UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Diplomarbeit
MapReduce with Deltas

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

David Saile

Erstgutachter:	Prof. Dr. Ralf Lämmel Institut für Informatik, AG Softwaresprachen
Zweitgutachter:	Andrei Varanovich Institut für Informatik, AG Softwaresprachen

Koblenz, im August 2011

ERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Unterschrift)

ABSTRACT

The MapReduce programming model is extended slightly in order to use deltas. Because many MapReduce jobs are being re-executed over slightly changing input, processing only those changes promises significant improvements. Reduced execution time allows for more frequent execution of tasks, yielding more up-to-date results in practical applications. In the context of compound MapReduce jobs, benefits even add up over the individual jobs, as each job gains from processing less input data. The individual steps necessary in working with deltas are being analyzed and examined for efficiency. Several use cases have been implemented and tested on top of Hadoop. The correctness of the extended programming model relies on a simple correctness criterion.

CONTENTS

1 INTRODUCTION

As the volume of available digital information increases, for example through accessibility via the internet, and storage hardware prices decrease, programs that process huge amounts of data become increasingly common. In order to speed up those programs which are driven by huge input volumes, data parallelism is desired that allows to parallelize these tasks, using distributed systems of machines. MapReduce [1] is a programming model which is widely used in the context of these tasks, since it provides the programmer with a high level of abstraction and it scales extremely well.

1.1 MOTIVATION

Due to huge input sizes that comprise gigabytes or even terabytes of data, MapReduce jobs often run for hours or even days, even when utilizing hundreds of machines. As these jobs are often re-executed in the context of input that is only changing slightly over time, we felt that users would benefit from processing the newly acquired alterations only. Reduced execution time would allow for more frequent re-execution of MapReduce computations and thereby yield more up-to-date results in practical computations. Many areas of applications even use compound MapReduce jobs and they could accumulate benefits by only pushing changes through such a *pipeline* of jobs.
Since MapReduce provides an isolated view on data items, no dependencies between the processing of two data items exist. Furthermore, the operations performed by MapReduce computations are associative, typically even commutative, and hence allow the incorporation of changes to a generation of data into earlier results. As the differences between two generations of data can be expressed by their delta, i.e. in terms of additions and deletions, we propose to process only this delta in a MapReduce job, instead of the entire second generation.

1.2 RESEARCH QUESTIONS

In the context of this thesis we will cover basic questions on the usage of deltas in the context of MapReduce. More specifically:

- Which preconditions have to be present in order to use deltas in the context of MapReduce?
- Which classes of MapReduce jobs are suitable to be used in combination with deltas?
- What are the individual steps required to use deltas in the context of MapReduce?

As we are concerned with efficiency, we try to get a good impression on the overall potential of deltas in conjunction with MapReduce:

- How can deltas be used efficiently?
- What are the possible speedups in different MapReduce scenarios?
- What limitations exist in using deltas in combination with MapReduce?

1.3 CONTRIBUTIONS

The contributions of this thesis are of both theoretical and practical nature: we provide algebraic verification on the usage of deltas in the context of MapReduce jobs. Furthermore, different methods of computing deltas are discussed and compared. We take a look at the necessary augmentations in order to process deltas in MapReduce jobs, as well as subsequent steps.
In order to demonstrate the applicability of our approach, several use cases are implemented in Hadoop, the most import one dealing with web crawlers. We test some of these use cases in separate benchmarks, in order to examine the benefits and limitations of the various ways of dealing with deltas.

1.4 STRUCTURE

Chapter 2 introduces the basic concepts of *MapReduce* and *deltas*, as well as the software frameworks *Hadoop* and *Nutch* we use in our approach. Several use cases that support our motivation are described in chapter 3, before we focus on extending the MapReduce programming model to incorporate deltas in chapter 4. An implementation of our most important use case, a web crawler that uses deltas, is discussed in chapter 5. Chapter 6 defines and executes benchmarks for delta-aware MapReduce computations. Chapter 7 discusses related work, before chapter 8 concludes this thesis.

2 BACKGROUND

2.1 MAPREDUCE

MapReduce is a programming model for processing huge datasets in distributed systems of computers. Introduced by Google in 2004 [1], it has influenced the area of parallel applications that run on clusters of machines significantly since. While Google never open sourced their framework, a vast quantity of open and close source frameworks exist. One of the most widely used, Hadoop [2], is introduced in detail in section 2.3.

MapReduce relies on a programming paradigm that originates in functional programming. In functional programming languages like *Haskell*, the `map` function takes a list and a unary function as arguments, and applies that function to every element of the list separately. The following example demonstrates the application of `map` with the `abs` function which computes the absolute value of a number:

```
Input: map abs [-1,-3,4,-12]
Output: [1,3,4,12]
```

Haskell's `reduce` function also takes a list as its second argument. The first argument however is a binary (instead of a unary) function. `reduce` then applies this binary function to every element of the list, taking the previous result as the second argument. This essentially combines the entire list to a single value, as can be seen if applied to '+':

```
Input: reduce (+) [1,2,3,4]
Output: 10
```

MapReduce loosely follows this concept by using mapper functions that process input items separately, one at a time, and reducer functions that aggregate multiple items to a single value. A MapReduce framework typically operates with one master-node that is responsible for assigning tasks to worker-nodes and monitoring progress as well as possible failures. If a job is submitted to a MapReduce cluster, the master splits the (typically huge) input data into smaller parts, and it assigns them to the worker-nodes.

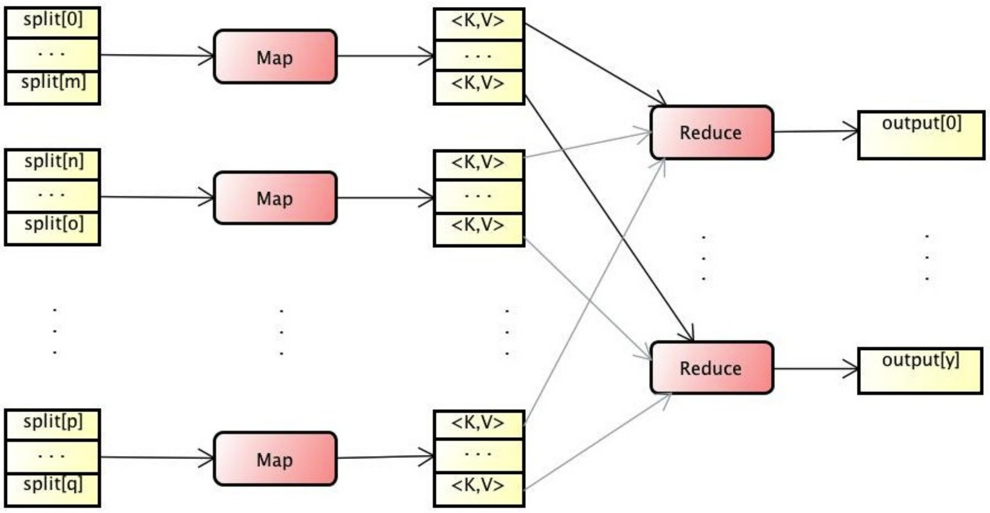

Figure 2.1: Workflow of MapReduce.

```
mapReduce :: (k1 −> v1 −> [(k2,v2)])       −− The MAP function
               −> (k2 −> [v2] −> Maybe v3)  −− The REDUCE function
               −> Map k1 v1   −− A key to input−value mapping
               −> Map k2 v3   −− A key to output−value mapping
```

Figure 2.2: Signature of MapReduce

These nodes then execute the specified map function on every single data item in their input split. After all input is mapped, the resulting intermediate data is grouped and copied to designated reducer nodes. Figure 2.1 shows how invocations of mappers and reducers are separated into stages. The second stage (the reducers) does not begin before the first stage (the mappers) has finished.

Data items in MapReduce are essentially pairs of keys and values. Key-value pairs are presented one at a time to the mappers, who then extract an arbitrary number of key-value pairs (possibly of different type than their input), and pass them along to the reducers. The decision which tuples are being passed to which reducer is based on the tuple's key, and it is made by a customizable partitioner-function. One reducer invocation then receives all values that share a specific key, and merges them together to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invokation [1]. Listing 2.2 illustrates this concept by showing the signature of MapReduce as proposed by [3].

Parallelism is enabled by the fact that mappers and reducers have no side effects, and each invocation processes a subset of all data items that is disjoint from other subsets processed at that stage. This allows for parallel execution of unlimited map or reduce instances.

2.2 DELTAS

The term *delta* is used in computer science in the context of version management. During the lifespan of an artefact (a machine readable object like source code files, text files, diagrams, etc.), the artefact can exist in different manifestations called versions. Simply adding a line to a text file would generate a new version of that file. A delta is a complete description of the differences between two versions, which from a logical point of view is the information required to obtain one value from the other.

Deltas can be used to compare versions of files, like in the context of the Unix program `diff` [4]. Deltas are also used to store one version by only enumerating all changes to another version. This can be utilized to reduce data redundancy in the context of files that typically undergo very little changes. Version management software like Apache's Subversion [5] create repositories of files using that mechanism.

Deltas can generally be distinguished into forward-deltas and backward-deltas. In the context of backward-deltas, the emergence of a new version requires to remove the latest version, and to replace it by all changes to the current version. The current version is then stored in its entirety. This enables very quick retrieval of the latest version, however storage of a new version requires some computations. Forward-deltas, on the other hand, work in the exact opposite way. An initial version is stored in its entirety, and the following versions are simply stored as changes to previous versions. This enables quick storing but slower retrieval of past versions.

2.3 HADOOP

2.3.1 Overview

Inspired by Google's MapReduce [1], Hadoop was originally created by Doug Cutting in order to support distribution for his open source web-search software Nutch [6] (see section 2.4). Hadoop is an open source Java implementation that is distributed under

Apache License 2.0 and is available in version *0.21.0* at the time of writing. Hadoop is being used by many widely known companies like *Amazon*, *Facebook*, *Yahoo*, etc. [7].

2.3.2 Architecture

Hadoop generally consists of three building blocks: *Common*, *HDFS*, and *MapReduce*. In the following, each of these will be discussed in detail.

HDFS

As MapReduce is used to distribute single tasks on clusters of machines, a distributed file system is needed for the storage of input and output data. Hadoop uses the *Hadoop Distributed File System (HDFS)* [8], which is modelled after the *Google File System (GFS)* [9]. This section describes HDFS, however as it is modelled after GFS all design aspects mentioned here apply to GFS as well.

HDFS is designed to run on commodity hardware, that is on clusters where quantity of machines is more important than quality. This leads to the effect that failure of nodes occurs frequently, hence HDFS has been designed to be highly fault-tolerant. Fault-tolerance is achieved by making detection of faults, and quick, automatic recovery a core architectural goal of HDFS. This is achieved by numerous design decisions, maybe most importantly by having a replication factor that defaults to three, but which is configurable for every file. In order to protect against failures of entire clusters (e.g. caused by power outage), HDFS tries to take locality of nodes into account when creating replicas, in order to distribute files across clusters.

As applications that use HDFS generally operate on huge data sets (file sizes are generally in the gigabyte or even terabyte area), high throughput is more critical than low latency. For this reason HDFS relaxes some POSIX semantics in key areas in order to achieve high throughput. In order to deal with the batch-jobs that read huge amounts of data, which are typical in the context of HDFS, the size of the blocks a file is split into is set to 64 MB by default (again configurable by the user). This reduces the amount of communication between clients and HDFS servers that is needed in order to retrieve file meta data and block locations.

The file system is organized into one master server, the *NameNode*, and many slave servers the *DataNodes*. The NameNode is responsible for coordination of critical processes, like naming, file replication and assignment of file blocks to DataNodes. The DataNodes on the other hand handle a set of file blocks assigned to them by the master.

In order to provide quick recovery from failures, the DataNodes frequently send so-called *heartbeat* messages to the master, in order to signal that they are alive. This allows the master server to detect when a node is down, and to assign the blocks that were managed by this node to other DataNodes, in order to maintain the required replication factor.

HDFS follows a *write-once-read-many* access model for files. This means once created, a file can never be mutated. This limitation however is acceptable because first, it fits nicely with the programming model of MapReduce where data processing jobs write their output to a set of new files instead of manipulating existing ones. In addition, this allows for a simple consistency model as updates never need to be propagated to existing blocks. In this context, there is a noteworthy difference between GFS and HDFS: According to the original GFS paper [9], GFS supports an append operation to extend existing files. HDFS was originally created without support for an append operation. However, while append was introduced in version 0.19.0, it still is not fully supported due to existing problems [10].

Having a single master server can quickly become a bottleneck. To avoid this, client data never flows through the master. For instance in the context of reading, a client only contacts the master to learn about current locations of file blocks. After the client received the location information it contacts any of these DataNodes directly, usually the closest one, in order to read data. In the context of writing, this workflow becomes slightly more advanced. As most clients perform streaming writes, it becomes impractical for the client to directly write to a remote file as throughput will be limited significantly by network latency. Instead, the client transparently caches the file data in a local temporary file. Once the file has grown to the size of a HDFS block, the client contacts the NameNode which inserts the file name into the namespace and answers with a list of DataNodes the block should be replicated to. Replication to DataNodes is then done in pipelined fashion, starting with the client copying the block to the first DataNode on the list. This node then copies the block to the second DataNode on the list, and so on. For a more detailed description of HDFS's design and additional features like snapshots and logs, please refer to [8].

Common

Hadoop Common provides RPCs for inter-node communication, as well as means to access the underlying file system. Please note that when running in non-distributed mode (i.e. on a single machine), Hadoop directly interacts with the local file system, skipping HDFS. The task to delegate and translate calls to the file system API to the appropriate file system is the responsibility of Hadoop Common. Hadoop Common also provides the most basic file type available in Hadoop: `SequenceFiles`. As we have seen above, files in Hadoop do not support random writes. Hence, any file format in Hadoop can only be written to by calling:

```
append(K key, T value);
```

The types of keys and values are wrapper classes for data types that implement a `Writable` interface, to take care of serialization and deserialization. This ensures that `SequenceFiles` know how to de-/serialize them.

MapReduce

The MapReduce engine consists of one *JobTracker* and many *TaskTrackers*, much like *NameNode* and *DataNodes* in HDFS. The JobTracker assigns jobs to available TaskTrackers, monitors progress and acts in the case of failures. One important concept is awareness of data locality during task scheduling. The JobTracker always tries to allocate tasks to nodes that already hold the necessary input data, avoiding expensive network transfer of data. In order to set up a MapReduce job, the user basically has to specify implementations of the following concepts:

- InputFormat
- Mapper
- Combiner
- Partitioner
- Reducer
- OutputFormat

The overall coherence of these concepts is visualised in figure 2.3. The `InputFormat` basically takes care of splitting the input data into appropriate chunks as input for each map task, and provides a `RecordReader` that converts bytes to keys and values. A `Mapper` provides a map-function that processes key-value pairs as explained in section 2.1. All the necessary setup and clean-up is also done here. By default (i.e. if the user does

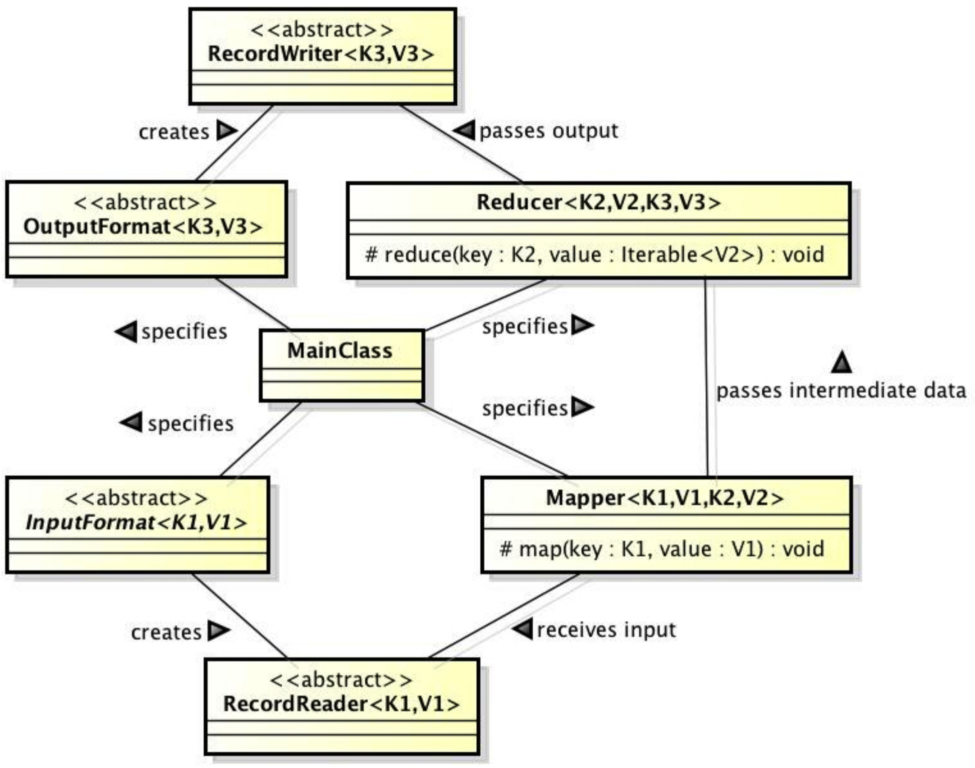

Figure 2.3: UML class diagram of the structure of a MapReduce job in Hadoop.

not determine a specific mapper) a mapper implementation is used, that simply copies each pair. In order to reduce the amount of pairs transferred over the network to the appropriate reducer nodes, the user can optionally specify a `Combiner`. Combiners typically behave exactly like reducers, hence *crunching* the data prior to sending it to the reducers. The assignment of keys to reducers is job of the `Partitioner`. The user can control this assignment by implementing a custom Partitioner. By default, partitioning is done by computing a hash-value of the key, and computing the *modulo* of the number of reducer nodes in order to obtain a valid reducer-ID. Similar to mappers, `Reducers` are also specified by means of an appropriate reduce function along with all the necessary setup and clean-up work. As the reducer processes all values that share the same key simultaneously, they are made accessible by some iterable structure. Again, the default implementation simply copies each pair, essentially not applying any reduction to the set of pairs. Finally, the `OutputFormat` is responsible for serialization of key-value pairs to disk. Typically, each reducer writes to its own file in the output directory. This avoids race-conditions and facilitates splitting of the input for subsequent MapReduce jobs.

```java
Configuration conf = new Configuration();
Job job = new Job(conf, "word count");

job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));
job.setInputFormatClass(TextInputFormat.class);
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setOutputFormatClass(SequenceFileOutputFormat.class);

job.waitForCompletion(true);
```

Listing 2.1: Setting up a WordCount job in Hadoop

2.3.3 Code Examples

Originally introduced in Google's MapReduce paper [1], *WordCount* is possibly the most commonly known example for MapReduce. The task is to simply count all occurrences of all words present in a document or a set of documents. This is achieved by processing each line of text of each input document separately in the map-function. For each occurring word a tuple is written that contains the word as the key, and the number '1' as the value. Reduction is done by simply summing up values (i.e. occurrences) for identical words. In Hadoop a WordCount job would be created as demonstrated in listing 2.1.

The first two lines simply create a job and a configuration for that job. Users can use the configuration of a job to specify certain of it's properties like the number of reducers. A job's configuration can also be used to submit information globally to every job via context. We will see how this is done in a second, when talking about the mapper implementation. After setting mapper, combiner and reducer, the types of output keys and values are specified in order to let the framework create appropriate output files. This is not necessary for the input as this information is directly encoded at the beginning of each SequenceFile. `IntWritable` and `Text` are wrappers for `Integer`

```
1  public static class TokenizerMapper
2        extends Mapper<Object, Text, Text, IntWritable>{
3
4     private final static IntWritable one = new IntWritable(1);
5     private Text word = new Text();
6
7     public void map(Object key, Text value, Context context
8                     ) throws IOException, InterruptedException {
9       StringTokenizer itr = new StringTokenizer(value.toString());
10      while (itr.hasMoreTokens()) {
11        word.set(itr.nextToken());
12        context.write(word, one);
13      }
14    }
15  }
```

Listing 2.2: The mapper used for WordCount

and `String` respectively, that implement the required `Writable` interface. Finally, before the job is submitted to the framework, input and output paths are specified along with the corresponding file formats.

The `TokenizeMapper` that was chosen in listing 2.1 can be seen in listing 2.2. Every mapper class has to extend Hadoop's `Mapper` along with the appropriate types of the input and output key-value pairs. The aforementioned context is available as a third parameter to the map-function. It provides access to meta-data, like information about the currently processed input or data specified by the user at the time of job creation. It also delegates write calls to the appropriate writer (see line 12).

Listing 2.3 presents the previously chosen `IntSumReducer`. For each encountered word, one call to the reducer-function is made, along with all occurrences that were extracted in the mappers. They are accessible via the `Iterable` passed as value. Note that in the final reduction some of these values may have been pre-reduced by the per mapper call to the combiner.

```java
public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

Listing 2.3: The reducer used for WordCount

2.3.4 Features

Hadoop offers multiple features that extend the original MapReduce model. Two of these, which are specifically useful in the context of deltas, are discussed in the following.

Map-side join

In traditional MapReduce all tuples are sorted by key per reducer before being presented to the reduce function. If the input data is already sorted, sorting and transferring of the data between mappers and reducers might be unnecessary overhead. *Map-side joins* provide sequential reading from multiple sorted inputs, and make all values that share a specific key accessible in a single map call. Although individual tasks may lose much of the advantage of data-locality mentioned in section 2.1, the overall job benefits from avoiding the entire reduce phase along with the cost of transferring intermediate data between mapper and reducer nodes. Similar to databases, several types of joins are supported (inner, outer, etc.), and users can even implement their own join (for instructions consolidate appropriate literature about Hadoop, like [11]). In order to use map-side joins, the different inputs need to be aligned carefully. All inputs need to be sorted and partitioned in the exact same fashion (again, for detailed constraints

```
 1 Configuration conf = new Configuration();
 2 Job job = new Job(conf, "word count merge");
 3
 4 job.setJarByClass(WordCountMerge.class);
 5 job.setMapperClass(JoinMapper.class);
 6 job.setNoReduceTasks(0);
 7
 8 job.setOutputKeyClass(Text.class);
 9 job.setOutputValueClass(IntWritable.class);
10
11 Path[] input = new Path[3];
12 input[0] = new Path(args[0]);
13 input[1] = new Path(args[1]);
14 input[2] = new Path(args[2]);
15
16 job.setInputFormatClass(CompositeInputFormat.class);
17 job.getConfiguration().set(
18                 CompositeInputFormat.JOIN_EXPR,
19                 CompositeInputFormat.compose("outer",
                       SequenceFileInputFormat.class,
20                          input));
21
22 FileOutputFormat.setOutputPath(job, new Path(args[3]));
23 job.setOutputFormatClass(SequenceFileOutputFormat.class);
24
25 job.waitForCompletion(true);
```

Listing 2.4: Setting up a map-side join in Hadoop

refer to [11]). For a simple example of map-side joins we refer again to the WordCount example. Let us assume that we have independent outputs of three previous WordCount jobs, and now want to merge them into one accumulated result. If the jobs were run with the same configuration (i.e. same partitioner, same number of reducers, etc.) they should be aligned appropriately for map-side joins. Listing 2.4 demonstrates how the job is set up.

Note that all summation will already be done in the mapper, hence no reduction will be needed and the number of reducers can be set to zero (see line 6). All input paths that are to be processed need to be combined into one `Path` array. Further, the `InputFormat` needs to be specified as `CompositeInputFormat` (line 16). However, a custom input

```java
public static class JoinMapper
        extends Mapper<Text, TupleWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void map(Text key, TupleWritable values, Context context)
                                throws InterruptedException,
                                    IOException {
        int sum = 0;
        for (Writable val : values) {
            IntWritable partResult = (IntWritable) val;
            sum += partResult.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

Listing 2.5: An example mapper used in a map-side join

format to extract tuples from the input files can be specified during join-definition (lines 17-20). The type of the join is specified as the first argument to:

```java
CompositeInputFormat.compose(String joinType, Class<? extends InputFormat>
    inputFormat, Path[] input)}.
```

In the context of map-side joins, values always have the predefined type `TupleWritable` (see listing 2.5). It can be thought of a kind of list, where every entry corresponds to one of the input files. Since not every input file necessarily contains an entry for a given key, some list-entries might be empty. This is solved in `TupleWritables` by offering an iterator that only returns list-entries whose value was set. In our example, this allows for implementation of a mapper that is almost identical to the reducer of the original WordCount implementation in listing 2.3.

MultipleInputs

Hadoop is generally designed to allow multiple input paths per job. As seen in the context of the WordCount example, each input path is simply added to a job using:

```
FileInputFormat.addInputPath(Job job, Path p);
```

Note that only a single type of InputFormat can be specified for all inputs using `job.setInputFormatClass(...)`. In the context of heterogeneous inputs however, we might need to be able to extract keys and values differently from each input. This is not possible using the static way input paths are normally added. For these scenarios, Hadoop provides the `MultipleInputs` class. It allows to specify a separate `InputFormat` for each input path. Optionally, the user can also specify separate mapper classes for each job, in order to process heterogeneous inputs differently:

```
MultipleInputFormats.addInputPath(Job job, Path p, Class<? extends
    InputFormat> inputFormatClass);
MultipleInputFormats.addInputPath(Job job, Path p, Class<? extends
    InputFormat> inputFormatClass, Class<? extends Mapper> mapperClass);
```

2.4 NUTCH

2.4.1 Overview

Browsing the web without the support of web-search engines is unthinkable today. Search engines like *Google, Yahoo!* or *Bing* provide means to quickly find a specific piece of information in the web. One important factor for the responsiveness of today's search engines is the fact that searching of the web is not done at the time of a query, but beforehand. Most search engines use so called *web crawlers* to traverse and store huge parts of the web. These crawlers move around by downloading a site's content and following links to other sites. After the crawlers have finished, the content of a site is analysed and prepared for searching. Queries are later executed on the previously accumulated data.

Nutch is an open source web-search software that is distributed under *Apache License 2.0*. It is available in version 1.2 at the time of writing, however version 1.3 is about to be released. Apache's core functionality is to crawl the web, while index creation and search-functionalities are delegated to *Apache Lucene*. Nutch is implemented entirely in

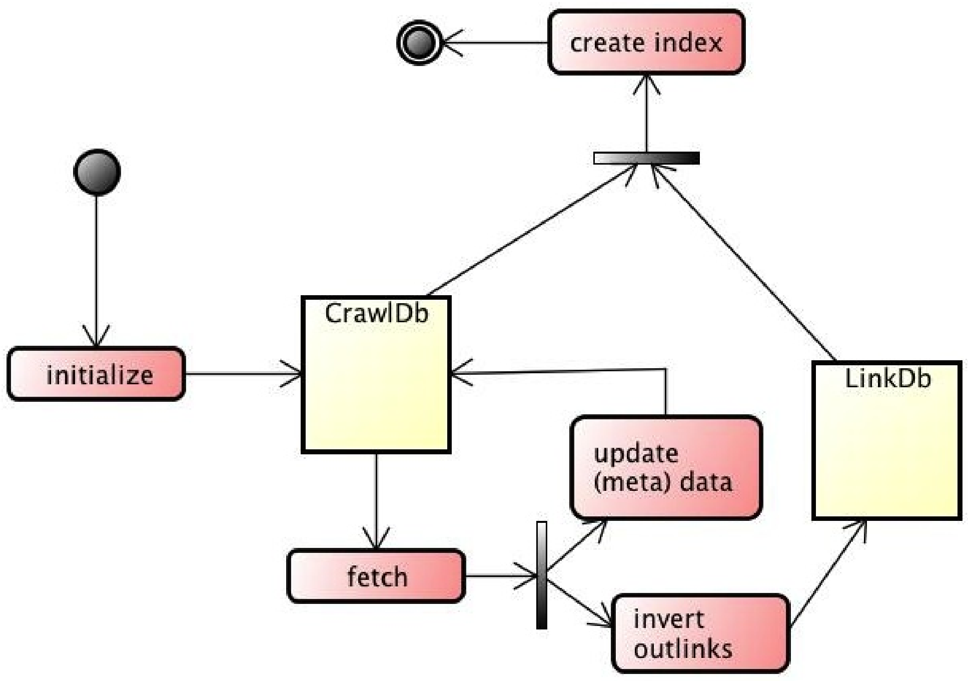

Figure 2.4: Workflow in a simple web crawler

Java and provides a plug-in mechanism that enables users to customize processes like parsing, data retrieval, querying, clustering, etc. In the following, we will focus on the description of the architecture of Nutch's web crawler. As Lucene is a fairly complex project on its own, we omit details and refer to the project's website [12] for further information.

2.4.2 Architecture

Nutch provides two basic ways to crawl the web: first, the user can use a crawl command that will perform all necessary steps. Alternatively, the steps performed by the crawl-command can be invoked separately by the user, enabling a more sophisticated configuration of the crawl. Also recrawling previously crawled sets of websites requires step-per-step invocation of the necessary commands. In order to describe how Nutch is designed, figure 2.4 shows the steps performed in a simple web crawler. The figure demonstrates how crawls are typically performed in cycles (`CrawlDb` → `fetch` → `update (meta) data`). Each cycle a set of URLs is fetched, and newly discovered links are added to the fetch-frontier.

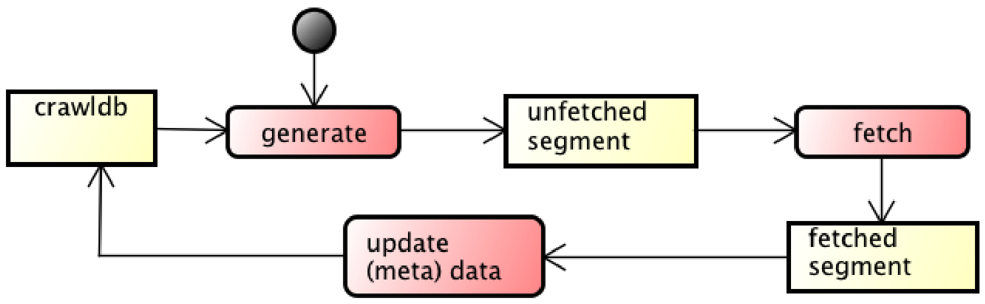

Figure 2.5: The crawl-cycle performed by Nutch

In the beginning `CrawlDb`, a repository that maintains all necessary meta data about encountered websites is initialized (`initialize`) with a set of seed-URLs. These URLs are the starting point of our web crawl. In order to quickly reach many different sites, the seeds should be chosen carefully.

As mentioned above, fetching is done in cycles in order to discover a large number of different URLs. Figure 2.5 shows the fetch cycle in detail. The `generate` step extracts a fetch-list (`unfetched segment`) from `CrawlDb` that contains all URLs that should be fetched this round. After downloading the content of all chosen websites (`fetch`), the *CrawlDb* is updated (`update (meta) data`) with relevant information (e.g. whether the site could be reached, whether it was modified, discovered links, etc.). During fetching, the content is also used to extract websites the downloaded sites are linked to. Fetched sites and links are used to create a collection of inverted links (`invert links` — again in figure 2.4), that is an entry for each discovered site that contains all sites that point to it. This *link-graph* is stored in the `LinkDb`. Finally, the data that was accumulated in the `CrawlDb` can be used along with the `LinkDb` to create an index (`create index`). This index is later being queried by the web-search interface.

2.4.3 Code Examples

The individual tasks performed by Nutch are each implemented as one or multiple MapReduce jobs, in order to provide parallelization and fault tolerance. We illustrate this by demonstrating the concept of link-inversion (`invert links`) using MapReduce. In order to traverse the web, fetched websites are parsed for outlinks a priori to the link-inversion step. Websites are then processed by the mapper, along with the set of URLs they are pointing to. The inversion is done by writing a tuple for each target URL, together with the source URL as the value. The reducer can then simply collect all *sources* that point to a specific target URL. Listing 5.2 shows pseudo code for the

```
map(URL key, List<URL> values) {
    for(URL target : values){
        List<URL> sources = new List<URL>();
        sources.append(key);        //use lists for combiner
        emit(target, sources);
    }
}

reduce(URL target, Iterable<List<URL>> values) {
    List<URL> inlinks = new List();
    for(List<URL> sources : values){
        inlinks.append(sources);
    }
    emit(target, inlinks);
}
```

Listing 2.6: A pseudo code implementation of Nutch's link-inversion step.

involved mapper and reducer. Please note that in order to enable local combining at the mapper node, lists are used as intermediate values.

3 USE CASES

This section introduces some simple use cases that demonstrate the benefits of using deltas in the context of MapReduce computations. Here, we focus on modifications to existing MapReduce jobs in order to process deltas in a way that enables later merging with previous results. Algebraic verification and details on how to compute a delta in the context of MapReduce are delayed until chapter 4.

3.1 WORDCOUNT

In the context of Hadoop we have introduced the WordCount example in section 2.3. Recall that documents are mapped by simply extracting a tuple for every word in the document that contains the word as the key, and the number 1 as the value. It is easy to realize that if a document has changed in a way that some text has been added to a previously existing document, these changes can be applied to the overall result of WordCount by simply producing a tuple for each added word in the conventional fashion. As the original WordCount reducer simply sums up values for tuples with the same key, changes can be introduced into the old result by simply reducing the old result along with the newly created tuples. However, changes to a document are usually not only done by append, text can also be deleted from the original document. Fortunately, deletions are as straightforward as additions: in order to reflect in the final result that some of the words are not present in the document any more, their occurrence has to be subtracted from the current count. This can be achieved by producing tuples in the mapper for every deleted word that contain the word as the key and -1 as the value. Using regular reduction, this ensures that the previously added occurrence of this word is now being removed again.

In order to demonstrate this approach in the context of our WordCount implementation, we need some kind of delta. Since we delayed the discussion on efficient delta creation in the context of MapReduce until chapter 4, we use the *UNIX* tool `diff` for this purpose. Listing 3.1 shows an example output of `diff`:

```
1c1
< MapReduce is a programming model and an associ-
---
> Map-Reduce is a programming model and an associ-

21c21
< sand MapReduce jobs are executed on Google's clusters
---
> sand Map-Reduce jobs are executed on Google's clusters
```

Figure 3.1: Example output of `diff`

For our purpose, only lines actually containing content are relevant: Lines starting with '<' mark content that is present in the first version of the document, but not in the second one (i.e. deleted text). Added text is marked by '>'. As additions and deletions could also be marked by '+' and '−', we will also refer to the *sign* of the line in this context. In order to ignore unimportant lines and to distinguish in the mapper whether a line contains added or deleted content, we have two possibilities: First, we could implement a custom `InputFormat` that skips lines not starting with '<' or '>', and that passes the *sign* of the line into to the mapper. Including the sign into a key-value pair can be done by using a `Pair`-type for the value, that contains both the line content and the sign (e.g. as a `boolean`).
The second possibility is to implement this behaviour directly in the mapper. For simplicity, we only show the second approach in listing 3.1, in order to avoid a detailed introduction of `InputFormats`. Reduction is simply done using the same reducer as the original WordCount implementation in listing 2.3.

3.2 MATRIX MULTIPLICATION

As seen in the previous section, using deltas in the context of simple mathematical operations is relatively straightforward. This section tries to illustrate this point using a slightly more complex example: the multiplication of two matrices. This is an important use case as matrix multiplication causes non-trivial computational cost, and is applied in many different areas of computer science.
Recall that the product of two matrices is calculated as demonstrated in listing 3.2.

```java
public static class DeltaTokenizerMapper extends Mapper<Object, Text,
    Text, IntWritable>{
    public static final String POS_STRING = ">";
    public static final String NEG_STRING = "<";

    private final static IntWritable count = new IntWritable();
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws
        IOException, InterruptedException {
        int sign;
        String line = value.toString();
        if(line.startsWith(POS_STRING)){
            sign = 1;
        }else if(line.startsWith(NEG_STRING)){
            sign = -1;
        }else{
            return;
        }
        count.set(sign);
        StringTokenizer itr = new StringTokenizer(line.substring(2));
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, count);
        }
    }
}
```

Listing 3.1: Mapper used to process `diff` output in the context of WordCount

$$A * B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} * \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} = C$$

Figure 3.2: Example matrix multiplication

Assuming that one or both of the input matrices have changed, we show that it is sufficient to process a subset of the matrices in order to obtain a result that can be used to update the original product to obtain the new product. The difference between two versions of a matrix can be determined by subtracting one from the other (assuming that both versions still have the same number of dimensions). The following illustrates the case that element $a_{0,0}$ of matrix A in our example multiplication changed from 1 to 0 (modifications to B can be handled similarly, by substituting 'row' by 'column' in the following):

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, A' = \begin{pmatrix} 0 & 2 \\ 3 & 4 \end{pmatrix}, A_{delta} = A' - A = \begin{pmatrix} 0 & 2 \\ 3 & 4 \end{pmatrix} - \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix}$$

We will show now (without proof) how this difference (*delta*) can be used to update the result C of the multiplication of A and B, to C' (the result of $A' * B$). Each element of row i in matrix A is only used to calculate row i of C. Similarly, each element of column j in B is only used to calculate column j of C. This shows that there is no need to recalculate the entire multiplication, but only rows of the result that contain changed elements from matrix A (or columns that contain changed elements from matrix B). After multiplying the modified row of matrix A by every column of matrix B, updating of the resulting matrix can be done by simply adding this product to the original result:

$$A_{delta} * B = \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix} * \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} -5 & -6 \\ 0 & 0 \end{pmatrix} = C_{delta}$$

$$C' = C + C_{delta} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} + \begin{pmatrix} -5 & -6 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 14 & 16 \\ 43 & 50 \end{pmatrix} = A' * B$$

It may seem that this procedure introduces more overhead than providing benefit. However, two aspects need to be taken into account: First, in order to calculate C_{delta} no full matrix multiplication is necessary. Instead, it suffices to only multiply the rows

of A_{delta} that do not contain solely 0 values (i.e. rows with changes) by matrix B. The second aspect that needs to be taken into consideration is an application where two large matrices are being multiplied again and again, and only few values in one of the two matrices change. For example [13] introduces *SA-Clustering*, a clustering algorithm for attributed graphs that iteratively recalculates a *random walk distance matrix* based on changing weights, using matrix multiplication. Zhou et. al. [14] already showed, that only parts of this matrix change in each iteration. They proposed an incremental algorithm to incrementally update the random walk distance matrix. Their approach to exploit incrementality to avoid full matrix multiplication is quite similar to the approach presented here.
It can easily be seen that incremental matrix multiplication reduces complexity from $O(n^3)$ to $O(n^2)$. For example, if multiplying two $1000x1000$ matrices by each other, the following overheads occur if one row/column changes in one of the two matrices:

- Regular Multiplication: 1000^3 single element multiplications (as each element of the resulting matrix requires 1000 factors being added to each other)
- Delta Multiplication: 1000^2 single element multiplications (as only one row/column of the resulting matrix is being recalculated)

A MapReduce implementation is omitted here, as existing MapReduce implementations of matrix multiplication are quite complex. They split the input matrices into blocks small enough, so that a pair of blocks can be multiplied in memory on a single node. Block multiplication is done at at the reducers who receive the right data via a complex intermediate key structure and partitioning function. A second MapReduce job is then used to unite the single blocks into the final result. A detailed description of an implementation can be found under [15].

3.3 WEB CRAWLER

3.3.1 Motivation

While section 2.4 introduced web crawlers in the context of Nutch, this section will focus on potential improvements in performance by using them in combination with deltas. Recall that web crawlers are used to create an index for a search engine. The web is constantly changing as sites are being added, deleted or modified, hence actions need

to be taken in order to keep the index up to date. Most search engines (like Nutch) deal with this problem by periodically recrawling the whole web, and recomputing the index from scratch. It is impossible to state a specific interval that sites are commonly being recrawled in, as sophisticated crawlers vary that interval between sites. Crawlers can learn an optimal recrawl policy over time, by observing importance (determined e.g. by PageRank [16]) and change frequency of each site. However it can be assumed that a commercial search engine is constantly crawling websites in order to keep its index up to date. This claim is supported in [17], by leaking the information that Google crawls several billion documents each day.
Constant recrawling leads to huge amounts of input data for index creation. For instance [1] mentions that index creation at Google processes about 20TB of input data. One could assume that index creation has become much more complex since 2004, hence the input volume for indexing might have increased significantly. This is supported by the fact that [17] claims that indexes at Google store tens of petabytes of data. The literature mentioned above further states that index creation was done, prior to their latest switch of concept, by feeding newly crawled documents along with a repository of existing documents through a series of 100 MapReduce jobs. They claim that this forced each document to be indexed for 2-3 days before it could be returned as a search result. The huge amount of data along with the number of MapReduce jobs used in order to create an index suggest significant gains using deltas in the context of web crawlers. The rest of this section will outline possible changes to an existing web crawler in order to use deltas.

3.3.2 Modifications to crawling

Based on figure 2.4, which we used to illustrate Nutch's architecture, figure 3.3 hints at possible modifications of a web crawler in order to use it in combination with deltas. Workflows that are modified in order to incorporate deltas are marked blue. This section introduces possible modifications, while details are delayed until chapter 5, which describes the implementation of a crawler that uses deltas.
The `CrawlDb` is initialized as before, and an initial crawl discovers and downloads websites. In subsequent recrawls that aim at updating the possibly deprecated index, we can now focus on changes. After fetching a chosen set of URLs, the content of these sites is being compared to their previously encountered versions, in order to determine which sites have changed. In the following, only these changed sites are used to create

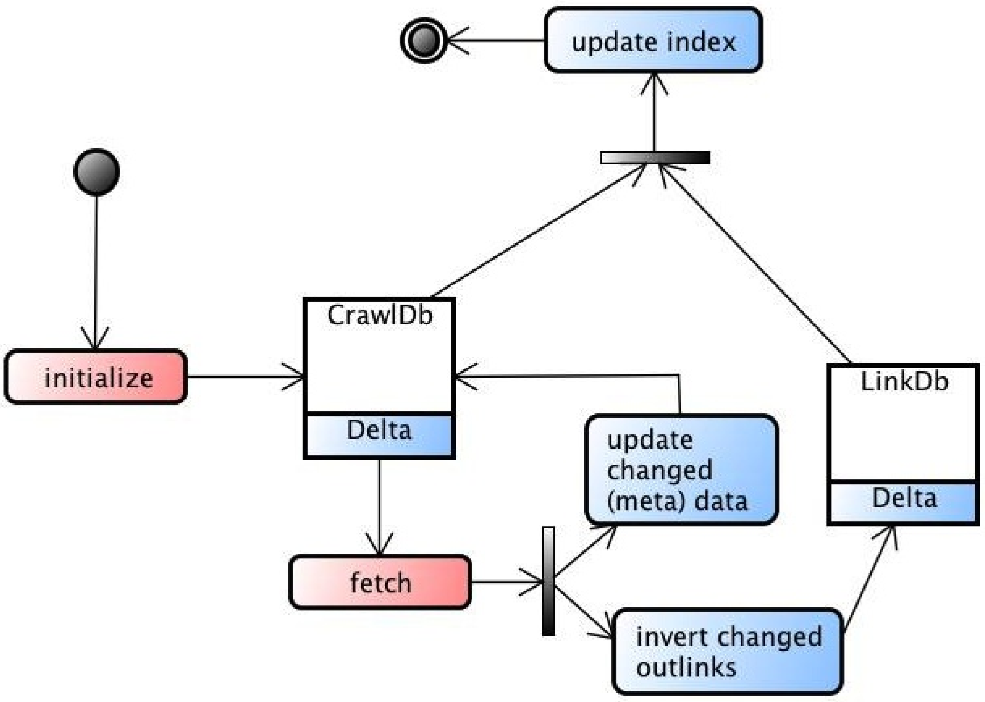

Figure 3.3: A web crawler optimized for deltas

a delta of the content, and to update the CrawlDb. Further, a delta of the discovered outlinks of these sites can be computed (i.e. only added/deleted/modified links). This delta is then processed by the `invert changed outlinks` job, and used to create a `Delta` of the `LinkDb`. We now have a delta of all added and deleted inlinks, as well as deltas of the websites' contents. These deltas can be used in the following to update an existing index, avoiding costly recomputation of the entire index.

3.4 INDEXER

Let us assume that our simple index is created by the execution of three individual tasks that are characteristic for index creation: *PageRank* computation, creation of an *inverted index*, and *clustering* of websites. In the following we will take an close-up look at each of these tasks.

3.4.1 PageRank

PageRank, as introduced by [16], is used to determine the 'importance' of a website, in order to contribute to a ranking of the search results that are presented to the user. The algorithm used for PageRank is based on the idea to assign a score to each web page that represents the likelihood that a person who randomly clicks on links will arrive at any particular page. The idea behind this concept is that the more important a site is, the more sites will point to it. Sites that have other important sites pointing to them are even more important than sites that are being pointed to by unimportant sites. The PageRank of crawled sites is calculated based on a weighted web graph, where nodes represent websites and edges represent outlinks. More specifically: an outlink from website A to website B results in an edge from node A to node B in the resulting graph. Initial weights are assigned to each site, which are refined by iteratively passing a score based on a node's current weight to all nodes it is pointing to. Incoming scores to a node are used to update its PageRank, and in the next iteration this updated score is passed to subsequent nodes.
Iterative algorithms in MapReduce have been a field of intensive research. Related work (chapter 7) introduces several approaches in this area that try to improve iterative MapReduce jobs. However, one publication that is especially suitable in our context is [18]. The proposed approach to incrementally update existing PageRank calculations based on a set of changed websites is highly useful in the context of deltas. With deltas, we already have the desired information at hand about which sites have actually changed, and hence only these sites need to be considered in an update of the PageRank graph.

3.4.2 Inverted Index

Inverted indices are often used in the context of search engines, in order to provide quick access to the information which sites contain a specific word. The inverted index stores for each word a list of documents which contain that word. This provides direct access to find the documents associated with a specific word, avoiding the need to search each individual site. In order to enable a more sophisticated ranking of sites, it is desirable to not only provide the information which sites contain a specific word, but also the number of occurrences for each site. This allows the search engine to return sites that contain many occurrences of a queried term to the user first, assuming that these are more relevant to the search query than sites having few occurrences.

At the end of crawling, our CrawlDb basically contains mappings from URLs to words, since the content of a website is being identified by the corresponding URL. The inverted index now requires two steps: first, inversion of the URL-word mapping in order to access URLs from words their content contains. Second, detection of the count of each word for every single website. Step one is similar to link inversion as seen in section 2.4.3. In the context of deltas, only words that were added or deleted are being processed in the inversion step. Step two requires an implementation similar to our WordCount example. Therefore, additions and deletions can be handled in the same way (i.e. by emitting $1/-1$). Details on the creation of an inverted index via MapReduce are given in section 5.3.

3.4.3 Clustering

Introduction to clustering

Clustering (or cluster analysis) is the process of grouping elements into subsets, so that elements in the same set are similar in some sense. Clustering is a common technique for static data analysis, meaning that no initial training of the system that computes the clusters is required. In order to determine the similarity of two elements, clustering requires some kind of distance measure. The choice of the distance function will influence the final clustering, as different distance functions might lead to different clusterings. Commonly used distance functions are for example the *Euclidean distance* or the *Manhattan distance*. In order to apply the chosen distance function to them, elements need to be represented as points in an n-dimensional space. This is done by extraction of features from the elements, and assigning numerical values for each feature to the elements that are to be clustered.
Different algorithms for clustering exist, one of the most commonly used probably being *k-means*. The k-means algorithm assigns each point (of an n-dimensional space) to the cluster whose center (called *centroid*) is the closest. The center of a cluster is the average of all points in the cluster. The algorithm to compute k-means performs the following steps:

- Choose the number of clusters k
- Randomly generate k cluster centres (centroids)
- Assign each element's point to the nearest cluster (determined by the distance function)

- Recompute the new centroids
- Iteratively repeat the last two steps until some convergence criterion is met

Clustering in the context of indexing

In the context of indexing, websites can be clustered by using term-frequency vectors that are calculated based on their content. For instance, the result of a WordCount computation can be used in order to create such a vector. After careful clustering of the resulting vectors, websites with similar content should be assigned to the same clusters. This enables the search engine to move some sites up in the hierarchy of the search results, based on their similarity to top-ranking results. It is further possible to return websites to a query, that do not actually contain one of the queried terms, but are located in the same cluster as one of the top-ranking results.
A different aspect for the usage of clusters is the grouping of sites by their content in the context of ambiguous topic names. Imagine a user querying for *"jaguar"*: he could either be interested in the car, or in the animal. A sophisticated index should be able to distinguish websites based on their actual content in order to return content related to the correct topic.

Mahout

As Nutch relies on Lucene for indexing, any clustering is also handled there. While Lucene used to implement individual machine learning algorithms, the entire field was outsourced at some point to a separate Apache project: *Mahout* [19]. Mahout is distributed under *Apache License 2.0* and is available in version 0.4 at the time of writing. The initial idea, to use a parallel programming method in the context of computationally expensive machine learning algorithms, was first introduced in [20] using MapReduce. This later lead to the foundation of Mahout, a machine learning library implemented on top of Hadoop.

Clustering and deltas

As recomputation of clusters is costly due to the necessity of multiple iterations, it is usually being avoided during recrawling. Instead all crawled documents are being reassigned to existing clusters, as their content and hence their position in the vector space might have changed. For high crawl-rates (i.e. a large part of the CrawlDb is being recrawled at once) this task becomes very complex, as every site's vector needs to be compared to every centroid in order to find the nearest one. While the exact number of clusters to be used is subject to analysis based on the data set, [17] describes a benchmark that results in 3.3 documents per cluster on average. Even though actual numbers will probably differ, this still gives us a general idea about the number of centroids (i.e. size of the web divided by some small number) each vector needs to be compared to. In the presence of deltas, the amount of vectors to compare is being reduced significantly, as only documents with changed content need to be re-compared to the existing centroids. One could imagine an even more sophisticated approach that calculates the exact delta of the content (as we did in the delta version of our WordCount example). This delta could then be used as a *displacement vector*, to implement an algorithm that is aware of the distances of the clusters to each other, and can hence trigger reassignment automatically if a documents vector is relocated to a point that is part of a different cluster. As this approach would require modifications inside the Mahout project, we delegate an implementation to future work.

An additional advantage of using deltas is that they provide a good indication when enough changes have been accumulated, so that recomputing of the clusters becomes necessary. Traditional recrawling techniques do not allow such observations, and will probably rely on heuristics for this decision.

4 MAPREDUCE WITH DELTAS

4.1 ALGEBRAIC APPROACH

This section [1] is concerned with algebraic verification of the usage of deltas in the context of MapReduce computations. We try to give a formal definition of deltas, that is further used to express correctness conditions for the incremental computations, to yield the same result as the non-incremental one.

4.1.1 Formal definition

MapReduce processes pairs of keys and values, therefore its input is basically 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 to removed pairs; parts three and four represent modified key-value pairs were part three reflects the state before the second generation was introduced, and part four the state after this. Since modifications can be expressed by deletion of the first version and addition of the second version of a tuple, $\Delta_{i,i'}$ can be simplified to consist of only two collections:

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

1 This section was taken from [21], with some modifications

This leaves us with the observation that MapReduce computations can be applied to these two parts of the delta, and can later be merged with the original result. We assume, that this yields the same result as processing the entire second version independently. However, correctness conditions are needed for the non-incremental and incremental execution to agree on the result. In order to state such a criterion, we first need to formalize common MapReduce computations.

4.1.2 MapReduce and Monoids

Recall that a *monoid* is an algebraic structure with a single associative binary operation and an identity element. For example, the natural numbers $\mathbb{N}$ form a monoid under addition with identity element zero. In classic MapReduce, the mapper is not constrained, but the reducer is required to be (the iterated application of) an associative operation [1]. Recent research argued that reduction is in fact *monoidal* in known applications of MapReduce [3], [22]. 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 [3].
We try to illustrate this point using some simple examples. A detailed analysis of common MapReduce computations on the basis of monoids can be found in [22].

Distributed Grep

Being part of Hadoop's example library, *Distributed Grep* is the parallel implementation of the UNIX tool `grep`. The mapper processes input documents line by line and emits a line if it matches a supplied pattern. Reduction is done via an identity function that simply copies each intermediate tuple to the output. To determine the used monoid, we notice that the collection of key-value pairs is basically a set of dictionary type (*Map*), that is keys map to values. As the reducer simply copies the provided tuples to the output, essentially appending each to the already collected set, we can express it using the following monoid: *(Map, unionWith(++), empty Map)*, where starting with the empty *Map*, reduction iteratively concatenates (*unionWith(++)*) an input tuple (represented as a Map with a single element) to the existing preliminary result.

Reverse Web-Link Graph

As demonstrated in section 2.4, a reverse web-link graph can be constructed using a mapper that processes an URL along with all outlinks contained by that website's content as the corresponding value (e.g. as a list: `<source,[target]>`). For each URL present in the value, a new tuple is emitted that contains the target URL as the key, and a collection containing the currently processed website's URL as the value: `<target,source>`. The reducer then receives all intermediate tuples sharing the same key and concatenates their values. As values of intermediate data are lists of URLs, the corresponding monoid operates on a dictionary that maps URLs to lists of URLs. Reduction then simply concatenates multiple lists for a given URL: (*Map<List>, unionWith(++), empty Map*).

4.1.3 Deltas and Abelian Groups

In the context of deltas, we need to extend this model in order to cope with deleted tuples. In the context of our WordCount use case, deletion was handled by emitting the negated count of the previous occurrence (i.e. `<word, -1>`). This implies an extension of the monoidal model for reduction to an *Abelian group*, i.e. a monoid with commutativity for "$\bullet$" and an operation "$\bar{\cdot}$ " for an inverse element such that $x \bullet \bar{x} = u$ for all x. Hence, we assume that MapReduce computations are described by two ingredients:

- A mapper function.
- An Abelian group—as a proxy for the reducer function.

4.1.4 Formal correctness

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:

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

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 $\bar{\cdot}$ on values
Algorithm:
  for each $\langle k, v \rangle$ in $s$ do
    yield $\langle k, \overline{v} \rangle$; // value-by-value inversion
```

Figure 4.1: Lifted inversion

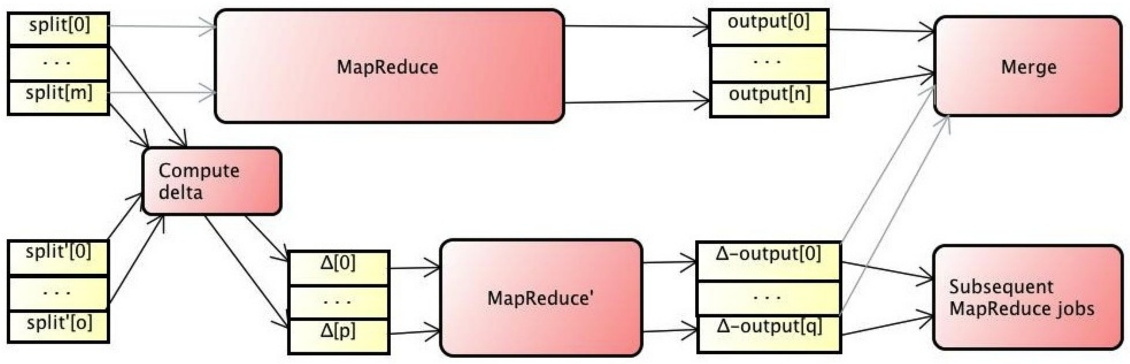

Figure 4.2: MapReduce with deltas

4.2 STEPS IN USING DELTAS

4.2.1 Delta Computation

This section is concerned with the the individual steps necessary in order to enhance a MapReduce computation with deltas. Figure 4.2 demonstrates the associated workflows. The upper part of the figure corresponds to the regular MapReduce job responsible for initial processing of the original version of the input data (represented as splits $0 - m$). Next, the second version of the input (represented as `split'[0]` — `split'[o]`) is compared to the original input, and their delta (Δ[0] — Δ[p]) is being computed (`Compute delta`). This delta can then be processed using a MapReduce computation `MapReduce'`, that is augmented as described in the previous section. The resulting output (Δ-output[o] — Δ-output[q]) can then either be merged (`Merge`) with the result of the original MapReduce job, yielding the same result as when processing the entire new input in regular fashion. Alternatively, the processed delta can be used by `Subsequent MapReduce jobs`, in fact reducing their workload.
In the rest of this chapter, we will take an close-up look at each of these steps.

Before and after MapReduce

The first choice in the computation of deltas is its position within the job ordering. Assume that you have multiple pipelined MapReduce jobs, i.e. subsequent jobs take the output of their predecessor as input (remember: web crawlers work this way). We are now left with the possibility to create a delta of the input data of that *job pipeline*, before actually invoking any MapReduce computation. This idea can be seen in the top section

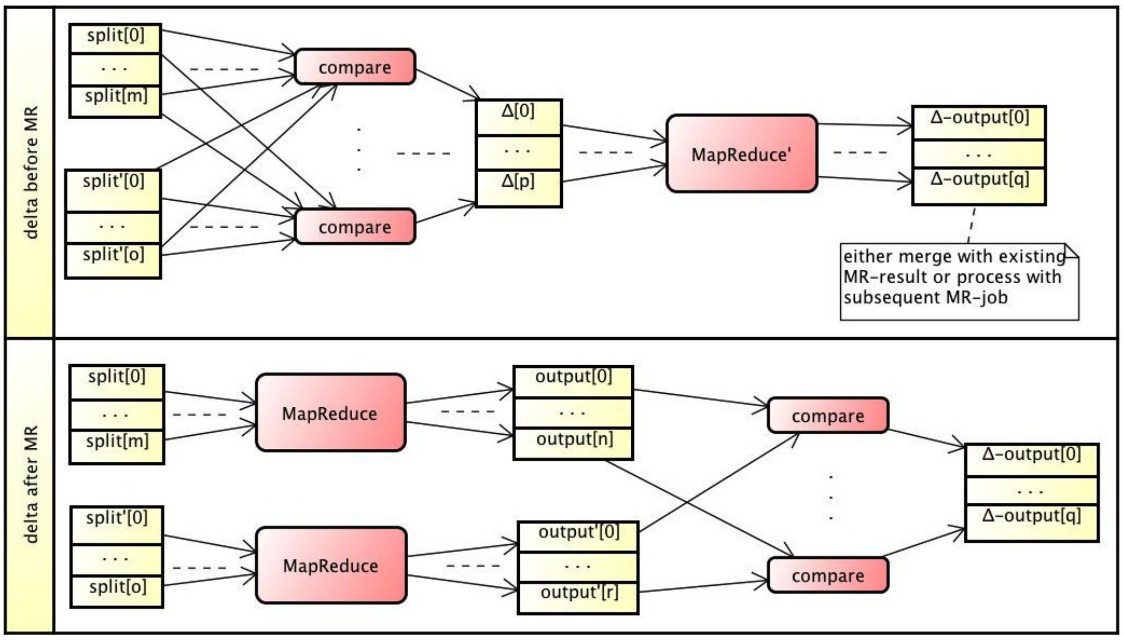

Figure 4.3: Computation of the delta before vs. after MapReduce.

of figure 4.3. If however, the first job decreases the volume of the data significantly by doing aggressive reduction of the input data, it might become feasible to compute the delta after this initial job (see lower part of the figure). Detailed analysis of read and write operations indicate that delta computation after MapReduce becomes possible if reduction reduces the input to one third or less of its initial size (using naive delta computation as presented in subsection 4.2.1).
Another advantage of creating the delta after an initial MapReduce job is the locality aspect: since both generations of data were created using the same partitioner, the data is likely to be aligned on the same nodes, allowing for local comparisons.

Naive approach

Various algorithms exist for computing the differences between two versions. For example, UNIX `diff` is based on solving the longest common subsequence problem. We omit a comparison of different algorithms, and ask the interested reader to refer to literature for comparisons of existing algorithms. For example, [23] compares various algorithms to compute the edit distance between two strings of characters. Instead we want to make use of the benefits gained by using MapReduce (such as massive parallelism and fault tolerance), and discuss techniques to compute deltas on the level of keys and values, using the MapReduce framework.
In order to compute a delta of two generations of a set of data items, both generations

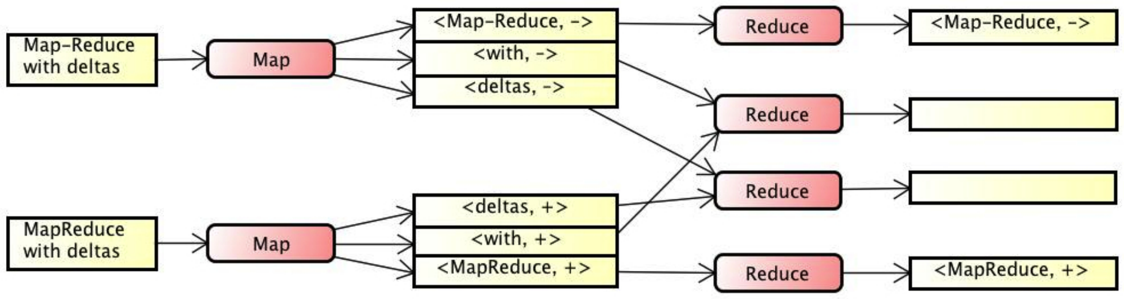

Figure 4.4: Example for creating a delta of two text documents using MapReduce.

need to be compared to each other. This requires for the data to be aligned in such a way that tuples sharing the same key can be accessed together. Using classic MapReduce, we can simply set up a job with both versions as input that does all comparisons at the reducer. A simple example of how a delta of two text documents could be created is illustrated in figure 4.4.

In order to distinguish the origin of a certain tuple, we use an input format implementation that wraps key and value in some pair-type, and passes the name of the corresponding file as the key. Since we want to augment each key-value tuple with a sign, we are basically producing triples (`<key,value,sign>`) in the mapper. However, since we want to avoid any modifications to existing frameworks, we can simply use said pair-type again (see listing 4.1). The implementation of the input format, as well as the pair type, can be found in the appendix (listings 3 & 2).

The reducer in listing 4.2 assumes for keys to be unique, as stated in the original MapReduce paper [1]. It receives 1-2 values per key, depending on whether a key exists only in one generation of the data, or in both. In the case of a single value, a potential deletion or addition becomes definite. In the case of 2 values, identical values cancel each other out, whereas two unequal values imply a modification that is expressed as an addition of the new value and a deletion of the old one.

However, in practice we have seen cases where multiple occurrences of the same key exist in given data sets. Therefore. we also need to consider the common case where an arbitrary number of occurrences of a given key is possible. Listing 4.3 introduces some changes to the previous reducer implementation in order to handle that case.

Unfortunately, the presented approach to compute deltas has some limitations due to the amount of necessary read and write operations. Figure 4.5 summarizes the required I/O operations to copy both versions in the mapper, *load* them into the reducer and finally write the delta.

This approximation does not even take into account the necessary intermediate sorting by key, as well as costs for MapReduce over the delta and final merging. Also,

```
private Collection<String> origFiles;

protected void setup(Context context) {
    origFiles = context.getConfiguration().getStringCollection("
       ORIG_FILES_SET");
}

public void map(String fileName, Pair<K,V> keyValue) {

    if(origFiles.contains(fileName)){
        emit(keyValue.first, new Pair(keyValue.second, true));
    }else{
        emit(keyValue.first, new Pair(keyValue.second, false));
    }
}
```

Listing 4.1: A pseudo code implementation of a mapper that assigns signs to tuples based on their origin.

$$\begin{array}{ll}
& \text{read}(V_0) + \text{read}(V_1) + \text{write}(V_0) + \text{write}(V_1) \quad \text{//map} \\
+ & \text{read}(V_0) + \text{read}(V_1) + \text{write}(\Delta_{0,1}) \quad \text{//reduce} \\
\hline
= & 4 * read(V) + 2 * write(V) + write(\Delta_{0,1})
\end{array}$$

Figure 4.5: I/O cost for naive delta computation (assuming that both versions have the same volume)

transfer of both generations over the network to the appropriate reducer nodes will add significant delay. Figure 4.6 indicates that classic MapReduce over the new version only requires about half the I/O costs of naive computation of the delta (no processing, no merging). This insight calls for a more efficient approach to compute deltas.

$$\begin{array}{ll}
& \text{read}(V) + \text{write}(V) \quad \text{//map} \\
+ & \text{read}(V) + \text{write}(V_{\text{reduced}}) \quad \text{//reduce} \\
\hline
= & 2 * read(V) + write(V) + write(V_{reduced})
\end{array}$$

Figure 4.6: I/O cost for regular processing of a version

```
public void reduce(K key, Iterable<Pair<V, Boolean>> values) {
    Iterator it = values.iterator();
    Pair<V, Boolean> v0 = it.next();
    if(it.hasNext()){
        Pair<V, Boolean> v1 = it.next();
        if(v0.first == v1.first){
            return; //same in both generations
        }else{
            //modified
            emit(key, new Pair(v0.first, v0.second ? "-" : "+"));
            emit(key, new Pair(v1.first, v1.second ? "-" : "+"));
        }
    }else{
        //only in one generation
        if(v0.second()){
            emit(key, new Pair(v0.first, "-")); //deleted
        }else{
            emit(key, new Pair(v0.first, "+")); //added
        }
    }
   }
```

Listing 4.2: A pseudo code implementation of a reducer, to create the delta of two versions.

```
public void reduce(Pair<K,V> keyValue, Iterable<Boolean> values) {
    int occ = 0;
    for(Boolean val : values){
        occ += val ? 1 : -1;
    }
    String sign;
    if(occ < 0){
        occ *= -1;
        sign = "+";
    }else{
        sign = "-";
    }

    for(int i = 0; i < occ; i++){
        emit(keyValue.first, new Pair(keyValue.second, sign));
    }
}
```

Listing 4.3: A reducer to create the delta of two versions, that deals with arbitrary occurrences of specific keys.

Using Map-side join

The limitations to the previous approach ask for a different strategy that reduces the necessary cost of I/O. Section 2.3.4 has introduced a useful abstraction called *map-side join*. If applied in the context of deltas, reduction can be omitted reducing the cost of I/O significantly (see figure 4.7), as network transfer between mappers and reducers is avoided. Please note that in order to compare these costs to regular processing of a version (as calculated in figure 4.6), costs for MapReduce of the delta and merging of the result with the result of the previous version need to be added.

$$\begin{array}{ll} & \text{read}(V_0) + \text{read}(V_1) + \text{write}(\Delta_{0,1}) \quad \text{//map-side join} \\ \hline = & 2 * read(V) + write(\Delta_{0,1}) \end{array}$$

Figure 4.7: I/O cost for delta computation using *map-side join*

While the general steps necessary in order to set up a map-side join have already been introduced in section 2.3.4, listing 4.4 shows pseudo code for a Hadoop implementation of a map-side join to create a delta of two versions.

```java
private Collection<String> origFiles;

protected void setup(Context context) {
    ...
}

public void map(PairWritable<K, V> keyValue, TupleWritable fileNames,
     Context context) {
    Text sign;
    int occ;
    for(Writable val : values){
        Text currentFile = (Text)val;
        occ += origFiles.contains(currentFile.toString()) ? -1 : 1;
    }

    if(occ > 0){
        //more occurences in the new file -> add them
        sign = new Text("+");
    }else if (total < 0){
        //more occurences in the old file -> delete them
        sign = new Text("-");
        occ *= -1;
    }else{
        //same amount in both
        return;
    }

    PairWritable<K,V> signedVal =
        new PairWritable<K, V>(keyValue.second(),sign);
    for(int i = 0; i < occ; i++){
        context.write(keyValue.first(),signedVal);
    }
}
```

Listing 4.4: A pseudo code implementation of a *map-side join* to create the delta of two version

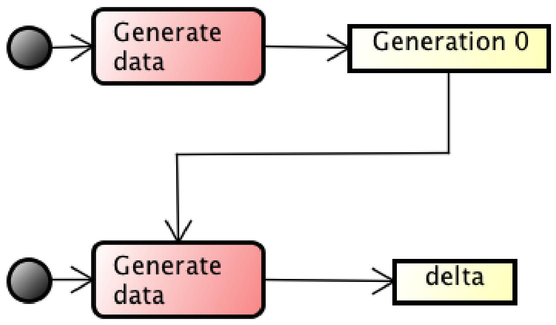

Figure 4.8: Creating the delta in streaming mode.

Streaming

The previous sections made clear that one limitation of using deltas is the overhead in I/O that is necessary for creation of the delta. In order to minimize that overhead, we propose an even more aggressive optimization, *streaming delta*, that avoids materialization of the second generation of input data (see figure 4.8).

In practice, the data processed by MapReduce jobs is often generated a priori, on the same cluster the jobs run on. The crawler, for example, creates generations of data in the *fetch*-step. In scenarios like this, the delta can be created in place, avoiding additional MapReduce computations. This requires the old version to be present for comparison during data creation. Figure 4.9 illustrates the associated costs. Please note that this approach also avoids the cost for materializing the entire second generation, as we only have to write the delta to disk.

$$\boxed{\text{read}(V_0) + \text{write}(\Delta_{0,1}) \quad //\text{streaming delta}}$$

Figure 4.9: I/O cost for delta computation using streaming delta in a MapReduce job

MapReduce jobs that create a delta in streaming mode are very application specific. The difficult part is to align the old version of the data in a way that makes sure that the new version of a data item is generated at the node its old version is hosted on. For example a web crawler could read old data, and generate the new data by extracting and *fetching* the URL associated with the respective input items. The fetcher used in the crawler presented in chapter 5 implements this concept.

Delta level

[2]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 listing 2.2:

```
for each $w$ in $words(d)$ do
  yield $\langle w, 1 \rangle$;
```

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 $\langle w, 1 \rangle$;
```

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" [4]. MapReduce computations with deltas are easily configured to exploit different levels. When computing the delta, as defined in listing 4.2, the modified-case (lines 9–11) must be refined to decompose v_0 and v_1 and to compute the delta at the more detailed level. In implementations of MapReduce, one can indeed exercise different levels. For instance, Hadoop [2] assumes that MapReduce jobs are configured with an `InputFormat` which essentially decomposes the input files (see section 2.3).

4.2.2 Processing Deltas

Assuming that we have successfully computed a delta, logically we will now process key-value-sign triples ($< K, V, sign >$) in the mapper. The mapper needs to pass the

2 This section was taken from [21]

sign to every tuple it produces in order to preserve it for reduction. We have to distinguish two cases: first, scenarios where the sign can be incorporated into the value (like negative occurrences in the WordCount example), and second, scenarios where this is not possible. For example in the context of the *Reverse Weblink Graph* job (section 2.4.3), tuples that are present in the first generation, but not in the second one, are marked as *to delete*. Unfortunately, this sign cannot be included directly into the value, but has to be attached explicitly. Reduction then has to preserve the sign of the tuples again, by either adequate reduction of positive and negative tuples, or by creating separate results.

4.2.3 Merging Deltas

After being processed using MapReduce, the delta needs to be combined with the original result in order to yield the same result as processing of the second generation of input data (see section 4.1). Attached negation signs are now used to *invert* the corresponding values from the original result. Merging can be done naively, using a MapReduce job that copies all data in the mapper and uses the same reducer as the original MapReduce job. However, in order to avoid unnecessary network transfer as we did with delta computation, we can again use map-side joins to merge the two results.

4.2.4 Limitation and Potential

Due to the necessary overhead for delta creation and merging, naive use of deltas in the context of a single MapReduce job does not promise significant improvement, if however, the map or reduce function perform tasks with worse than linear computational complexity, increasing volume of the input will eventually let those costs outweigh the overhead introduced by delta usage.
In practice, MapReduce jobs are often organized in pipelined or even graph-like workflows, where the output of one job serves as the input for the next one. In scenarios like these, the benefits gained from reducing the data volume by using deltas increase for each performed job. Chapter 6 examines these assertions.

4.3 IMPORTANCE OF LOCALITY

An important aspect in the creation of deltas is locality of data. In order to compute a delta, two or more versions need to be compared to each other. If these generations are not aligned, so that related data items are hosted on the same machines, costly network transfer becomes necessary in order to move the data items to each other. Therefore, it would be desirable to have some kind of influence on the locality of the output of a MapReduce job. Unfortunately, common MapReduce frameworks like Hadoop do not permit such a level of control. We do not seem to be the first to require such a level of control, as other approaches to determine locality of reducer nodes exist. In related work (chapter 7) we discuss such an approach, and possible improvements in the context of deltas are mentioned in future work 8.2.

5 A SIMPLE DELTA CRAWLER

In order to demonstrate the applicability of deltas in a complex scenario, we have implemented a web crawler that is modelled after Nutch and uses deltas to build and incrementally update an index.

5.1 ARCHITECTURE

The general architecture of our web crawler is identical with Nutch, as we basically used the same kinds of MapReduce jobs and data collections. However, their implementation differs, because we followed the vision presented in section 3.3 as shown in figure 3.3.

In contrast to Nutch, our crawler performs content-aware recrawling, where the fetcher reads previous generations of websites and creates the delta in streaming fashion directly in the fetch-job, as proposed in section 4.2.1. The crawler provides two different levels of delta computations: document level, where modified documents are replaced in their entirety, and word level deltas.
Besides dealing with content and metadata, the fetcher is also responsible for extraction of outlinks to other sites. These links are used to discover new sites, as well as for the creation of a link graph (`LinkDb`). Depending on the implementation of the index, a changed outlink may require a modification of the index entry of the target page. Furthermore, common ranking algorithms like PageRank rely on the link structure of pages, and hence might trigger updates of a site's score based on modified inlinks. These facts led to the decision to always create a detailed delta of a page's outlinks, regardless of the chosen delta-level.
In order to efficiently create word level deltas of contents or deltas of outlinks directly in the fetcher without using any further MapReduce jobs, we chose a *multiset* based approach. That is, we used an extended `HashMap` that maps from words (or links) to occurrences, and allows single access updates by using a mutable `Integer` container. Implementation of the `OccurrenceMap` and the `MutableInt` classes can be found in the appendix (listings 4 & 5). If all elements from the last generation are inserted with

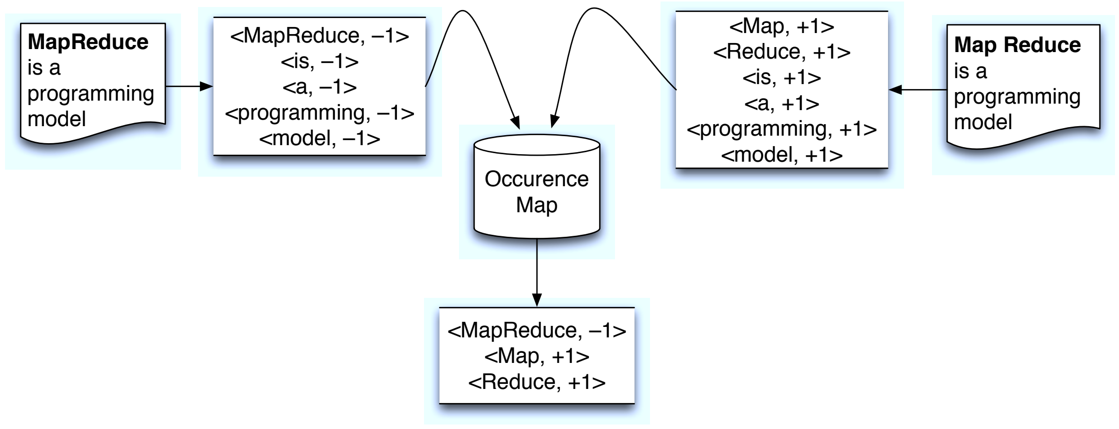
MapReduce is a programming model
<MapReduce, –1> <is, –1> <a, –1> <programming, –1> <model, –1>
Occurence Map
<Map, +1> <Reduce, +1> <is, +1> <a, +1> <programming, +1> <model, +1>
Map Reduce is a programming model
<MapReduce, –1> <Map, +1> <Reduce, +1>

Figure 5.1: Example of a delta using an `OccurrenceMap`.

negated counts into such an `OccurrenceMap`, and all elements from the current generation with positive counts, the map will finally contain the exact delta. Please note that this approach can also be applied if the last generation is represented as an initial generation together with a set of deltas. Figure 5.1 shows an illustrative example.

The output of a fetch-job is stored in a new segment directory with different subfolders for *content*, *metadata* and *links*. As these contain only the deltas of changed sites, subsequent jobs like update of the `CrawlDb` and link-inversion need to be *delta-aware*. While we introduced regular link-inversion in section 2.4.3, listing 5.1 shows an implementation of a delta-aware MapReduce job to compute a *Reverse Web-Link Graph*.

5.1.1 Locality of content

Section 4.3 introduced the problem of comparing different generations of data in the context of distributed storage. In the context of our delta-aware web crawler this is avoided by generating data by extraction of the URL from read content. Since Hadoop tries to assign map tasks to nodes that host the corresponding input data, websites will be likely fetched by the nodes that host previous versions of their data. However, in the context of changing recrawl intervals based on change-rate and score of a site as mentioned in section 3.3, the situation becomes more complicated. We might now be faced with imbalanced node utilization, as the amount of websites to crawl will differ per node, depending on the websites' recrawl policies.
In order to avoid this problem, we propose a crawler that does not realize detailed recrawl policies per site, but divides all URLs into sets with different recrawl policies.

```java
public static class LinkGraphMapper extends Mapper<Text, ListWritable
    , Text, ListWritable>{

  protected void map(Text srcUrl, ListWritable values, Context
      context) {
    for(PairWritable<Text, BooleanWritable> p : values){
      ListWritable inlinks = new ListWritable();
      inlinks.add(new PairWritable(src, p.second));
      context.write(p.first, inlinks);
    }
  }
}

public static class LinkGraphReducer extends Reducer<Text,
    ListWritable, Text, ListWritable>{

  protected void reduce(Text targetUrl, Iterable<ListWritable> values
      , Context context) {
    OccurenceMap<Text> map = new OccurenceMap();
    for(ListWritable l : values){
      for(PairWritable<Text, BooleanWritable> p : l){
        int occ = p.second ? 1 : -1;    //sign == true, if added link
        map.put(p.first, occ);          // incr/decr count by one
      }
    }
    //create list again
    ListWritable allInlinks = new ListWritable();
    for(Text srcUrl : map.keySet()){
      allInlinks.add(new PairWritable(srcUrl, map.get(srcUrl)));
    }
    context.write(targetUrl, allLinks);
  }
}
```

Listing 5.1: Delta aware MapReduce jon to invert links.

One could imagine the the following distinction: *VERY OFTEN* — *OFTEN* — *MEDIUM* — *RARE* — *VERY RARE*. We assume that *clean recrawls* will be performed periodically in order to compress multiple deltas and to recalculate clusters. These *clean re-crawls* could then be used to split URLs into sets, based on previous experience. If a delta aware re-crawl is being performed, the crawler checks for each *sub-list* whether it is due to fetch, and deals with each sub-list separately. This ensures that websites that are stored next to each other are always fetched together.

5.2 PAGE SCORING

As of version 1.2, Nutch uses *OPIC* (online page importance calculatation [24]) to calculate scores for crawled web pages. In contrast to offline approaches (like PageRank) that calculate the scores of all pages after crawling has finished, OPIC computes the importance of pages on-line, while crawling the web. Initial scores are assigned to websites, and are passed to all targets of outlinks discovered during crawling. As Nutch maintains a centralized *CrawlDb*, scores for pages can be accumulated there, and are distributed again, as soon as a page is being recrawled.
At first it may seem straightforward to only distribute scores of changed websites. However, an added or deleted link will influence the score of the target site, which again has to report its new score to all sites it is pointing to. Eventually, this leads to an avalanche-like updating of scores, that will expand the size of the delta in an unnecessary way, as sites that did not actually change still have to report a new score based on their predecessors.
PageRank, the most commonly used method to calculate importances of websites, was already discussed in the context of deltas in section 3.4.1. The approach introduced by [18] promises good results in conjunction with deltas. Unfortunately, even though [25] describes an application of the proposed algorithm in the context of Hadoop, no implementation is available online. We hence defer an implementation of a PageRank computation using deltas to future work.

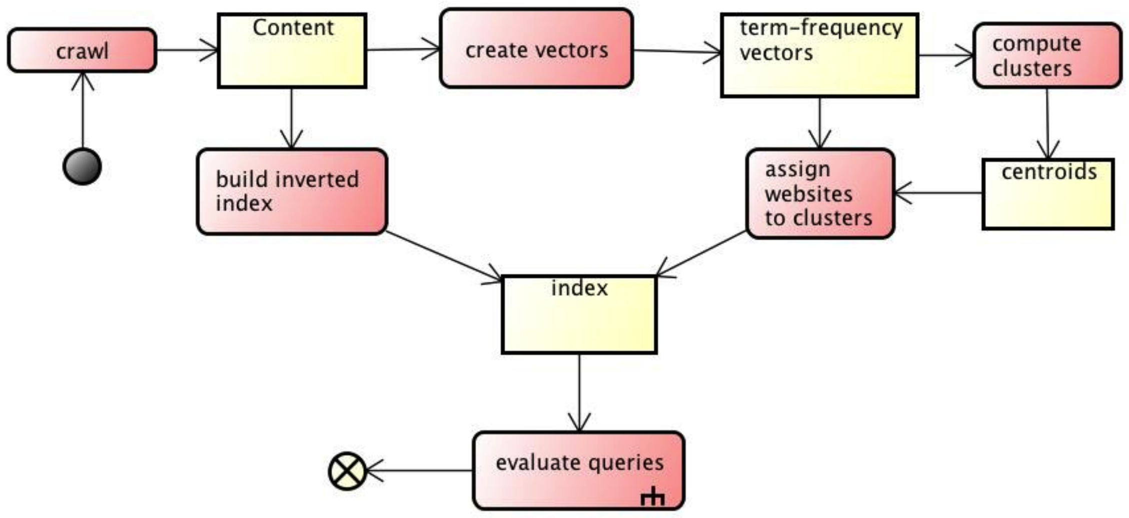

Figure 5.2: Overview of the necessary steps in indexing.

5.3 INDEXING

As figure 2.4 alluded, the final step in order to use the crawled data in a search engine is the creation of an appropriate index. Figure 5.2 illustrates the involved steps. The rest of this chapter explains how our implementation deals with each of them. Please note that any crawled content needs to be parsed (*normalized*) prior to index creation. Text that is related to formatting and functionality of the web page (like HTML tags, scripts, etc.), as well as unimportant words, so called stop words (like *the*, *and*, *a*, ...), have to be removed. This parsing already happens during crawling, either as part of fetching, or as a separate step.

5.3.1 Inverted Index

Recall from section 3.4.2 that an inverted index typically consists of mappings from words to lists of URLs along with the number of occurrences of this word on the respective websites. In order to find word occurrences, a WordCount job could be run for each website separately. However, as websites are typically small enough to be processed in memory, the overhead affiliated with a MapReduce job does not justify its use in this context. Listing 5.2 shows an example implementation that solves this task by combining the concept of link-inversion with in-memory word counting at the reducer.

```
public static class InvertedIndexMapper extends Mapper<Text, Text,
    Text, Text>{
  protected void map(Text url, Text value, Context context) throws
      IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      Text word = new Text(itr.nextToken());
      context.write(word,url);
    }
  }
}

public static class InvertedIndexReducer extends Reducer<Text, Text,
    Text, ListWritable>{
  protected void reduce(Text key, Iterable<Text> values, Context
      context) throws IOException, InterruptedException {
    OccurenceMap<Text> docMap = new OccurenceMap<Text>();
    for(Text url : values){
      //count occurences
      docMap.put(url);
    }

    ListWritable list = new ListWritable();
      for(Text url : docMap.keySet()){
        //create a list of all <URL,Count> pairs
        IntWritable count = new IntWritable(docMap.get(url).get());
        PairWritable<Text,MutableInt> p = new PairWritable<Text,
            MutableInt>(url,count);
        ListWritable l = new ListWritable();
        l.add(p);
      }
    context.write(key, list);
  }
}
```

Listing 5.2: A MapReduce implementation to create an inverted index

Each call to the mapper processes the content of a single website, which is passed as the value. The site is being identified by passing its URL as the key. The mapper simply emits each occurring word along with the current website's URL. Each call to the reducer then receives a list with all URLs containing a specific word, with one entry for each occurrence per website. In order to count the number of occurrences of each URL, we use again our `OccurrenceMap` implementation. After all occurrences have been collected in the `OccurrenceMap`, the map is converted into a list of pairs, each holding a URL along with its counter. The resulting lists can then be used by an index to directly access occurrence information for specific words.

In order to use deltas for the creation of an inverted index, some modifications need to be performed. Since the crawler has already determined the differences between two versions of content and only passes these changes as input to our *InvertedIndex* job, we can use negated occurrences for deleted content, like we did in our WordCount example (section 3.1). The resulting delta of the inverted index can finally be used to update the existing index.

5.3.2 Clustering

We use the distributed version of *k-means* that is provided by the *Mahout* project to cluster websites based on their content. As described in section 3.4.3, re-computation of clusters is not done for every recrawl. Instead, changed and new sites are only re-/assigned to the existing clusters. The delta provided by the crawler allows to decide which sites need to be considered. We rely on *Mahout* to create term-frequency vectors for each site, and to assign websites to the clusters based on those vectors. This cluster membership information can be used in the following to determine rankings of search results during evaluation of queries.

5.3.3 Evaluation of Queries

Once all previous steps have been performed, search queries can be evaluated on the grounds of the computed results. As the user expects URLs with the highest relevance to be returned first, the order in which results are being returned plays a significant role. Figure 5.3 illustrates the involved workflow.
After the different search terms have been extracted from the query, a direct lookup in

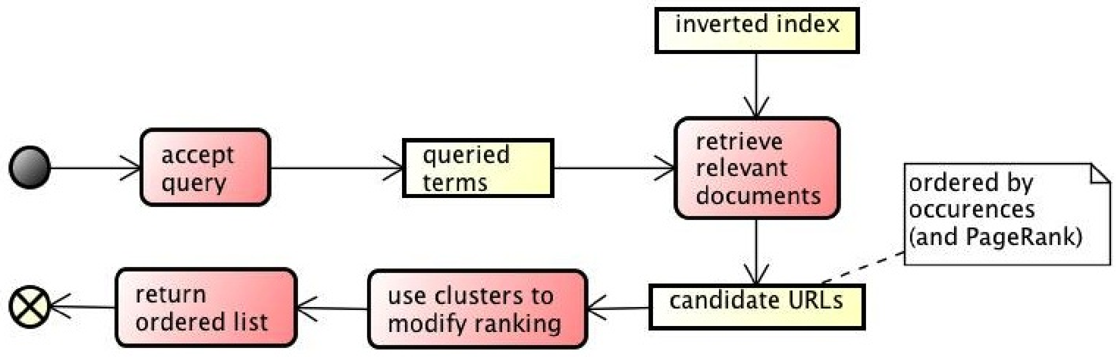
inverted index
accept query
queried terms
retrieve relevant documents
ordered by occurences (and PageRank)
candidate URLs
use clusters to modify ranking
return ordered list

Figure 5.3: Activities performed during evaluation of a search query.

the inverted index is done for each term, in order to find possible candidate websites that contain them. The resulting list is sorted in ascending order by the number of occurrences, favouring websites that contain multiple queried terms. Once page scoring has been implemented, the sites' scores will also be taken into consideration for the ordering. As cluster membership expresses similarity to some degree, sites located in the same cluster as the top-ranking results can be moved up in the list, or even become included although they do not actually contain occurrences of queried terms. Once the final ordering of the results has been determined, they can be returned to the user. In order to enable high responsiveness, as much of the index as possible should be present in memory. Therefore content of sites is not being stored in the index itself. Instead, page summaries and access to cached pages need to be handled by reading of the individual segments.

6 BENCHMARKS

This chapter[1] presents various 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.

6.1 TERABYTE SORT

TeraByte Sort is an established benchmark for testing the throughput on a MapReduce implementation. TeraByte Sort (or the variation—MinuteSort) is run as an annual benchmark contest, which was won in 2008 by Hadoop as the first Java or open source implementation. The task is to sort (in one typical configuration) 100-byte records, out of which 10 bytes constitute the key [26, 27, 28]. Hadoop's winning implementation has since been bundled with every release in order to provide a benchmark for Hadoop clusters.

6.1.1 Hadoop's TeraSort

In order to produce the input data, a generator is provided that randomly creates keys and values. Sorting is already built into the MapReduce framework, as all tuples are presented in increasing order of keys to the reducers. Therefore, *TeraSort* uses the default implementation of mapper and reducer, which simply copy the data. The only necessary task is to enforce a total ordering of keys among reducers. This is achieved by sampling keys form the input, to build a trie for fast assigning of keys to reducers.

During our tests we encountered uneven reducer utilization for MapReduce jobs on

1 Parts of this chapter were taken from [21]

2 Cluster characteristics: we used Hadoop version 0.21.0 on a cluster of 40 nodes (50 for later tests) 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.

sorted data that were necessary for our second benchmark (see section 6.2). This forced us to implement our own version of TeraSort, which only differs by usage of datatype `long` (8 bytes) for keys instead of byte sequences of length 10. The missing two bytes are added to the value to leave the total at 100. Using numbers as keys allowed us to simplify partitioning. We ran tests (with the same input sizes as our sorting benchmark) that showed no notable differences in performance between the two implementations.

6.1.2 Results

Figure 6.1 shows the benchmark results for our version of TeraByte Sort. The incremental version computes the delta using a *complete* MapReduce job, as described in section 4.2.1. We also tested the optimized incremental versions: (map-side) "join" and "streaming" as described in sections 4.2.1 and 4.2.1 respectively. The shown costs for the delta-aware versions include all costs necessary to obtain the same result as the non-incremental computation, specifically: cost for delta creation, processing of the delta, and merging. It is important to note that we implemented merge by map-side join.
It is not surprising that the non-incremental version is faster than all incremental versions except for streaming. 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. 6.1. 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.

6.2 PIPELINED JOBS

In practice, MapReduce jobs are often organized in pipelines or even more complicated networks—remember the use case of crawling in sec. 3.3. In such compounds, the ben-

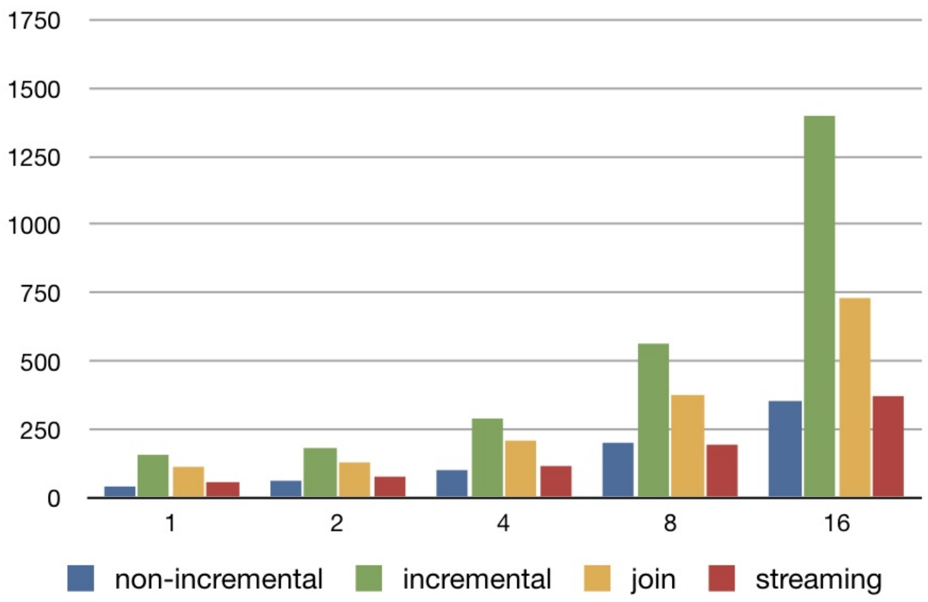

Figure 6.1: **Runtimes in seconds (y-axis) for non-incremental and incremental TeraByte Sort for different input sizes in GB (x-axis), where the size of the deltas for the incremental versions is assumed to be 10 % of the input size.**

efit 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. Also the reduce step does not decrease the volume of the data, which results in high cost for merging.

The results are shown in Fig. 6.2. 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.

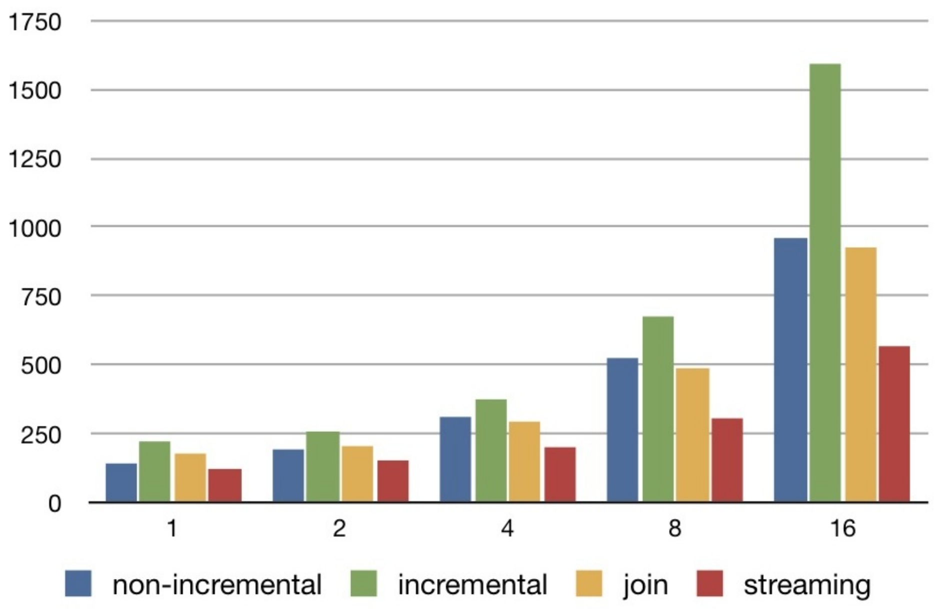

Figure 6.2: Sort followed by three copy jobs.

6.3 SIMPLE DELTA CRAWLER

In order to show the possible speedup in compound MapReduce jobs on a more realistic scenario, we ran several tests with our implementation of a delta-aware web crawler that has been introduced in chapter 5, along with the mentioned indexing steps. In order to create a stable testing environment that is not influenced by unstable change frequencies of websites we modified our crawler slightly to control the size of the delta. For our tests, a delta that is 10 % of the input size was chose again. In order to highlight the costs for recrawling, we crawled an initial set of URLs, and built the corresponding structures like *CrawlDb*, *LinkDb*, clusters and inverted index. The tests then comprised a single recrawl that considers only sites that have already been fetched, and therefore can be compared to their previous version. Specifically, a single recrawl comprised the following steps (each implemented as a MapReduce job):

- Generate list of URLs to fetch.
- Download the content.
- Update the *CrawlDb*.
- Invert links found on websites.
- Create term-frequency vectors from content.

- Assign websites to existing clusters.
- Create inverted index.

Figure 6.3 shows the resulting execution times of the non-incremental version ("No delta") and incremental recrawls, with two different levels of delta creation as described in section 5.1 ("Word delta" & "Doc delta"). For a more detailed overview, the percentages of the incremental recrawls are expressed relative to the non-incremental crawls in figure 6.4. The benefits gained by using deltas increase with the size of the URL set, since the complexity of clustering grows in a non-linear fashion. That is, assigning of sites to clusters has complexity $O(n^2)$, as both the number of centroids and the number of vectors they are being compared to increase with the input.

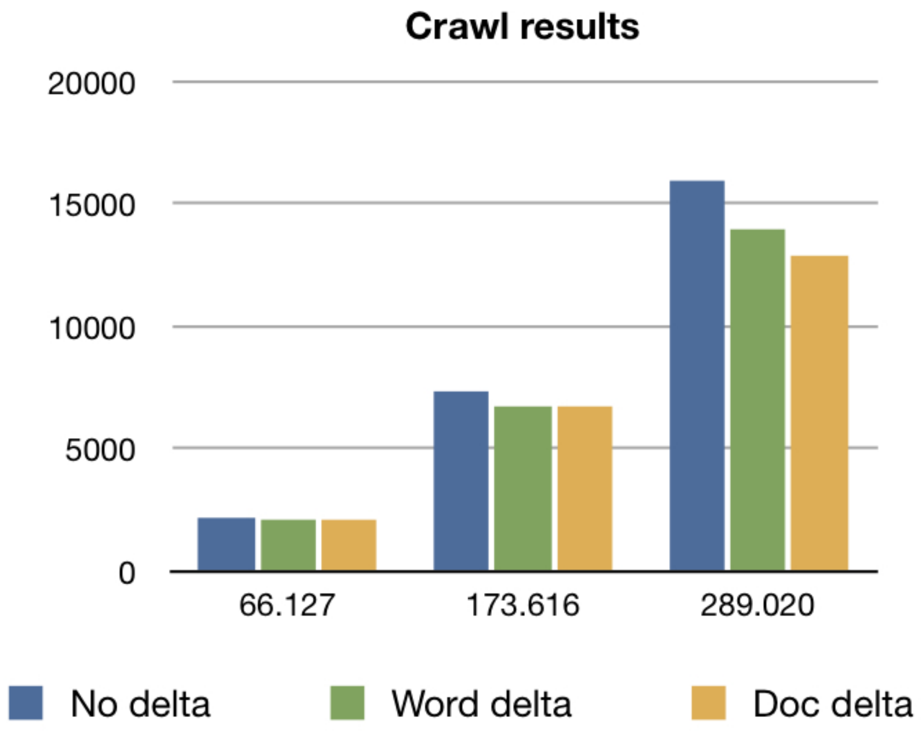

Figure 6.3: **Runtimes in seconds (y-axis) for non-incremental and incremental recrawls of a web crawler, based on an existing crawl base of different sizes in URLs (x-axis). Results for delta creation on document– and word–level are given.**

While recrawling again and again, one could be concerned that the cost of including multiple deltas from multiple previous recrawls into the creation of the current delta would slow things down. However, we argue that the cost of reading and uniting multiple deltas is negligible in this context. In order to verify this claim, we conducted

Cost (Percentage)	66.127	173.616	289.020
No delta	100	100	100
Word delta	96.2	91.7	87.6
Doc delta	96.5	91.7	80.8

Figure 6.4: Percentages of crawl execution times, relative to the non-incremental version.

three recrawls, each building on the previous one and producing deltas of the same size (10 %). It is difficult to compare the absolute duration times, as later recrawls exclude URLs previous recrawls have experienced transfer issues with. We hence compare the execution times relative to the number of URLs actually generated for fetching. Figure 6.5 shows that no difference is notable for the first three recrawls. We assume that in reality the number of recrawls will not increase significantly, as the index will be periodically recreated from scratch, in order to compress the data and to recalculate the clusters.

Cost per 1000 URLs	Word delta	Doc delta
1	46.3	46.3
2	46.6	46.4
3	46.8	46.3

Figure 6.5: Crawl execution times in seconds per 1000 URLs, for multiple incremental iterations.

Due to network latency and transfer time of the websites, the fetch step dominated the execution times with about 90 % of the total duration. Therefore, we conducted a second test that omits the actual downloading of the content and simulates changes using the content from the previous fetch step. The results in figure 6.6 show the potential of deltas in the context of MapReduce computations. Now, the dominant factor is the assignment of websites to existing clusters. With a delta size of 10 % of the original crawl set, we only need to compare 10 % of the sites to the existing centroids, cutting this cost in a linear fashion to 10 %.

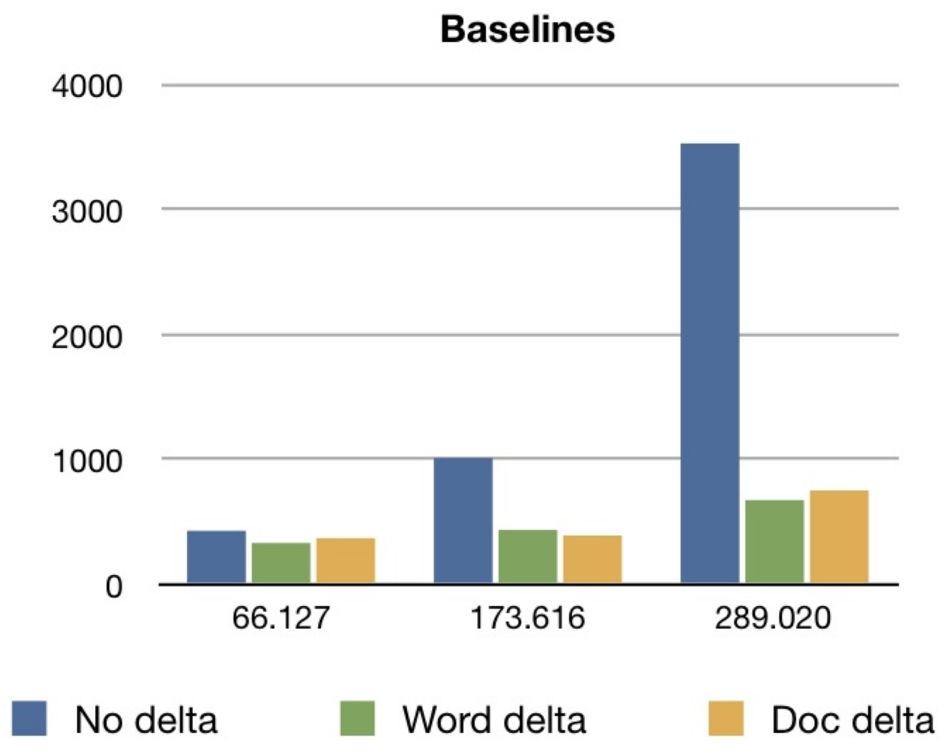

Figure 6.6: Baseline of recrawls that expresses the execution times without the overhead incurred by downloading web content.

7 RELATED WORK

[1]*Percolator* [17] 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 [29], Google's high performance proprietary database system. Percolator adds trigger-like procedures to BigTable columns, which 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 as 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).

In contrast to the acyclic nature of MapReduce, *Spark* [30] focuses on cyclic computations such as iterative jobs and interactive analysis (repeatedly ad-hoc querying of datasets) in the context of a new programming model. It handles these classes of computations by letting the user specify RDDs (resilient distributed datasets), read-only collections of objects, which are cached in memory across machines and reused in multiple MapReduce-like parallel operations. *Twister* [31], 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* [32], 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 to 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.

1 This chapter was taken in parts from [21]

Dryad [33, 34] 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 [35] enhances MapReduce so it can 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 [2], an open source Java implementation of Google's MapReduce framework [1]. Hadoop's MapReduce-component [36] is built on top of HDFS [8], the *Hadoop Distributed File System* which has been modeled after the *Google File System (GFS)* [9]. Hadoop happens to provide a form of streaming (i.e., *Hadoop Streaming*) for the composition of MapReduce computations [37]. This form of streaming is not directly related to streaming in our sense of delta computation.

MapReduce Online [38] 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* [39]. 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* [40], 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 [3], and a systematic review of published MapReduce use cases [22].

Common algorithms for page importance calculations (like [16]) rely on explicit creation of a web graph and iterative score calculations on that graph until scores converge. [41] presents a method to continuously refine page scores while the web is being

crawled. This is especially useful in evolving graphs, where computations need to be updated frequently. It can also be used to focus crawling on high scoring pages, as the ranking is already available during crawling. For instance Nutch 1.2 is implemented using this algorithm. An approach to update *PageRank* computations in the context of changes in the web is introduced by [18]. 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.
[42] introduces locality- and fairness-aware partitioning in order to deal with significant variance in both intermediate key frequencies and their distribution among reducer nodes. As describted in section 4.3, delta-aware computations would benefit highly from the ability to control key-range assignment to specific reducer nodes. Possible contributions in this context are described in future work (sec. 8.2).

After crawling and processing multiple web sites, Nutch relies on Apache's Lucene project [12] to create and maintain a search index. In the context of indexing, several machine learning algorithms are applicable. Most of them are computationally expensive, as they are computed iteratively, hence parallelizing this class of algorithms yields significant improvements. [20] describes the initial approach to use the MapReduce programming model for parallelization of machine learning algorithms. While first used in the Lucene project, this approach was eventually extracted to the standalone Apache project Mahout [19].

8 CONCLUSION

8.1 RESULTS

We have described how to use deltas in the context of MapReduce computations in order to reduce the volume of input data that drives the complexity of MapReduce jobs. When using deltas, different steps including creation of the delta, processing the delta in an augmented MapReduce job, and merging of the resulting output with previous output need to be performed. We have discussed each of these steps, and concluded that use of deltas in single MapReduce jobs has to meet computationally complex map or reduce tasks for the benefits to outweigh the additional costs. However, usage of deltas in compound MapReduce scenarios provides significant speedup, as the benefits add up over each pipelined job.

In order to verify the correctness of deltas, we deduced a simple correctness criterion. In contrast to most incremental approaches to MapReduce, deltas are a very general model and can be used without any modifications to existing frameworks. They further provide predictable speedup, allowing for abstract analysis of their applicability in concrete scenarios.

8.2 FUTURE WORK

[1] This thesis tried to illustrate the usage of deltas in the context of the complex MapReduce scenario of a web crawler. While we implemented a simple web crawler that performs all basic steps to create a searchable index, modification of an existing web crawler like Nutch to be delta-aware would further back our approach. Additionally, showing applicability of deltas in the context of other complex MapReduce use cases would be useful.

Further, MapReduce tasks with high computational complexity of the mapper and/or the reducer should be benchmarked in order to further examine applicability of deltas

1 This section was taken in parts from [21].

in single MapReduce tasks. For example, assigning websites to clusters has quadratic complexity if the number of clusters depends on the input. In scenarios with worse than linear complexity, benefits of deltas increase as seen in section 6.3.

Since we still lack a delta-aware implementation of a ranking algorithm, usage of deltas to incrementally compute PageRank as described by [18] would further strengthen our web crawler use case.

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 discussion of related work 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 of storing multiple versions of data using timestamps. This could facilitate delta creation substantially.

While our approach enables incrementality without the need to modify any existing framework, one could imagine a *delta-layer* on top of a MapReduce framework like Hadoop that shields the user from basic sign maintenance. The map step in a delta job needs to maintain the sign of the tuples, i.e. whether they have been added or deleted, and hence requires augmentation of key-value tuples to include some kind of sign. This could be facilitated by said *delta-layer*, by taking such a *key-value-sign* triple, extracting the sign prior to the map-call, and attaching it to every tuple produced by the mapper. Similarly, this *delta-layer* could automatically take care of signs in the reduction step, by reducing positive and negative values separately instead of merging them into one result. Even automatic merging with previous results might be possible, by requiring the user to specify a *merge-function*, similarly to map- and reduce-function.

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.

9 BIBLIOGRAPHY

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

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

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

[5] “Apache Subversion (SVN),” http://subversion.apache.org/.

[6] “Apache Nutch,” http://nutch.apache.org/.

[7] “Alphabetical list of institutions that are using Hadoop,” http://wiki.apache.org/hadoop/PoweredBy.

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

[9] 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.

[10] T. White, “File Appends in HDFS,” http://www.cloudera.com/blog/2009/07/file-appends-in-hdfs/, 2009.

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

[12] “Apache Lucene,” http://lucene.apache.org/.

[13] H. C. Yang Zhou and J. X. Yu, “Graph clustering based on structural/attribute similarities,” *Proc. VLDB Endow.*, vol. 2, pp. 718–729, August 2009.

[14] Y. Zhou, H. Cheng, and J. X. Yu, “Clustering Large Attributed Graphs: An Efficient Incremental Approach,” in *Proceedings of the 2010 IEEE International Conference on Data Mining*, ser. ICDM ’10, 2010, pp. 689–698.

[15] J. Norstad, “ MapReduce Algorithm for Matrix Multiplication,” http://homepage.mac.com/j.norstad/matrix-multiply/index.html, 2009.

[16] 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.

[17] 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.

[18] 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.

[19] “Apache Mahout,” http://mahout.apache.org/.

[20] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, “Map-Reduce for Machine Learning on Multicore,” in *NIPS*, 2006, pp. 281–288.

[21] R. Lämmel and D. Saile, “MapReduce with Deltas,” in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2011*, 2011.

[22] A. Brandt, “Algebraic Analysis of MapReduce Samples,” 2010, Bachelor Thesis, University of Koblenz-Landau.

[23] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, “A comparison of string distance metrics for name-matching tasks,” 2003, pp. 73–78.

[24] S. Abiteboul, M. Preda, and G. Cobena, “Adaptive on-line page importance computation,” in *Proceedings of the 12th international conference on World Wide Web*, ser. WWW ’03. ACM, 2003, pp. 280–290.

[25] I. B. Abdullah, “Incremental pagerank for twitter data using hadoop,” Master’s thesis, University of Edinburgh, 2010.

[26] “Sort Benchmark,” web site http://sortbenchmark.org/.

[27] O. O'Malley, "TeraByte Sort on Apache Hadoop," 2008, contribution to [26].

[28] A. C. Murthy, "Winning a 60 second dash with a yellow elephant," 2009, contribution to [26].

[29] 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.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *2nd USENIX workshop on Hot Topics in Cloud Computing*, 2009.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

[38] 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.

[39] 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.

[40] 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.

[41] S. Abiteboul, M. Preda, and G. Cobena, "Adaptive on-line page importance computation," in *WWW '03: Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 280–290.

[42] S. Ibrahim, H. Jin, L. Lu, B. He, L. Qi, and S. Wu, "LEEN: Locality/Fairness- Aware Key Partitioning for MapReduce in the Cloud," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, 2010, pp. 17–24.

```java
public class Pair<A, B> {
    public A first;
    public B second;

    public Pair(A first, B second) {
        super();
        this.first = first;
        this.second = second;
    }

    public String toString() {
        return "(" + first + ", " + second + ")";
    }
}
```

Listing 1: An implementation of a pair-type.

```
public class PairWritable<T extends Writable, V extends Writable>
    implements WritableComparable {
  private Pair<T,V> value;
  private Class fstClass;
  private Class sndClass;
  private AtomicReference<Configuration> conf;

  public PairWritable() {
    value = new Pair();
    this.conf = new AtomicReference<Configuration>();
  }

  public PairWritable(T fst, V snd) { ... }

  public T first(){ return value.first(); }

  public V second(){ return value.second(); }

  public void readFields(DataInput in) throws Exception {
    //first, read in the class-names
    fstClass = Class.forName(in.readUTF());
    sndClass = Class.forName(in.readUTF());
    //second, read in the values
    Writable fst = (Writable)ReflectionUtils.newInstance(fstClass,
        conf);
    fst.readFields(in);
    Writable snd = (Writable)ReflectionUtils.newInstance(sndClass,
        conf);
    snd.readFields(in);
    value.first = (T)fst;
    value.scond = (V)snd;
  }

  public void write(DataOutput out) throws IOException {
    //first, write out the classes of the Pair-element
    out.writeUTF(fstClass.getName());
    out.writeUTF(sndClass.getName());
    //then the values
    value.first.write(out);
    value.second.write(out);
  }
}
```

Listing 2: A wrapper class for the Pair class (listing 1) that implements Hadoop's `Writable` interface.

```java
public class FileNameInputFormat<K,V> extends FileInputFormat<Text,
    PairWritable<K,V>> {

  static class FileNameRecordReader<K,V> extends RecordReader<Text,
      PairWritable<K,V>> {
    private RecordReader<K,V> reader;
    private Text key;
    private PairWritable value;
    private Text fileName;

    public void initialize(InputSplit split, TaskAttemptContext
        context) {
      ...  //initialize reader
      this.fileName = new Text(((FileSplit) split).getPath().toString
          ());
    }

    public void close() throws IOException { reader.close(); }

    public Text getCurrentKey() { return key; }

    public PairWritable<K,V> getCurrentValue() { return value; }

    public boolean nextKeyValue() throws Exception {
      if (key == null) {
        key = new Text(fileName);
        value = new PairWritable<K,V>();
      }
      if (reader.nextKeyValue()) {
        value.setFirst(reader.getCurrentKey());
        value.setSecond(reader.getCurrentValue());
        return true;
      } else { return false; }
    }
  }

  public RecordReader<Text, PairWritable<K,V>> createRecordReader(
            InputSplit split, TaskAttemptContext context) {
    return new FileNameRecordReader<K,V>();
  }
}
```

Listing 3: An `InputFormat`, that passes the name of the source file as the key.

```java
public class MutableInt implements Writable{
  private int value;

  public MutableInt() { value = 0; }

  public MutableInt(int value){ this.value = value; }

  public int inc() { return ++value; }

  public int incBy(int amount) {
    value += amount;
    return value;
  }

  public int dec() { return --value; }

  public int decBy(int amount) {
    value-= amount;
    return value;
  }

  public int get() { return value; }

  public void set(int newVal) { value = newVal; }

  @Override
  public void write(DataOutput out) throws IOException {
    IntWritable w = new IntWritable(value);
    w.write(out);
  }

  @Override
  public void readFields(DataInput in) throws IOException {
    IntWritable w = new IntWritable();
    w.readFields(in);
    value = w.get();
  }
}
```

Listing 4: A container class providing a mutable `Integer`

```java
public class OccurrenceMap<T> extends HashMap<T, MutableInt>{

    public OccurrenceMap(){ super(); }

    public OccurrenceMap(int initialCapacity){  super(initialCapacity
       ); }

    public int put(T elem){ return put(elem, 1); }

    public int put(T elem, int occurences){
        MutableInt currentCount = get(elem);
        if(currentCount == null){
            currentCount = new MutableInt(0);
            put(elem, currentCount);
        }
        currentCount.incBy(occurences);
        if(currentCount.get() == 0){
            remove(elem);
        }
        return currentCount.get();
    }


}
```

Listing 5: A proxy for a `HashMap`, counting occurences of elements using single access.