MapReduce with Deltas

R. Lämmel and D. Saile
Software Languages Team, University of Koblenz-Landau, Germany

Abstract— *The MapReduce programming model is extended conservatively to deal with deltas for input data such that recurrent MapReduce computations can be more efficient for the case of input data that changes only slightly over time. That is, the extended model enables more frequent re-execution of MapReduce computations and thereby more up-to-date results in practical applications. Deltas can also be pushed through pipelines of MapReduce computations. The achievable speedup is analyzed and found to be highly predictable. The approach has been implemented in Hadoop, and a code distribution is available online. The correctness of the extended programming model relies on a simple algebraic argument.*

Keywords: MapReduce; Delta; Distributed, Incremental Algorithms

1. Introduction

We are concerned with the MapReduce programming model [1], which is widely used for large-scale data processing problems that can benefit from massive data parallelism. MapReduce is inspired by functional programming idioms, and it incorporates specific ideas about indexing and sorting; see [2] for a discussion of the programming model. There exist several proprietary and open-source implementations that make MapReduce available on different architectures.

Research question

The problem of crawling the WWW may count as the archetypal application of MapReduce. A particular crawler may operate as follows: web sites are fetched; outlinks are extracted; accordingly, more web sites are fetched in cycles; a database of inverse links ("inlinks") is built to feed into page ranking; eventually, an index for use in web search is built; see Fig. 1 for the corresponding workflow.

In many MapReduce scenarios (including the one of crawling and indexing), the question arises whether it is possible to achieve a speedup for recurrent executions of a MapReduce computation by making them incremental.

A crawler is likely to find about the same pages each time it crawls the web. Hence, the complete re-computation of the index is unnecessarily expensive, thereby limiting the frequency of re-executing the crawler as needed for an up-to-date index. A more up-to-date index is feasible if the index is incrementally (say efficiently) updated on the grounds of the limited changes to the crawl results.

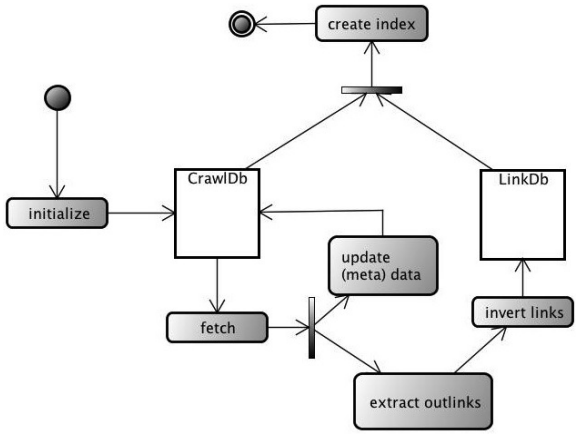

Fig. 1: Workflow of a simple crawler with indexing

Contributions

- The MapReduce programming model is enhanced to explicitly incorporate deltas of inputs of recurrent MapReduce computations. This enhancement is based on a simple algebraic insight that has not been exploited elsewhere.
- Based on benchmarks for delta-aware MapReduce computations, it is found that deltas are of limited use when used naively, but they provide substantial, predictable speedups—when applying specific techniques for computing deltas and merging them with previous results.

Our implementation and corresponding measurements are based on Apache's Hadoop [3]—an open-source implementation of MapReduce which targets clusters of networked computers with a distributed file system. A code distribution is available online through the paper's website.[1].

Road-map

Sec. 2 expands on the introductory problem of crawling and indexing, thereby clarifying the motivation of our work. Sec. 3 rehashes MapReduce in a way that is specifically suitable for initiating a discussion of deltas. Sec. 4 extends the MapReduce programming model to incorporate deltas. Sec. 5 discusses different options for computing deltas. Sec. 6 defines and executes benchmarks for delta-aware MapReduce computations. Sec. 7 discusses related work. Sec. 8 concludes the paper.

2. Motivation

Crawling without deltas Any search engine relies on one or more indexes that are computed from information that is

[1] `http://softlang.uni-koblenz.de/deltamr`

obtained by web crawls. A typical crawler, such as *Nutch* [4], performs several tasks that can be implemented as a *pipeline* of MapReduce jobs; we refer again to Fig. 1 for a simple workflow for crawling and indexing. The crawler maintains a database, *CrawlDb*, with (meta) data of discovered websites. Before crawling the web for the first time, *CrawlDb* is initialized with seed URLs. The crawler performs several cycles of fetching. In each cycle, a *fetch list* (of URLs) is obtained from *CrawlDb*. The corresponding web sites are downloaded and *CrawlDb* is updated with a time stamp and other data. Further, the crawler extracts outlinks and aggregates them in *LinkDb* so that each URL is associated with its inlinks. The resulting reverse web-link graph is useful, for example, for ranking sites such as with *PageRank* [5]. Eventually, *CrawlDb* and *LinkDb* are used to create an index, which can be queried by a search engine.

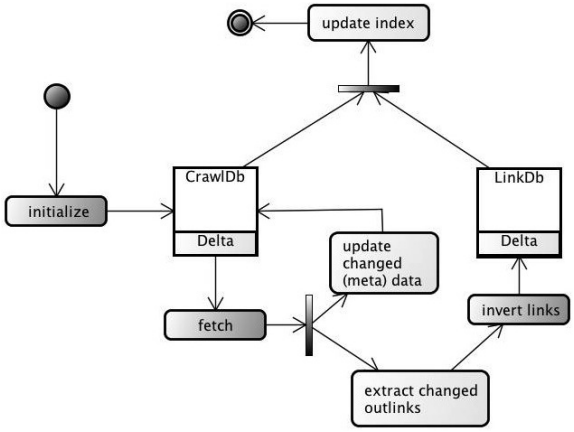

Fig. 2: Crawler using deltas

Crawling with deltas Suppose only a small fraction of all web sites changes. Then it can be more efficient to determine those changes (say, "deltas") and to update the index accordingly. Fig. 2 revises the simple crawler from Fig. 1 so that deltas are used in several positions. That is, in each crawl cycle, a delta of changed sites is determined and corresponding deltas of outlinks, *CrawlDb*, and *LinkDb* are derived so that the index can be updated incrementally.

3. A simple view on MapReduce

For the rest of the paper, we will not deal with the complex scenario of crawling and indexing. We resort to "the problem of counting the number of occurrences of each word in a large collection of documents" [1]. In sequential, imperative (pseudo) code, the problem is solved as follows:

Input: a collection of uri-document pairs c
Output: a map m from words to counts
Algorithm:
```
for each ⟨u, d⟩ in c do
  for each w in words(d) do
    m[w] = m[w] + 1; // m[w] is initially 0.
```

Fig. 3: Sequential, imperative word-occurrence count

This direct approach does not stipulate massive parallelism for iterating over c because of the use of a global data structure for the map (say, dictionary) m. The aspects of data extraction and reduction are to be separated. Extraction is supposed to produce a stream of word-occurrence counts as follows:

Input: a collection of uri-document pairs c
Output: a stream s of words-occurrence counts
Algorithm:
```
for each ⟨u, d⟩ in c do
  for each w in words(d) do
    yield ⟨w, 1⟩;
```
// per-document extraction

Fig. 4: Extraction amenable to parallelism and distribution

(The role of the boxed code is explained in a second.) In general, extraction returns a stream of key-value pairs to be reduced eventually (see below). In the example, words are keys and counts are values. The intermediate stream can be produced in a massively parallel manner such that input partitions are assigned to nodes in a cluster of machines. Subject to a distributed file system, the partitions may be readily stored with the nodes that process them.

Reduction requires grouping of values by key:

Input: a stream s of key-value pairs
Output: a map (say, a dictionary) m' of key-list pairs
Algorithm:
```
for each ⟨k, v⟩ in s do
  m'[k] = append(m'[k], v); // m'[k] is initially the empty list.
```

Fig. 5: Group key-value pairs

Reduction commences as follows:

Input: a map m' of key-list pairs
Output: a map m from words to counts
Algorithm:
```
for each ⟨k, g⟩ in m' do {
  r = 0;
  for each v in g do
    r = r + v;
  m[k] = r;
}
```
// per-key reduction

Fig. 6: Reduction amenable to parallelism and distribution

(The role of the boxed code is explained in a second.) Grouping and reduction can be distributed (parallelized) by leveraging the fact that the key domain may be partitioned.

The original sequential description of Fig. 3 is much more concise than the sliced, parallelism-enabling development of Fig. 4–6. However, it is easy to realize that most of the code is problem-independent. In fact, the only problem-specific code is the one that is boxed in Fig. 4 and Fig. 6. That is, the first box covers data extraction at a fine level of granularity; the second box covers data reduction per intermediate key. In practice, MapReduce computations are essentially specified in terms of two functions *mapper* and *reducer*:

```
function mapper(u, d) {
  for each w in words(d) do
    yield ⟨w, 1⟩;
}
function reducer(k,g) {
  r = 0;
  for each v in g do r = r + v;
  return r;
}
```

Fig. 7: The functionality for word-occurrence counting

Summary MapReduce computations extract intermediate key-value pairs from collections of input documents or records. Such extraction can be easily parallelized if input data is readily partitioned to reside on machines in a cluster. The resulting intermediate key-value pairs are to be grouped by key. The key domain is partitioned so that parallelism can be applied for the reduction of values per key. MapReduce implementations allow the specification of the number of mapper and reducer nodes as well the specification of a *partitioner* that associates partitions of the intermediate key domain with reducers.

4. MapReduce with deltas

Deltas The input for MapReduce computations is generally a keyed collection, in fact, an ordered list [1]. Given two generations of input data i and i', a delta $\Delta_{i,i'}$ can be defined as a quadruplet of the following sub-collections:

$\Delta_{i'_+}$	Part of i' with keys not present in i.
Δ_{i_-}	Part of i with keys not present in i'.
$\Delta_{i_\neq}$	Part of i whose keys map to different values in i'.
$\Delta_{i'_\neq}$	Part of i' whose keys map to different values in i.

The first part corresponds to added key-value pairs; the second part corresponds to removed key-value pairs; the third and fourth parts correspond to modified key-value pairs ("before" and "after"). Modification can be modeled by deletion followed by addition. Hence, we simplify $\Delta_{i,i'}$ to consist only of two collections:

$$\begin{aligned} \Delta_+ &= \Delta_{i'_+} + \Delta_{i'_\neq} \\ \Delta_- &= \Delta_{i_-} + \Delta_{i_\neq} \end{aligned}$$

The simple but important insight is that MapReduce computations can be applied to the parts of the delta and combined later with the result for i so that the result for i' is obtained more efficiently than by computing i' naively.

Algebraic requirements Correctness conditions are needed for the non-incremental and incremental execution to agree on the result. This is similar to the correctness conditions for classic MapReduce that guarantee that different distribution schedules all lead to the same result.

In the case of classic MapReduce, the mapper is not constrained, but the reducer is required to be (the iterated application of) an associative operation [1]. More profoundly, reduction is *monoidal* in known applications of MapReduce [2], [6]. That is, reduction is indeed the iterated application of an associative operation "$\bullet$" with a unit u. In the case of the word-occurrence count example, reduction iterates *addition* "$+$" with "0" as unit. The parallel execution schedule may be more flexible if commutativity is required in addition to associativity [2].

Additional algebraic constraints are needed for MapReduce computations with deltas. That is, we require an *Abelian group*, i.e., a monoid with commutativity for "$\bullet$" and an operation "$\bar{\cdot}$" for an inverse element such that $x \bullet \overline{x} = u$ for all x. In the case of the word-occurrence count example, addition is indeed commutative, and the inverse element is to be determined by negation. Hence, we assume that MapReduce computations are described by two ingredients:

- A mapper function—as illustrated in Fig. 7.
- An Abelian group—as a proxy for the reducer function.

MapReduce computations with deltas We are ready to state a law (without proof) for the correctness of MapReduce computations with deltas. Operationally, the law immediately describes how the MapReduce result for i needs to be updated by certain MapReduce results for the components of the delta so that the MapReduce result for i' is obtained; the law refers to "$\bullet$"—the commutative operation of the reducer:

$$\begin{aligned} MapReduce(f,g,i') &= MapReduce(f,g,i) \\ &\bullet\ MapReduce(f,g,\Delta_+) \\ &\bullet\ MapReduce(\overline{f},g,\Delta_-) \end{aligned}$$

Here, f is the mapper function, g is an Abelian group, and $\overline{f}$ denotes lifted inversion. That is, if f returns a stream of key-value pairs, then $\overline{f}$ returns the corresponding stream with inverted values. In imperative style, we describe the inversion of extraction as follows:

```
Input: a stream s of key-value pairs
Output: a stream s' of key-value pairs
Parameter: an inversion operation ¯ on values
Algorithm:
    for each ⟨k, v⟩ in s do
      yield ⟨k, v̄⟩; // value-by-value inversion
```

Fig. 8: Lifted inversion

Fig. 9 summarizes the workflow of MapReduce computations with deltas. Clearly, we assume that we can compute deltas; see the node "Compute delta". Such deltas are then processed with the MapReduce computation such that deleted pairs are inverted; see the node "MapReduce′". One can either merge original result with the result for the delta, or one can propagate the latter to further MapReduce computations in a pipeline.

5. Computation of deltas

Deltas can be computed in a number of ways.

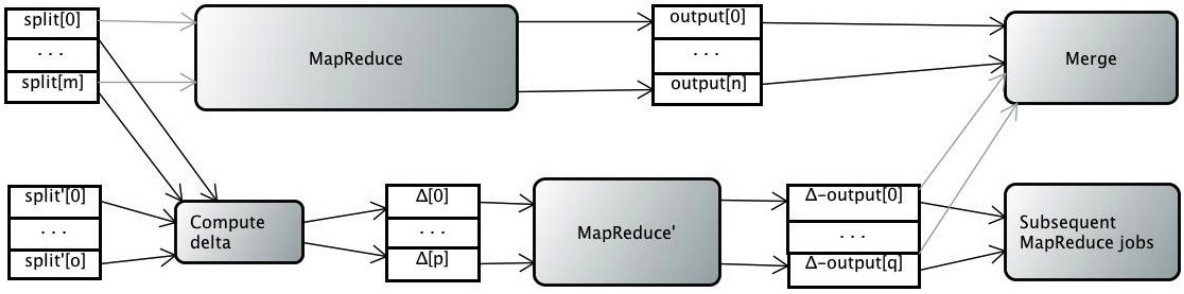

Fig. 9: MapReduce with deltas

MapReduce-based delta If we assume that both generations of input, i and i', have been regularly stored in the distributed file system, then the delta can be computed with classic MapReduce as follows:

```
Input: the concatenated input append(i, i')
Output: the (encoded) delta Δ_{i,i'}
Algorithm (MapReduce):
  function mapper(k, v) {
    if k in i then sign := "-"; else sign := "+";
    return ⟨k, ⟨sign, v⟩⟩; // attach sign
  }
  function reducer(k,g) {
    ⟨s1, v1⟩ := g.next();
    if ¬ g.hasNext()
      then return [⟨s1, v1⟩]; // "added" or "deleted"
      else {
        ⟨s2, v2⟩ := g.next();
        if v1 == v2
          then return [ ]; // "preserved"
          else return [⟨s1, v1⟩, ⟨s2, v2⟩]; // "modified"
      }
  }
```

Fig. 10: Computing a delta with MapReduce

The mapper qualifies the values of key-value pairs from i and i' with "-" and "+" respectively—for *potential* deletion or addition; see the condition "k in i" in the figure. Hadoop [3] and other implementations of MapReduce can discriminate between different input files in the map phase.

Reduction receives 1-2 values per original key depending on whether a key occurs in either i or i' or both. For simplicity, keys are assumed to be unique in each of i and i'. (Irregular cases require a slightly more advanced reduction.) In the case of a single value, a potential deletion or addition becomes definite. In the case of two values, two equal values cancel out each other, whereas two unequal values contribute to both deletion and addition.

Delta after iteration It is possible to aggressively reduce the volume of delta by exploiting a common idiom for MapReduce computations. That is, extraction is typically based on uniform, structural decomposition, say iteration. Consider the for-loop for extracting word-occurrence counts from documents—as of Fig. 7:

```
for each w in words(d) do
  yield ⟨w, 1⟩;
```

That is, the document is essentially decomposed into words from which key-value pairs are produced. Instead, the document may also be first decomposed into lines, and then, in turn, into words:

```
for each l in lines(d) do
  for each w in words(l) do
    yield ⟨w, 1⟩;
```

In general, deltas could be determined at all accessible levels of decomposition. In the example, deltas could be determined at the levels of documents (i.e., the values of the actual input), lines, and words. For the problem at hand, line-level delta appears to be useful according to established means for delta creation such as "text diff" [7]. MapReduce computations with deltas are easily configured to exploit different levels. When computing the delta, as defined in Fig. 10, the case "$\neg(v_1 == v_2)$" must be refined to decompose v_1 and v_2 and to compute the delta at the more detailed level. In implementations of MapReduce, one can indeed exercise different levels. For instance, Hadoop [3] assumes that MapReduce jobs are configured with "input formatters" which essentially decompose the input files.

Delta based on map-side join Overall, the costs of MapReduce-based computation of the delta are substantial. Essentially, both generations of input have to be pumped through the network so that a reducer can cancel out matching key-value pairs. These costs would need to be matched by the savings achievable through deltas in a MapReduce computation or a pipeline.

There is a relevant MapReduce-like abstraction, which can be used to drastically reduce network communication during delta computation. That is, *map-side join* [8], [9] can be used to map over multiple inputs simultaneously such that values with the same key but from different inputs are mapped together. To this end, the inputs must be equally sorted and partitioned so that matching partitions can be dealt with simultaneously. Network communication is reduced since no reduction is involved. Network communication is completely eliminated if matching partitions are available on the same machine. (Map-side join is available, for example, in Hadoop [3].) It is often possible to meet the requirements of map-side join. For instance, a crawler may be set up to write crawling results accordingly.

Streaming delta An even more aggressive optimization is to produce and consume the second generation of input data in streaming mode. Just as before, it is necessary to assume that both generations are sorted in the same manner. Such streaming is feasible for tasks that essentially generate "sorted" data. Streaming can be also used to *fuse* two MapReduce computations—as known from functional programming [10]. Compared to all other forms of computing deltas, streaming delta does not write (and hence not read) the second generation.

6. Benchmarking

We present simple benchmarks to compare non-incremental (say, classic) and incremental (say, delta-aware) MapReduce computations. We ran the benchmarks on a university lab.[2] The discussion shows that speedups are clearly predictable when using our method.

TeraByte Sort TeraByte Sort (or the variation—MinuteSort) is an established benchmark to test the throughput on a MapReduce implementation when using it for *sorting* with (in one typical configuration) 100-byte records out of which 10 bytes constitute the key [11], [12], [13]. The mapper and reducer functions for this benchmark simply *copy* all data. The built-in sorting functionality of MapReduce implies that intermediate key-value pairs are sorted per reducer. The partitioner is defined to imply *total ordering* over the reducers. Hadoop—the MapReduce implementation that we use—has been a winner of this benchmark in the past.

The established implementation of TeraByte Sort (see, e.g., [12], [13]) samples keys in the input from which it builds a trie so that partitioning is fast. Instead, our partitioner does not leverage any sampling-based trie because we would otherwise experience uneven reducer utilization for MapReduce jobs on sorted data. Here we note that we must process sorted data in compound MapReduce computations; see the discussion of pipelines below. We use datatype `long` (8 bytes) for keys instead of byte sequences of length 10, thereby simplifying partitioning.

Fig. 11 shows the benchmark results for TeraByte Sort. The "incremental" version computes the delta by a variation of Fig. 10. There are also optimized, incremental versions: (map-side) "join" and "streaming"—as discussed in Sec. 5. The shown costs for the incremental versions include *all* costs that are incurred by recomputing the same result as in the non-incremental version: this includes costs of computing the delta and performing the merge. It is important to note that we implement merge by map-side join.

It is not surprising that the non-incremental version is faster than all incremental versions except for streaming.

[2]Cluster characteristics: we used Hadoop version 0.21.0 on a cluster of 40 nodes with an *Intel(R) Pentium(R) 4 CPU 3.00GHz* and 2 x 512MB SDRAM and 6GB available disk space. All machines are running *openSUSE 11.2* with Java version *1.6.0_24* and are connected via a *100Mbit Full-Duplex-Ethernet* network.

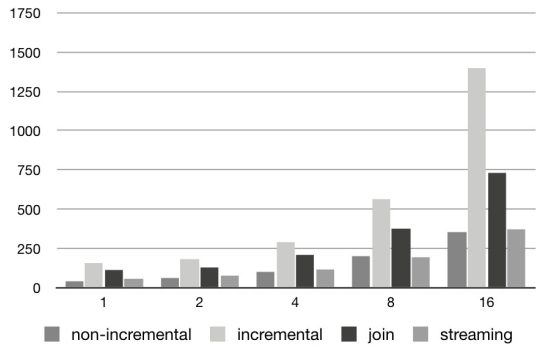

Fig. 11: **Runtimes in seconds (y-axis) for non-incremental and incremental TeraByte Sort for different input sizes in GB (x-axis) where the size of the deltas for the incremental versions is assumed to be 10 % of the input size.**

That is, computing a delta for data on files means that both generations are processed whereas non-incremental sorting processes only the new generation. Also, the merge performs another pass over the old generation and the (small) delta.

Streaming stays very close to the non-incremental baseline. Its costs consist of the following parts: read original input data on file and compare it with new input data available through streaming so that delta is written (15.3 %); process delta (20.8 %); merge processed delta with original output (63.9 %)—the percentages of costs are given for the rightmost configuration in Fig. 11. Essentially, merging original input and delta dominates the costs of streaming, but those costs are below the costs of processing the new input in non-incremental fashion because the former is a map-side join while the latter is a regular MapReduce computation.

Pipelines In practice, MapReduce jobs are often organized in pipelines or even more complicated networks—remember the use case of crawling in Sec. 2. In such compounds, the benefit of processing deltas as opposed to complete inputs adds up. We consider a simple benchmark that shows the effect of cumulative speedup. That is, four MapReduce jobs are organized in a pipeline, where the first job sorts, as described above, and the subsequent jobs simply copy. Here, we note that a copy job is slightly faster than a sort job (because of the eliminated costs for partitioning for total order), but both kinds of jobs essentially entail zero mapper/reducer costs, which is the worst case for delta-aware computations.

The results are shown in Fig. 12. The chosen pipeline is not sufficient for the "naive" incremental option to outperform the non-incremental option, but the remaining incremental options provide speedup. MapReduce-scenarios in practice often reduce the volume of data along such pipelines. (For instance, the counts of word occurrences require much less volume than the original text.) In these cases, costs for merging go significantly down as well, thereby further improving the speedup.

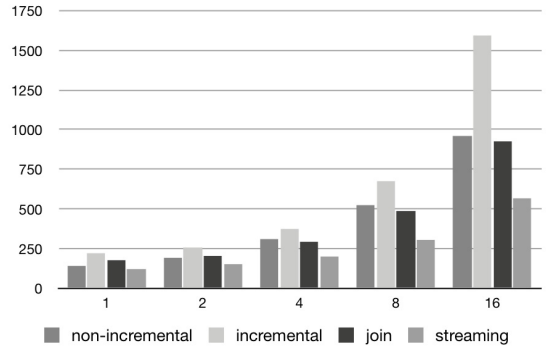

Fig. 12: **Sort followed by three copy jobs.**

7. Related work

An approach to update *PageRank* computations in the context of changes in the web is introduced by [14]. Similar to our approach, existing results are updated according to computed additions and deletions. However, the approach specifically applies to graph-computations, whereas our approach deals with incremental MapReduce computations in general.

Percolator [15] is Google's new approach in dealing with the dynamic nature of the web. Percolator is aimed at updating an existing index that is stored in BigTable [16], Google's high performance proprietary database system. Percolator adds trigger-like procedures to BigTable columns, that are triggered whenever data is written to that column in any row. The paper states that Percolator requires more resources than MapReduce and only performs well under low crawl rates (i.e., the new input is a small fraction of the entire repository). Our approach uses essentially the same resources than classic MapReduce. We do not understand well enough how to compare our speedups (relative to delta sizes and other factors in our approach) with Percolator's scalability (relative to crawl rates).

Twister [17], a distributed in-memory MapReduce runtime, is optimized for iterative MapReduce by several modifications to the original MapReduce model. Iterative jobs are run by a single MapReduce task, to avoid re-loading static data that does not change between iterations. Furthermore, intermediate data is not written to disk, but populated via distributed memory of the worker nodes. CBP, a system for *continuos bulk processing* [18], distinguishes two kinds of iterative computations: several iterations over the same input (e.g., PageRank), and iteration because of changed input (e.g., URLCount). CPB introduces persistent access to state re-use prior work along reduction. Our approach does not introduce state, which contributes to the simple correctness criterion for MapReduce computations with deltas. Our approach does not specifically address iterative computations, but instead it enables a general source for speedup for MapReduce computations.

Dryad [19], [20] is a data-parallel programming model like MapReduce, which, however, supports more general DAG structures of dataflow. Dryad supports reuse of identical computations already performed on data partitions and incrementality with regard to newly appended input data for which computed results are to be merged with previous results. While the idea of merging previous and new results is similar to deltas, our approach is not restricted to append-only scenarios.

Map-reduce-merge [21] enhances MapReduce to deal with multiple heterogenous datasets so that regular MapReduce results are merged in an extra phase. The enhanced model can express relational algebra operators and implement several join-algorithms to unite multiple heterogenous datasets. In contrast, the merge phase in our approach is a problem-independent element of the refined programming model which simply combines two datasets of the same structure.

For our implementation we used Hadoop [3], an open source Java implementation of Google's MapReduce framework [1]. Hadoop's MapReduce-component [22] is built on top of HDFS [23], the *Hadoop Distributed File System* which has been modeled after the *Google File System (GFS)* [24]. Hadoop happens to provide a form of streaming (i.e., *Hadoop Streaming*) for the composition of MapReduce computations [25]. This form of streaming is not directly related to streaming in our sense of delta computation.

MapReduce Online [26] is a modified MapReduce architecture which introduces pipelining between MapReduce jobs as well as tasks within a job. The concept is implemented as a modification of Hadoop. A more general stream-based runtime for cloud computing is *Granules* [27]. It is based on the general concept of *computational tasks*, that can be executed concurrently on multiple machines, and work on abstract datasets. These datasets can be files, streams or (in the future) databases. Computational tasks can be specialized to map and reduce tasks, and they can be composed in directed graphs allowing for iterative architectures. *Granules* uses *NaradaBrokering* [28], an open-source, distributed messaging infrastructure based on the publish/-subscribe paradigm, to implement streaming between tasks. We believe that such work on streaming may be helpful in working out streaming deltas in our sense.

Our programming model essentially requires that reduction is based on the algebraic structure of an Abelian group. This requirement has not been set up lightly. Instead, it is based on a detailed analysis of the MapReduce programming model overall [2], and a systematic review of published MapReduce use cases [6].

8. Conclusion

We have described a refinement of MapReduce to deal with incremental computations on the grounds of computing deltas, and merging previous results and deltas possibly throughout pipelines. This refinement comes with a simple correctness criterion, predictable speedup, and it can be

provided without any changes to an existing MapReduce framework. Our development is available online.

There are some interesting directions for future work.

The present paper focuses on the principle speedup and the correctness of the method. A substantial case study would be appreciated to reproduce speedup in a complex scenario. For instance, an existing WebCrawler could be migrated towards MapReduce computations with deltas.

Currently, we do not provide any reusable abstractions for streaming delta. In fact, the described benchmark for streaming TeraByte Sort relies on summation of assumed components of the computation, but we continue working on an experimental implementation.

Our approach to streaming delta and map-side join for merge may call for extra control of task scheduling and file distribution. For instance, results of processing the delta could be stored for alignment with the original result so that map-side join is most efficient.

As the related work discussion revealed, there is a substantial amount of techniques for optimizing compound data-parallel computations. While the art of benchmarking classic MapReduce computations has received considerable attention, it is much harder to compare the different optimizations that often go hand in hand with changes to the programming model. On the one hand, it is clear that our approach provides a relatively general speedup option. On the other hand, it is also clear that other approaches promise more substantial speedup in specific situations. Hence, a much more profound analysis would be helpful.

Modern MapReduce applications work hand in hand with a high performance database system such as BigTable. The fact that developers can influence the locality of data by choosing an appropriate table design, could enable very efficient delta computations. Database systems such as BigTable also offer the possibility to store multiple versions of data using timestamps. This could facilitate delta creation substantially.

Acknowledgment The authors are grateful for C. Litauer and D. Haussmann's support in setting up a MapReduce cluster at the University of Koblenz-Landau for the purpose of benchmarking.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004, pp. 137–150.

[2] R. Lämmel, "Google's MapReduce programming model—Revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, 2008.

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

[4] "Apache Nutch," http://nutch.apache.org/.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Digital Library Technologies Project, Tech. Rep., 1998.

[6] A. Brandt, "Algebraic Analysis of MapReduce Samples," 2010, Bachelor Thesis, University of Koblenz-Landau.

[7] J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Bell Laboratories, Tech. Rep., 1976.

[8] J. Venner, *Pro Hadoop*. Apress, 2009.

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

[10] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion: from lists to streams to nothing at all," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*. ACM, 2007, pp. 315–326.

[11] "Sort Benchmark," web site http://sortbenchmark.org/.

[12] O. O'Malley, "TeraByte Sort on Apache Hadoop," 2008, contribution to [11].

[13] A. C. Murthy, "Winning a 60 second dash with a yellow elephant," 2009, contribution to [11].

[14] P. Desikan and N. Pathak, "Incremental PageRank Computation on evolving graphs," in *Special interest tracks and posters of the 14th international conference on World Wide Web*, ser. WWW '05. ACM, 2005, pp. 10–14.

[15] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, 2010, pp. 1–15.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06, 2006, pp. 205–218.

[17] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010*. ACM, 2010, pp. 810–818.

[18] D. Logothetis, K. C. Webb, C. Olston, K. Yocum, and B. Reed, "Stateful Bulk Processing for Incremental Analytics," in *SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 51–62.

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

[20] L. Popa, M. Budiu, Y. Yu, and M. Isard, "DryadInc: Reusing work in large-scale computations," in *HotCloud'09 Proceedings of the 2009 conference on Hot topics in cloud computing*, 2009.

[21] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker Jr., "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2007, pp. 1029–1040.

[22] "Hadoop MapReduce," http://hadoop.apache.org/mapreduce/.

[23] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, The Apache Software Foundation, 2007.

[24] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. ACM, 2003, pp. 29–43.

[25] "Hadoop Streaming," http://hadoop.apache.org/common/docs/r0.15.2/streaming.html, 2008.

[26] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proceedings of the 7th Symposium on Networked Systems Design and Implementation*, ser. NSDI'10. USENIX Association, 2010, pp. 313–328.

[27] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for Map-Reduce," in *Proceedings of the 2009 IEEE International Conference on Cluster Computing*. IEEE, 2009, pp. 1–10.

[28] S. Pallickara and G. Fox, "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids," in *Proceedings of 2003 ACM/IFIP/USENIX International Middleware Conference*. Springer, 2003, pp. 41–61.