. The results show that Anti-Combining effectively reduces the dominating cost of the workload even when the Combiner is highly effective.

7.7.2 Page Rank

We evaluated the effectiveness of Anti-Combining for **PageRank** by computing 5 iterations on `ClueWeb09`. At each iteration, the map phase processes each document, dividing up evenly each document's page rank for outgoing edges (links) and emitting each outgoing edge with its page rank contribution. Reduce aggregates these contributions along incoming edges for each document.

AdaptiveSH reduces the total amount of data shuffled across the network by a factor of 2.7. The total amount of disk read/write is also reduced by a factor of 3.5 and 3.2 for total disk reads and writes, respectively. As expected, these reductions also result in reduced total CPU time by a factor 2.8. *AdaptiveSH* also reduces the total runtime by a factor of 2.4.

7.7.3 Join Processing

Finally, we study the performance of Anti-Combining for join processing using the following query on the `Cloud` data:

```
SELECT S.date, S.longitude, S.latitude, T.latitude
FROM Cloud AS S, Cloud AS T
WHERE S.date = T.date
      AND S.longitude = T.longitude
      AND ABS(S.latitude - T.latitude) <= 10
```

For our experiments, we use the memory-aware version of the 1-Bucket-Theta algorithm [19]. This algorithm ensures that data chunks are just small enough so that each local join task on a reducer can be executed in-memory. Optimization opportunities arise because each input record might be assigned to multiple chunks to enable parallel processing.

Figure 12 compares the total Map output size produced by each algorithm. Results for *LazySH* are not reported, because *AdaptiveSH* ended up choosing *LazySH* encoding for all map output records. Similar to the results in the theta-join paper [19], the creation of data chunks for parallel join processing causes an average data replication by a factor of 67 when comparing total map phase output to input. Since the join does not admit the use of a Combiner, this huge amount of data replication directly affects *Original*, producing 926 GB of Map output. The adaptive Anti-Combining technique reduces Map output by a factor of 9.5.

Since a Combiner was not an option for the join, we repeated the same experiments using compression on Map output data (denoted by suffix "-CP"). All techniques result in similar compression rates. Note that Map output size of *Original* with compression is still significantly larger than for Anti-Combining without compression.

Figure 12 also compares the runtime for each technique, showing that Anti-Combining improved over *Original* by a factor of 9.6 and 6 for the no-compression and compression scenarios, respectively. Notice that the 1-Bucket-Theta algorithm achieves almost perfect load balancing between the different worker nodes. Hence the runtime improvement tracks closely the reduction in Map output size.

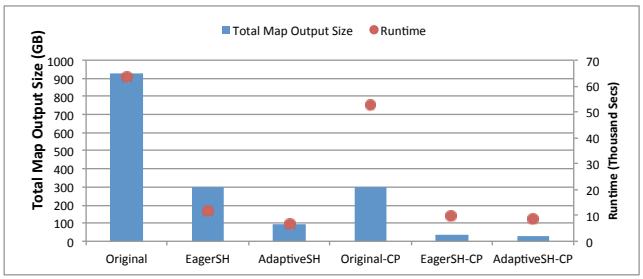


Figure 12: Total Map Output Size and Runtime for Theta-Join Query

8. RELATED WORK

Multi-query optimization [22] aims at reducing query cost by sharing data and computation. Data sharing was demonstrated to be beneficial also for MapReduce workloads. Multiple jobs that share the same input are merged into a single job in Pig to avoid redundant I/O [8]. Hive [23] follows a similar approach to share input among multiple jobs. Agrawal et al. [4] study how to schedule scans of large data files when there are many simultaneous requests to a common set of files. CoScan [25] is another scheduling framework to merge multiple jobs working on the same datasets while trying to meet individual job deadlines. Wolf et al. [27] propose a scheduler where jobs are decomposed into sub-jobs and cyclic piggybacking is performed instead of batching for sharing scans. Prior work also focused on improving MapReduce by sharing data and computation for iterative tasks. Haloop [5] provides a MapReduce based framework that improves execution of iterative jobs by supporting data re-use. Restore [7] also supports data re-use among MapReduce jobs in a workflow to speed up future workflows executed in the system. Our work is complementary to these data sharing approaches. In fact, they create additional opportunities for Anti-Combining when shared data has to be **transmitted to multiple queries**.

MRshare [18] proposes cost-based optimization for data sharing in MapReduce. In addition to input sharing, Map output sharing is also studied. However, intermediate data sharing is limited to overlapping parts of the map output generated from shared input of multiple jobs. Therefore, cases where the Map function produces multiple key/value pairs for a single job cannot be addressed. In addition, our techniques enable sharing of non-overlapping parts of the Map output by pushing the Map operator to the reduce phase. YSmart [15] translates SQL queries into MapReduce jobs and exploits correlations among the operators in the query plan. In addition to input sharing, intermediate data sharing is also studied. However, intermediate data is shared only among operators having the same key. Therefore, this approach also does not solve the problem introduced in our work. Jahani et al. [13] propose a static analysis mechanism for automatic detection of selection, projection, and data compression optimizations in MapReduce programs. Pro-

posed data compression optimization aims to work directly on compressed data where applicable. Delta-compression is also used which is only applicable to numeric datasets where sequential data values change slightly.

9. CONCLUSIONS

We proposed Anti-Combining, a novel approach for reducing the amount of data transferred between mappers and reducers. It can be enabled in any MapReduce program by applying the appropriate (purely syntactic) transformations, but will be most effective for problems where the shuffle-and-sort phase dominates the overall cost. Anti-Combining shifts mapper-side processing to the reducers and is much more lightweight than general compression techniques.

Anti-Combining can be used together with existing database-style optimizations such as sharing of scans and intermediate results. Since it does not need to understand the semantics of a given MapReduce program, it is perfectly suited for adding dynamic optimizations to statically optimized MapReduce programs generated by compilers for high-level languages such as PigLatin and HiveQL.

In our future work, we plan to explore extensions that allow optimization not only for the input of a single Map call, but also across all Map calls in the same map task.

10. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant No. IIS-1017793. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

11. REFERENCES

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE*, pages 498–509, 2012.

[3] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[4] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *PVLDB*, 1(1):958–969, 2008.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.

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

[7] I. Elghandour and A. Aboulnaga. Restore: reusing results of mapreduce jobs. *PVLDB*, 5(6):586–597, 2012.

[8] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[9] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[10] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *ICDE*, pages 522–533, 2012.

[11] C. Hahn and S. Warren. Extended edited synoptic cloud reports from ships and land stations over the globe, 1952-1996.

[12] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDEW*, pages 41–51, 2010.

[13] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, Mar. 2011.

[14] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, Sept. 1999.

[15] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, pages 25 –36, 2011.

[16] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[17] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10):1016–1027, 2012.

[18] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: sharing across multiple queries in mapreduce. *PVLDB*, 3(1-2):494–505, 2010.

[19] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[22] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[23] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[24] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[25] X. Wang, C. Olston, A. D. Sarma, and R. Burns. Coscan: cooperative scan sharing in the cloud. In *SOCC*, pages 11:1–11:12, 2011.

[26] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 3rd edition, 2012.

[27] J. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.-L. Wu, and R. Vernica. On the optimization of schedules for mapreduce workloads in the presence of shared scans. *The VLDB Journal*, 21:589–609, 2012.

[28] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, pages 38–49, 2012.

[29] J. Zhou, N. Bruno, and W. Lin. Advanced partitioning techniques for massively distributed computation. In *SIGMOD*, pages 13–24, 2012.