Marimba: A Framework for Making MapReduce Jobs Incremental

Johannes Schildgen*, Thomas Jörg†, Manuel Hoffmann*, Stefan Deßloch*
*University of Kaiserslautern, Germany, {schildgen, m_hoffmann09, dessloch}@cs.uni-kl.de
†Google, Munich, Germany, tjoerg@google.com

***Abstract*—Many MapReduce jobs for analyzing Big Data require many hours and have to be repeated again and again because the base data changes continuously. In this paper we propose Marimba, a framework for making MapReduce jobs incremental. Thus, a recomputation of a job only needs to process the changes since the last computation. This accelerates the execution and enables more frequent recomputations, which leads to results which are more up-to-date. Our approach is based on concepts that are popular in the area of materialized views in relational database systems where a view can be updated only by aggregating changes in base data upon the previous result.**

***Keywords*-MapReduce; Hadoop; incremental; framework**

I. Introduction

In the past few years, the MapReduce paradigm [5] and its open-source implementation Hadoop [1] gained popularity for analyzing Big Data. Soon high-level languages like Pig Latin or Hive in order to query large amounts of structured data were established. This way, commonly-used tasks like joins and filters need not be implemented manually. Instead, a declarative language similar to SQL is used to describe the problem. Furthermore, many optimizations like index support had been added to Hadoop, but these modifications only make MapReduce faster and easier to use when dealing with structured data. In this paper, we will go back to the roots of MapReduce, namely processing unstructured data like websites, blog posts and other large amounts of text or graph data.

As the amount of input data is very large, MapReduce jobs often take many minutes or hours. Moreover, this input data changes permanently, which is why these jobs have to be repeated again and again. This is similar to materialized views in relational databases: A long-running query is executed periodically to update a materialized view. In that area many optimizations exist. The main idea is to just analyze deltas in the base data, which is data that has changed since the last computation. Along with the previous result, the new result is produced. When this kind of incremental computation is possible, we call the materialized view *self-maintainable*.

The main difference is defining a materialized view by a declarative query, opposed to a MapReduce job that is implemented by the two functions Map and Reduce in a specific programming language. Additionally, a MapReduce job can process any data (unstructured text, graph data, ...). Materialized views can benefit from statistics and indexes on tables which are not available in MapReduce. A NOSQL database like Google's Big Table or its open-source implementation HBase is often used as an input and output for MapReduce jobs. They differ from relational databases not only in the lack of a well-defined schema and a non-declarative query language. Big Table does also not support secondary indexes. Thus, data can only be read and written via primary-key access or a full scan. When not using a database but the Google File System (GFS) or the open-source Hadoop Distributed File System (HDFS), text files can only be read sequentially and written in an append-only mode.

In the next chapter, we introduce some related work and in chapter III we present our idea to make MapReduce jobs self-maintainable. Based on these findings, in chapter IV we introduce *Marimba*[10], a framework for incremental MapReduce jobs, and some example jobs in chapter V. We will test the performance of incremental Marimba jobs in chapter VI. In chapter VII, the conclusion of this paper is drawn.

II. Related Work

There is some research for making MapReduce jobs incremental. At Google, they modified the MapReduce programming model and created Perlocator [12] which replaces the former MapReduce-based web-indexing system. Instead of doing full recomputations, Perlocator processes data by using so-called observers which are similar to database triggers.

Another approach which introduces a new programming model for continuous bulk processing (CBP) is presented in [9]. A CBP computation generates a continuous output based on continuously arriving input data. This approach as well as Perlocator are far apart of materialized views. They both use a new programming model to describe the reaction on changes in the input data. This forces the user to use this new programming model which is different from the functions Map and Reduce, so existing MapReduce jobs cannot be reused.

In [3], Incoop is presented, an approach for incremental data processing which executes a normal MapReduce job in an incremental way by caching Map and Reduce outputs and re-executing the map phase only for splits that changed and the reduce phase only for changed intermediate key-values.

Similar to the caching in Incoop is IncMR [15]. Hadoop jobs can be ran incrementally without changing the code by mapping only the new data and executing the Reduce function not only on the current Map output but also on the Map output of the previous computation. The latter data is called state data in IncMR.

On the one hand, the benefit of Incoop and IncMR is the possibility of making many existing MapReduce jobs incremental simply by caching intermediate results from former computations. On the other hand, caching needs to write a huge overhead.

i^{2}MapReduce [16] is one different idea which focuses on iterative computations. These computations—like PageRank— have to be computed multiple times in a row until the result converges. i^{2}MapReduce remains the state of a MapReduce job beyond the runtime of the job so that computed values can be reused. This approach is not applicable to non-iterative MapReduce jobs.

DryadInc [13] also caches intermediate data for reuse in recomputations. However it only supports insertions but no deletions or updates.

Our idea is to benefit from these caching and non-caching approaches. For one thing the MapReduce programming model is modified just in a minimal way so existing jobs can easily be made incremental. For another thing, intermediate results are not being cached so there is no large overhead.

III. How to Make MapReduce Jobs Self-Maintainable

A. Detecting Deltas

An incremental computation means that only the changes since the previous computations will be analyzed. So, first of all those changes have to be determined. Generally, one can specifying three input sources—one for inserted, one for preserved and one for deleted data—or the changes can be detected automatically by use of change-data capture (CDC). In [6], CDC approaches for wide-column stores like HBase or Cassandra are compared. We developed one input format for timestamp-based CDC in HBase which reads only those cells of an input table that did change since the previous job execution. This approach is very efficient because HBase provides a history of each cell value together with timestamps, and a time-range scan is supported.

One other CDC approach would be the usage of a log file. This can be used if the storage systems logs insertions, deletions and updates. On systems which support triggers or ECA rules, trigger-based CDC can be used, so that after every change operation on the base data a specific action is performed (flagging a record or building sets of inserted and deleted items). If triggers are not supported, one can manually set an audit column whenever base data is written. This is an annotation for a change in an item. At analysis time, items with an empty audit column can be skipped, the others are treated as inserted, deleted or changed. Another possibility to capture changes is the snapshot-differentials approach in which the current snapshot of the base data is compared to the state of the previous computation completely.

B. Incremental WordCount

The textbook example *WordCount* is often used to explain the usage of MapReduce: A *Map* function reads a line of text and emits for each word w in it a key-value pair $(w, 1)$. For each key, all values are added in a *Reduce* function, so that the output of this task is a list of all words together with their occurrences. When we want to make WordCount self-maintainable [7], first of all the input of the job is not the whole text data, but only the deltas together with the result of the former computation. The Map function distinguishes between inserted data, deleted data and old results. Lines which are updated are treated as if they would be deleted and inserted again. In case of inserted data, the algorithm behaves as usual: it emits a 1 for each word. Is the line deleted, -1 is emitted. Old rows are just passed on to the Reduce function.

An example: We already counted all the words in a long text and now a line "That's all" is removed. The reducers for the keys "That's" and "all" now receive two values each: A -1 and the number from the previous result. Both are added, so that the numbers of "That's" and "all" are now decreased by one. For all other words from the old result, the mappers just have to pass them to the reducers, and the reducers put them out again. As the output usually overwrites the old result, this approach is called *Overwrite Installation* (see figure 1).

In this example, the amount of changes is very low, but we had to read the complete previous result. An alternative to Overwrite Installation is the *Increment Installation*, which avoids reading the previous result (see figure 2). So in our example, there is only one input line for the Map function ("That's all") and two intermediate key-value pairs (("That's", -1), ("all", -1)). In this approach, the reducer must not overwrite the old result but increment it so the current value has to be read and updated by adding the calculated value (here: -1). This kind of increment operation is supported by HBase [2]. As a result, only the inserted and deleted lines have to be read. A disadvantage is the expensiveness of this increment operation because the corresponding line has to be fetched in a non-sequential way.

As explained further below, it will be shown that the Increment strategy is useful when there are only few data changes. In case of many changes, the Overwrite strategy is faster.

C. Formalization

Evident from previous examples, it is possible to use the Overwrite and Increment strategy in MapReduce jobs. We developed further jobs and noticed that they all follow

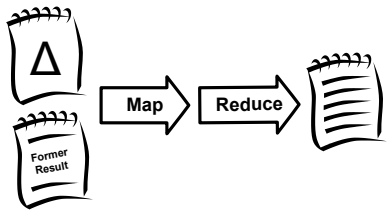

Figure 1. Incremental MapReduce - Overwrite Installation

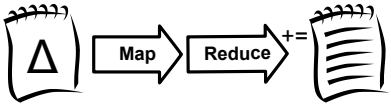

Figure 2. Incremental MapReduce - Incremental Installation

a common pattern: A Map function has to read deltas as well as the previous result and distinguish between deleted, inserted, preserved values and old results. A Reduce function has to aggregate all the values and write them either in an overwriting or in an incremental way. We realized that all of our implemented incremental MapReduce jobs belong to a common class. This class of self-maintainable MapReduce jobs we now want to define by modifying the MapReduce paradigm. Given the following two functions:

$$Map : K \times V \rightarrow List < K' \times V' >$$
$$Reduce : K' \times List < V' > \rightarrow (K'' \times V'')$$

The Map function produces intermediate key-values and the Reduce function generates an output by aggregating all intermediate values for one key. Let $\circ$ be an aggregation function and $v_1, ..., v_n \in V'$ a set of intermediate values, then the aggregated value $a \in V'$ can be computed as $a = v_1 \circ v_2 \circ ... \circ v_n$.

Now the Map function will only be applied on the insertions and deletions instead of the full input data. Every intermediate value $v \in V'$ which originates from a deleted input will be inverted by a function $*$ so that $v \circ v^* = e$, with e being the identity element w.r.t the function $\circ$.

We call a MapReduce job self-maintainable if it can be recomputed only by reading deltas (inserted and deleted values) and the result of the previous computation. Let $\Delta = \{v_{n+1}, ..., v_m\}$ be the inserted values and $\nabla = \{v_i, ..., v_k\}, 1 \leq i < k \leq n$ be the deleted values. Then the latest result a' can be computed as $a' = v_1 \circ ... \circ v_{i-1} \circ v_{k+1} \circ ... \circ v_m$. This full recomputation can be avoided by computing the new result a' by reusing the previous result a: $a' = a \circ v_{n+1} \circ ... \circ v_m \circ v_i^* \circ ... \circ v_k^*$.

In summary, we define the Abelian group $(V', \circ)$ as follows:

V' (Abelian group)		
$\circ$	$V' \times V' \rightarrow V'$	– aggregates two map-output values – associative and commutative
$*$	$V' \rightarrow V'$	– inverse element
e	$\emptyset \rightarrow V'$	– identity element

In the *WordCount* example we have:

$$Map : \mathbb{N} \times Text \rightarrow List < Text \times \mathbb{Z} >$$
$$:= \forall word : emit(word, 1)$$

Z (Abelian group)		
$\circ$	$\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} := +_{\mathbb{Z}}$	– addition of two numbers
$*$	$\mathbb{Z} \rightarrow \mathbb{Z} := \cdot(-1)_{\mathbb{Z}}$	– additive inverse
e	$\emptyset \rightarrow \mathbb{Z} := 0_{\mathbb{Z}}$	– zero

The input data for the map function now are the deltas. In case of a deletion, the Map-output values will be inverted by the function $*$. A Reduce function is not needed here. All the values for one specific key (i.e., normal Map-output values, inverted Map-output values and old results) are automatically aggregated to one final output value by the function $\circ$.

Algorithm 1 shows how a job that follows our modified programming model can be mapped to incremental MapReduce using Incremental Installation. The user-defined Map function *MAP'* is called on inserted and deleted data, the user-defined Reduce function *REDUCE'* (if defined) is called to handle the aggregated value. A generic Combine function aggregates values to decrease the amount of data which is transferred from the Map to the Reduce function. For Overwrite Installation, the Map functions has to distinguish not only between inserted and deleted data, but also between old results. These will simply be passed on the the Reduce function which aggregates the changes on it.

IV. Marimba Framework

Based on the ideas of incremental recomputations with Overwrite and Incremental Installation and the formalization seen in the previous chapter, we created a framework named Marimba[14]. It is based on Hadoop and can be used for implementing self-maintainable MapReduce jobs. We did not change the underlying Hadoop system, so Marimba can be run on top of any Hadoop version. A Marimba job basically consists of the two functions Map and Reduce which are exactly the same as in Hadoop. Furthermore the user has to define how to deserialize results from a former computation and how to invert and aggregate intermediate values.

A. How to Write a Marimba Job

Writing a Marimba job is very similar to writing a Hadoop job. You start implementing a *Tool* class and create a new *MarimbaJob*. This is used to configure input and output format, types and so on. Then you create a *Mapper* class as usual. One thing to keep in mind is to use a so-called

Algorithm 1 Mapping of user-defined Map and Reduce functions (*MAP', REDUCE'*) to incremental MapReduce with Increment Installation

```
1: function MAP(K key, V value)        ▷ Reads deltas only
2:     (k, v) = MAP'(key, value)       ▷ Call user-def. Map
3:     if key is inserted then
4:         emit(k, v)
5:     else if key is deleted then
6:         emit(k, v*)
7:     end if
8: end function
9: function COMBINE(K' key, List⟨V'⟩ values)
10:    a ← e
11:    for all v ∈ values do
12:        a ← a ∘ v
13:    end for
14:    emit(key, a)
15: end function
16: function REDUCE(K' key, List⟨V'⟩ values)
17:    a ← e
18:    for all v ∈ values do
19:        a ← a ∘ v
20:    end for
21:    emit(REDUCE'(key, {a}))  ▷ Call user-def. Reduce
22: end function
```

Abelian type as map-output-value class this time. This is a type you create individually for example by extending an existing *Writable* type from Hadoop and implementing the methods `aggregate`, `invert` and `neutral`.

Starting your Marimba job, the map function will be called for every inserted and deleted item. Map-output values of the user-defined Abelian type get inverted when the item is a deleted one. Eventually, in the Reduce phase all the values for one key along with the previous result get aggregated and form the output. The user can optionally define a *Reducer* when there are additional steps needed to write the final output, for example building a Put object to write the aggregated value into an HBase table.

When using the Overwrite strategy, one has to define a *Deserializer* which converts results from the previous computation into an Abelian object, so that it can later be aggregated upon the Map-output values.

B. Marimba Internals: Job Execution

Instead of the user-defined mapper, in the underlying Hadoop job a generic *MarimbaMapper* is set (see figure 3). This distinguishes between the different kinds of input data. Old results will be deserialized using the user-defined Deserializer. Inserted and deleted values will be translated into key-value pairs by the user-defined Mapper. Mind that the Map-output values are Abelian objects and will be inverted if the input was a deletion. Preserved values (these

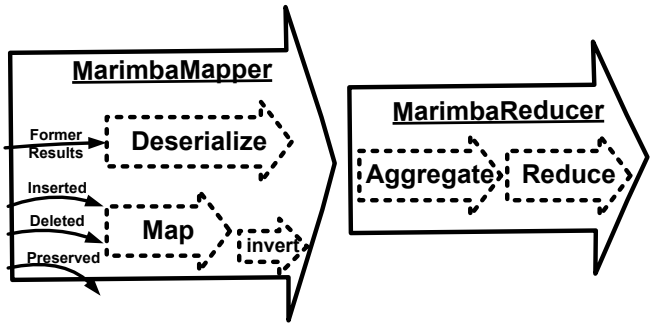

Figure 3. MarimbaMapper and -Reducer. Dotted arrows are UDFs.

are values which have already been processed in a former computation) can simply be skipped.

Depending on the strategy, a specific Reducer will be chosen, *MarimbaIncDecReducer* or *MarimbaOverwriteReducer*. In all cases, the (Abelian) intermediate values will be aggregated to one value. Next, the user-defined reducer will be invoked with both a key and a list, though the list only contains the one aggregated value, so that the user-defined reducer just needs to write the value and not to aggregate it. In case of Increment Installation the reducer's output will be converted into *Increment* objects which results in updates of column values in HBase. This Increment operation is similar to a Put operation but HBase has to read the current column value, increment it and write it back again. The HBase HRegionPartitioner distributes the intermediate data to the machines which are responsible for the given key. Thus, all reads and writes happen locally. Nevertheless, the Increment operation is still much slower than the Put operation because an HBase row has to be randomly accessed, whereas Put objects can be written in a sequential way. In the overwrite case the result gets written unless it is neutral. For example in WordCount neutral output values can occur if after a deletion the new number of occurrences of a word is zero now. The user can choose a *NeutralOutputStrategy* to define the behavior of the MarimbaOverwriteReducer:

- *PUT* (default case): The value will be written although it is neutral (in the WordCount example there would be zeros in the output).
- *IGNORE*: The value is not written. Overwriting an HBase table will make the old values stay in the table but one can distinguish between old and new values by their timestamps.
- *DELETE*: The value gets deleted in HBase.

The PUT strategy was chosen as the default case because it is the same behavior as in Increment Installation. There it is not possible to detect if a value gets neutral, it will always be written.

C. Realization Details

Marimba can be seen as a layer between Hadoop jobs and the Hadoop Framework. On the one hand, this means that an existing Hadoop job can be executed with Marimba with just a few modifications. On the other hand, every Marimba job will be mapped to a Hadoop job, so it can be executed through the Hadoop engine.

MarimbaJob vs. Hadoop's Job class: The MarimbaJob class is a subtype of Job and inherits methods to configure a MapReduce job (types, classes, formats, ...). One additional method *setStrategy* is used to choose the desired strategy. It can be set to *FULL_RECOMPUTATION*, *INCDEC* or *OVERWRITE*. A job that uses the first one is equivalent to a non-incremental Hadoop job. The latter strategy can only be used when a Deserializer class is defined. This class can be set like Mapper, Reducer and Combiner classes in Hadoop. Another new method, namely *setNeutralOutputStrategy* is used to set the behavior of overwrite jobs when the output is the identity element (*PUT*, *IGNORE*, *DELETE*).

Map-Input-Value Types: In Hadoop, the input-value type of the function Map depends on the input format. When using HDFS text files it is *Text*; when using HBase it is *Result*. In Marimba it is similar, but also the origin of each map-input value matters. So there are three Java interfaces *InsertedValue*, *DeletedValue* and *PreservedValue* which are used to flag the input-format types. As an example, we developed a TextWindowInputFormat which reads HDFS text files and treats each line as deleted, preserved or inserted depending on whether the line lies in a new or old window or their intersection. The begin and end of each window are given by the user before job execution. This input format was used to test the runtimes on Marimba jobs depending on the amount of changes in the input data (see chapter VI). The InputFormat produces *InsertedText* objects for each line which is in the new window and not in the old one and *DeletedText* objects for the opposite. Both are subtypes of Text, so the Map-input type is still *Text*. But as these classes implement the interfaces *InsertedValue* respectively *DeletedValue*, the MarimbaMapper can distinguish them.

When an HBase table is used as input, a change-data-capture input format can be used to deliver *InsertedResult* or *DeletedResult* objects to the Map function, e.g. based on their timestamps.

Additionally to the inserted, deleted and preserved data, the MarimbaMapper has to read the former results. For that we developed an *OverwriteInputFormat* that is the combination of the input format which is set by the user in the job configuration, and an input format to read the former results. In the latter one the values will be wrapped to *OverwriteResult* objects, so that the MarimbaMapper can forwand them to the user-defined Deserializer.

V. Example Jobs

A. Text Processing

In most cases, Hadoop code can simply be reused for writing a Marimba job. To make the non-incremental WordCount job incremental, the mapper can remain unchanged. The lines in the reducer where the sum of the values is calculated can be removed and the Map-output-value class has to be changed to *LongAbelian*, a new subtype of LongWritable with the methods aggregate (sum), invert (*=-1) and neutral (0). Figure 4 shows the WordCount algorithm using the Overwrite installation and DELETE as NeutralOutputStrategy.

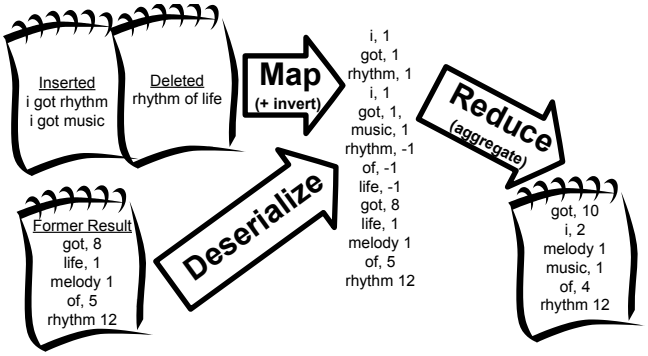

Figure 4. WordCount with Marimba - Overwrite Installation

We developed a more complex text-processing task which calculates probabilities for word sequences by generating bigrams. After analyzing large amounts of texts, the algorithm can tell you that the phrase "Wednesday night" is more probable than "Wednesday light". This can be used in voice-recognition systems or auto-completion tools. This *Bigrams* algorithm consists of two chained Marimba jobs. The first one is very similar to WordCount, but it doesn't count the individual words, but word pairs. So the output can look like this: ("I got", 1), ("got rhythm", 1). In the second job all successors of one word are collected together with their count value. Therefore a *TextLongMapAbelian* $:= List < Text \times \mathbb{Z} >$ is used.

$$Map : Text \times \mathbb{Z} \rightarrow Text \times List < Text \times \mathbb{Z} >$$

Map splits a word pair into two words, the first one forms the key, the second one forms together with the count value the value.

TextLongMapAbelian $:= List < (Text \times \mathbb{Z})^* >$	
$\circ$	$List < Text \times \mathbb{Z} > \times List < Text \times \mathbb{Z} > \rightarrow List < Text \times \mathbb{Z} > := \cup$ – union of successors; addition of their count values
$*$	$List < Text \times \mathbb{Z} > \rightarrow List < Text \times \mathbb{Z} > := \cdot(-1)_{\mathbb{Z}}$ – additive inverse of each count value
e	$\emptyset \rightarrow List < Text \times \mathbb{Z} > := \emptyset$ – empty list

The result is one word as a key and a set of words together with a number of occurrences which follow on that first word.

B. Graph Processing

MapReduce jobs can be used for many graph algorithms. The input data is either structured data, such as an adjacency list of a large graph or semi-structured data like a set of websites with HTML links in it. In the latter case, the map function needs to parse the websites to find all the links. A simple graph algorithm is reversing a web-link graph, i.e. generating a list which displays the incoming links from another website for every site.

In this algorithm, the *TextLongMapAbelian* is used as map-output value type again. It collects all the sites which link to a specific website. As this type can be reused from the Bigrams example, the operations $\circ$, $*$ and e are exactly the same. So when doing a recomputation of this algorithm, the map function produces $(link, \{(url, 1)\})$ for each link on an added website with the given URL. For deleted websites it is the same but the output gets inverted, so it will be $(link, \{(url, -1)\})$. After the map phase all values for one key will be aggregated to one list, which is the union of all map outputs and the previous result. If one URL occurs twice (for example once with a 5 from the old result and once with a -1 because of a link deletion), the numbers will be added (here: 4).

Another graph algorithm called *Friends of Friends* [8] reads an adjacency list of a social network. An edge in this network stands for: person A is a friend of person B. The algorithm produces another graph of indirect friends. So if A has got another friend C, then there will be an edge from B to C in the output graph. The map-output-value type is *TextLongMapAbelian* again.

Algorithm 2 Friends of Friends

```
1: function MAP(String personA, String friends)
2:     for all personB ∈ friends ≠ personC ∈ friends do
3:         emit(personB, {(personC, 1)})
4:     end for
5: end function
```

The Map function in algorithm 2 computes for each person the cross product of his or her friends (skipping the reflexive relationships). The rest of the algorithm is completed by the aggregate and invert methods of the *TextLongMapAbelian* and the Marimba framework. If some persons or friendships are added or deleted, the output will be recomputed in an incremental way. As the TextLongMapAbelian can be seen as a kind of multiset data structure, some friend-of-friend relationships can be in the map multiple times. The amount indicates the number of common friends between two persons. This information can for example be used for ordering people in a social-network search function. When searching for people, your own friends are displayed first and then friends of friends (ordered by the number of common friends).

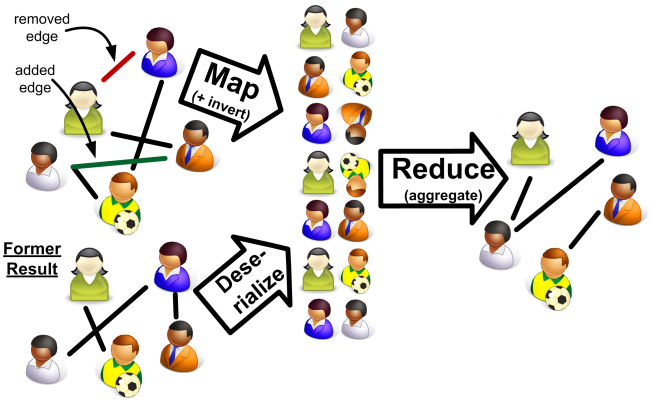

Figure 5. Friends-Of-Friends Algorithm

Figure 5 shows an example where one edge in a graph was added and one was removed. The added one produces two more friends-of-friends relationships, the removed one deletes two.

One iterative graph algorithm is PageRank [11]. The PageRank of a website A is determined by the PageRank of every other website that includes a link to A:

$$PR(A) = (1-d) \cdot \frac{1}{N} + d \cdot \sum_{B \in \bullet A} \frac{PR(B)}{|B \bullet|}$$

In this formular, N is the size of the web graph, $\bullet X$ are the incoming links to X, $X\bullet$ the outgoing links, and d a damping factor. In order to compute the PageRank for a given network, the value of every node is initialized with $\frac{1}{N}$ and then more exact PageRank values are computed iteratively with the upper formula until the changes of PageRank values are below a given threshold. The input and output is an adjacency list.

Algorithm 3 PageRank

```
1: function MAP(String url, (double pagerank, String[] links))
2:     for all link ∈ links do
3:         emit(link,(pagerank / links.length, ∅))
4:     end for
5:     emit(url, (0, links))          ▷ To reconstruct the graph
6: end function
```

The Map function in algorithm 3 reads a line of the adjacency list which consists of the URL of the website, the pagerank of the last iteration (initially $\frac{1}{N}$), and links to other websites. This function emits for every link its own pagerank devided by its outdegree. Furthermore the set of

adjacent nodes is emitted to keep up the graph structure. As map output a PageRankAbelian is being used, which is defined as follows:

PageRankAbelian (pagerank, links) $:= (\mathbb{R} \times TextLongMapAbelian)$	
$c = a \circ b$	$pagerank_c = pagerank_a + pagerank_b$
	$links_c = links_a \circ links_b$
$c = a*$	$pagerank_c = (-1) \cdot pagerank_a$
	$links_c = links_a*$
$c = e$	$pagerank_c = 0$
	$links_c = TextLongMapAbelian.e$

The map-output key class is a Text, thus all emitted PageRankAbelians for the same URL are aggregated by adding the pagerank fraction. This corresponds to the sum over all incoming links in the formula above.

The TextLongMapAbelian is then used to add new edges. For example, if there is a new link from website A to B and a removed link from website A to C, the PageRankAbelians (A, (0, {(B, 1)})) and (A, (0, {(C, 1)})) are emitted. The latter one will be inverted due to the deletion.

In the next iteration, the output of the map function is based on the new links on every website and the job is repeated until a convergence criterion is met.

C. Other Algorithms and Limitations

There are nearly unlimited MapReduce jobs which can be made incremental by using Marimba. Whenever values have to be aggregated, e.g., summed up, counted, or unified, it is easy to implement an Abelian type or use the predefined LongAbelian or TextLongMapAbelian. But when an algorithm needs aggregation functions like average, minimum, or maximum, it is hard to make it self-maintainable as they do not an Abelian group but a monoid.

When a MapReduce job should compute an average price for each product in a shop, the result cannot be reused for computing a new averages, when the base data changes. To fix this, the job has to store the count value in addition to the average.

Implementing this in Marimba leads to the following Abelian class:

AvgAbelian	
$c = a \circ b$	$count_c = count_a + count_b$
	$avg_c = \frac{avg_a \cdot count_a + avg_b \cdot count_b}{count_c}$
$c = a*$	$count_c = -1 \cdot count_a$, $avg_c = avg_a$
$c = e$	$count_c = avg_c = 0$

As you can see, there is a little overhead because the count value has to be stored. But at the same time it enables updates of the average in an incremental way.

If the task would be calculating the lowest price for every product, it is not possible to make it self-maintainable, unless all the possible prices would be remembered. Otherwise there would be no chance to detect the lowest price when the currently lowest price is being deleted. It can be optimized by storing just for example the lowest three values for each key by using a so-called *mincache* of size three (like in [4]). Then the job only fails if the mincache gets empty. Another possibility would be not to allow deletions, then computations of minima and maxima is not a problem.

Disallowing deletions makes it also possible to implement a shortest-path algorithm using breath-first search. There a Reduce function chooses the smallest distance to a start node and throws away the others.

VI. Evaluation

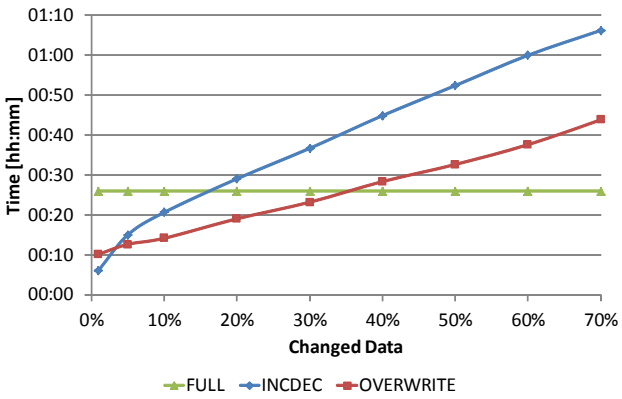

Figure 6. WordCount execution time with Marimba

First we implemented the algorithms WordCount, Reverse Web-Link Graph, Friends-Of-Friends and PageRank both natively in Hadoop and with Marimba and executed the jobs on a six-node cluster (Xeon Quadcore CPU at 2.53GHz, 4GB RAM, 1TB SATA-II disk, Gigabit Ethernet). In all tests the number of map and reduce tasks stays the same. The number of map tasks is determined by the number of splits of size 64MB (480 map tasks for an 30GB input file), the number of reduce tasks was 18. Performance tests showed that the execution times of Hadoop jobs correspond roughly to the times for the Full-Recomputation strategy in Marimba. Then we compared the three Marimba strategies depending on the percentage of input data which changed since the previous computation. Figure 6 shows that for WordCount on a 30GB text file the Incremental Installation is only faster than Overwrite when there are less than two percent of changes. At a change amount of 40% a full recomputation is faster than using the Overwrite strategy.

Figure 7 shows another test, namely the execution time of computing the Reverse Web-Link Graph on a 30GB web graph. As you can see, here the Overwrite strategy is always slower than a full recomputation. The reason for this is that reading the previous result is very time-consuming. Different from WordCount where the multiple count values are summed up to one single value, here the aggregation is a union operation which doesn't compact the data.

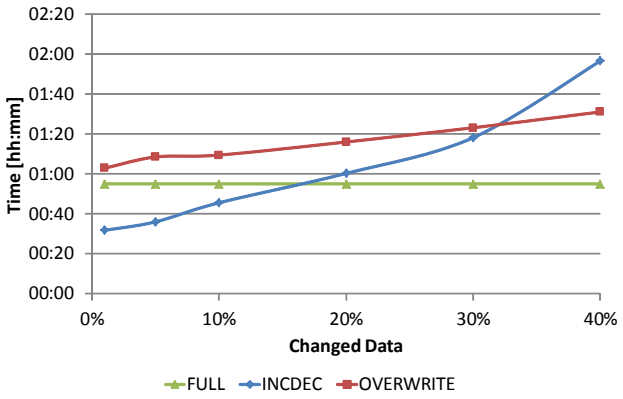

Figure 7. Reverse Web-Link Graph execution time with Marimba

VII. Conclusion

In this paper we presented Marimba, a framework for incremental MapReduce jobs based on Hadoop and Abelian groups. The idea behind Marimba is to use the Increment and Overwrite Installation strategies for self-maintainability that are well-known in the area in materialized views in relational databases. We defined the class of self-maintainable MapReduce jobs by modifying the MapReduce programming model. Nevertheless, the new programming model is very similar to standard MapReduce, so existing MapReduce jobs can be easily reused. One main variation is that the intermediate values have to form an Abelian group, so that they can be aggregated and inverted.

When developing a Marimba job, one has just to implement a Map and Reduce function as well as one Deserializer and one Abelian type. Instead of developing an own Abelian type, we saw that in many use cases two simple predefined types can be reused, namely LongAbelian and TextLongMapAbelian. We developed some text analysis and graph algorithms with Marimba and showed that executing jobs in an incremental way is much faster than a full recomputation. In some cases, the Overwrite strategy is faster than Increment Installation, or vice versa. We are currently searching for heuristics to decide which strategy is the best. This can either be done before a job is executed or even during the execution. Another goal is to implement more algorithms with Marimba like recommendation systems or spacial algorithms.

As another approach for accelerating incremental recomputations, intermediate results can be cached [3] (see chapter II). We wanted to compare the performance of this approach with Marimba but its code is not publicly available yet. In the future we plan to combine both techniques.

References

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

[2] Apache HBase. http://hbase.apache.org/.

[3] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[4] Miranda Chan. Incremental update to aggregated information for data warehouses over internet. In *In: 3rd ACM International Workshop on Data Warehousing and OLAP (DOLAP '00). McLean, Virginia, United States*, pages 57–64, 2000.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, pages 137–150, 2004.

[6] Yong Hu and Stefan Dessloch. Extracting deltas from column oriented nosql databases for different incremental applications and diverse data targets. In Barbara Catania, Giovanna Guerrini, and Jaroslav Pokorný, editors, *Advances in Databases and Information Systems*, volume 8133 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg, 2013.

[7] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental recomputations in mapreduce. In *CloudDB 2011*, October 2011.

[8] Steve Krenzel. MapReduce: Finding Friends. 2010. http://stevekrenzel.com/finding-friends-with-mapreduce.

[9] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 51–62, New York, NY, USA, 2010. ACM.

[10] Marimba framework. http://code.google.com/p/marimba-framework.

[11] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[12] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, 2010.

[13] Lucian Popa et al. DryadInc: Reusing work in large-scale computations. In *HotCloud*, 2009.

[14] Johannes Schildgen, Thomas Jörg, and Stefan Deßloch. Inkrementelle Neuberechnungen in MapReduce. *Datenbank-Spektrum*, 2012.

[15] Cairong Yan, Xin Yang, Ze Yu, Min Li, and Xiaolin Li. Incmr: Incremental data processing based on mapreduce. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 534–541. IEEE, 2012.

[16] Yanfeng Zhang and Shimin Chen. i 2 mapreduce: incremental iterative mapreduce. In *Proceedings of the 2nd International Workshop on Cloud Intelligence*, page 3. ACM, 2013.