2012 IEEE 12th International Conference on Data Mining

Tuple MapReduce: Beyond classic MapReduce

Pedro Ferrera, Ivan de Prado,
Eric Palacios
Datasalt Systems S.L.
Barcelona, SPAIN
pere,ivan,
epalacios@datasalt.com

Jose Luis Fernandez-Marquez
Giovanna Di Marzo Serugendo
University of Geneva, CUI
Geneva, SWITZERLAND
joseluis.fernandez,
Giovanna.DiMarzo@unige.ch

***Abstract*—This paper proposes Tuple MapReduce, a new foundational model extending MapReduce with the notion of tuples. Tuple MapReduce allows to bridge the gap between the low-level constructs provided by MapReduce and higher-level needs required by programmers, such as compound records, sorting or joins. This paper presents as well Pangool, an open-source framework implementing Tuple MapReduce. Pangool eases the design and implementation of applications based on MapReduce and increases their flexibility, still maintaining Hadoop's performance.**

***Keywords*-MapReduce; Hadoop; Big Data; Distributed systems; Scalability**

I. Introduction

During the last years, the amount of information handled within different fields (i.e. webs, sensor networks, logs, or social networks) has increased dramatically. Well established approaches, such as programming languages, centralised frameworks, or relational databases, do not cope well with current companies requirements arising from needs for higher-levels of scalability, adaptability, and fault-tolerance. These requirements are currently demanded by many companies and organisations that need to extract meaningful information from huge volumes of data and multiple sources. Even though many new technologies have been recently proposed for processing huge amounts of data, there is still room for new technologies that combine efficiency and easiness in solving real-world problems.

One of the major recent contributions, in the field of parallel and distributed computation, is MapReduce [1] - a programming model introduced to support distributed computing on large data sets on clusters of computers. MapReduce has been proposed in 2004 [2] and is intended to be an easy to use model, that even programmers without experience with parallel and distributed systems can apply. Indeed the MapReduce programming model hides parallelisation, fault-tolerance or load balancing details. Additionally, it has been shown that a large variety of problems can easily be expressed as a MapReduce computation.

MapReduce has been massively used by a wide variety of companies, institutions, and universities. This booming has been possible mainly thanks to an open-source implementation of MapReduce, Hadoop, in 2006 [3]. Since then, many higher-level tools built on top of Hadoop have been proposed and implemented (e.g. Pig [4], Hive [5], Cascading[1], FlumeJava [6]). Additionally, many companies have engaged in training programmers to extensively use them (e.g. Cloudera[2]). The massive investment in programmers training and in the development of these tools by the concerned companies would suggest some difficulties in the use of MapReduce for real-world problems and actually involves a sharp learning curve. Specifically, we have noticed that most common design patterns, such as compound records, sort or join, useful when developing MapReduce applications, are not well covered by MapReduce fundamentals. Therefore, derived with direct experience with our customers, we found it necessary to formulate a new theoretical model for batch-oriented distributed computations. Such a model needs: to be as flexible as MapReduce; to allow easier direct use; and to let higher-level abstractions to be built on top of it in a straightforward way.

In this paper, we propose a new foundational model of MapReduce - Tuple MapReduce - which targets those applications that perform batch-oriented distributed computation. Moreover, an implementation of the proposed model - Pangool - is provided and compared with existing implementations of MapReduce. As a result of this work, we suggest that Tuple MapReduce can be used as a direct, better-suited replacement of the MapReduce model in current implementations without the need of modifying key system fundamentals.

This paper is structured as follows: Next section provides a brief overview of MapReduce, and existing implementations in the literature. In Section III, we identify current problems that arise when using MapReduce. In order to overcome the problems mentioned previously, a new theoretical model called Tuple MapReduce is proposed in Section IV. In Section V, we propose and analyse a Tuple MapReduce implementation called Pangool, which has been built on

[1] http://www.cascading.org
[2] http://www.cloudera.com/

1550-4786/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDM.2012.141

top of Hadoop. Finally, conclusions and future work are explained in Section VI.

II. Related Work

In this paper we concentrate on MapReduce, a programming model that was formalized by Google in 2004 [2] and on Hadoop, its associated de-facto open-source implementation. We briefly describe the main idea behind MapReduce and its Hadoop implementation, then we review a series of abstraction and tools built on top of Hadoop, and mention an alternative model to MapReduce.

MapReduce can be used for processing information in a distributed, horizontally-scalable fault-tolerant way. Such tasks are often executed as a batch process that converts a set of input data files into another set of output files whose format and features might have mutated in a deterministic way. Batch computation allows for simpler applications to be built by implementing idempotent processing data flows. It is commonly used nowadays in a wide range of fields: data mining [7], machine learning [8], business intelligence [9], bioinformatics [10], and others.

Applications using MapReduce often implement a *Map function*, which transforms the input data into an intermediate dataset made up by *key/value* records, and a *Reduce* function that performs an arbitrary aggregation operation over all registers that belong to the same key. Data transformation happens commonly by writing the result of the reduced aggregation into the output files. Despite the fact that MapReduce applications are successfully being used today for many real-world scenarios, this simple foundational model is not intuitive enough to easily develop real-world applications with it.

Hadoop is a programming model and software framework allowing to process data following MapReduce concepts. Many abstractions and tools have arisen on top of MapReduce. An early abstraction is Google's Sawzall [11]. This abstraction allows for easier MapReduce development by omitting the Reduce part in certain, common tasks. A Sawzall script runs within the Map phase of a MapReduce and "emits" values to tables. Then the Reduce phase (which the script writer does not have to be concerned about) aggregates the tables from multiple runs into a single set of tables. Another example of such abstractions is FlumeJava, also proposed by Google [12]. FlumeJava allows the user to define and manipulate "parallel collections". These collections mutate by applying available operations in a chained fashion. In the definition of FlumeJava, mechanisms for deferred evaluation and optimisation are presented, leading to optimised MapReduce pipelines that otherwise would be hard to construct by hand. A private implementation of FlumeJava is used by hundreds of pipeline developers within Google. There are recent open-source projects that implement FlumeJava although none of them are mainstream[3].

One of the first and most notable higher-level, open-source tools built on top of the mainstream MapReduce implementation (i.e. Hadoop) has been Pig [4], which offers SQL-style high-level data manipulation constructs that can be assembled by the user in order to define a dataflow that is compiled down to a variable number of MapReduce steps. Pig is currently a mature open-source higher-level tool on top of Hadoop and implements several optimizations, allowing the user to abstract from MapReduce and the performance tweaks needed for efficiently executing MapReduce jobs in Hadoop.

Also worth mentioning is Hive [5], which implements a SQL-like Domain Specific Language (DSL) that allows the user to execute SQL queries against data stored in Hadoop filesystem. These SQL queries are then translated to a variable length MapReduce job chain that is further executed into Hadoop. Hive approach is specially convenient for developers approaching MapReduce from the relational databases world.

Jaql [13] is a declarative scripting language for analyzing large semistructured datasets built on top of MapReduce. With a data model based on JSON, Jaql offers a high level abstraction for common operations (e.g., join) that are compiled into a sequence of MapReduce jobs.

Other abstractions exist, for instance, Cascading[4], which is a Java-based API that exposes to the user an easily extendable set of primitives and operations from which complex dataflows can be defined and further executed into Hadoop. These primitives add an abstraction layer on top of MapReduce that remarkably simplify Hadoop application development, job creation and job executing.

The existence of all these tools, and the fact that they are popular sustain the idea that MapReduce is a too low-level paradigm that does not map well to real-world problems. Depending on the use case, some abstractions may fit better than others. Each of them have their own particularities and there is also literature on the performance discrepancies between each of them [14], [15], [16].

While some of these benchmarks may be more rigorous than others, it is expected that higher-level tools built on top of MapReduce perform poorly compared to hand-optimized MapReduce. Even though there exists many options, some people still use MapReduce directly. The reasons may vary, but they are often related to performance concerns or convenience. For instance, Java programmers may find it more convenient to directly use the Java MapReduce API of Hadoop rather than building a system that interconnects a DSL to pieces of custom business logic written in Java. In spite of these cases, MapReduce is still the foundational model from which any other parallel computation model on top of MapReduce must be written.

MapReduce limitations for dealing with relational data

[3]https://github.com/cloudera/crunch

[4]http://www.cascading.org

have been studied by [17]. The authors illustrate MapReduce lack of direct support for processing multiple related heterogeneous datasets and perform relational operations like joins. These authors propose to add a new phase to MapReduce, called *Merge*, in order to overcome these deficiencies. This implies changes in the distributed architecture of MapReduce.

Although we share the same concerns about MapReduce weaknesses, the solution we propose in this paper beats these problems but without the need of a change in MapReduce distributed architecture and in a simpler way.

III. The problems of MapReduce

Although MapReduce has been shown useful in facilitating the parallel implementation of many problems, there are some many common tasks, recurring when developing distributed applications, that are not well covered by MapReduce. Indeed, we have noticed that most of the common design patterns that arise with typical Big Data applications, although simple (e.g. joins, secondary sorting), are not directly provided by MapReduce, or, even worst, they are complex to implement with it. In this section, we analyse and summarise the main existing problems arising when using MapReduce for solving common distributed computation problems:

1) **Compound records:** In real-world problems, data is not only made up of single fields records. But MapReduce abstraction forces to split the records in a key/value pair. MapReduce programs, processing multi-field records (e.g. classes in object oriented programming terminology), have to deal with the additional complexity of either concatenating or splitting their fields in order to compose the key and the value that are required by the MapReduce abstraction. To overcome this complexity, it has become common practice to create custom data-types, whose scope and usefulness is confined to a single MapReduce job. Frameworks for the automatic generation of data-types have then been developed, such as Thrift [18] and Protocol Buffers [19], but this only alleviates the problem partially.
2) **Sorting:** There is no inherent sorting in the MapReduce abstraction. The model specifies the way in which records need to be grouped by the implementation, but does not specify the way in which the records need to be ordered within a group. It is often desirable that records of a reduce group follow a certain ordering; an example of such a need is the calculation of moving averages where records are sorted by a time variable. This ordering is often referred to as "secondary sort" within the scope of Hadoop programs and it is widely accepted as a hard-to-implement, advanced pattern in this community.
3) **Joins:** Joining multiple related heterogeneous datasets is a quite common need in parallel data processing, however it is not something that can be directly derived from the MapReduce abstraction. MapReduce implementations such as Hadoop offer the possibility of implementing joins as a higher-level operation on top of MapReduce; however, a significant amount of work is needed for efficiently implementing a join operation - problems such as 1) and 2) are strongly related to this.

In this paper, we propose a new theoretical model called Tuple MapReduce aimed at overcoming these limitations. We state that Tuple MapReduce can even be implemented on top of MapReduce so no key changes in the distributed architecture are needed in current MapReduce implementations to support it. Additionally we present an open-source Java implementation of Tuple MapReduce on top of Hadoop called Pangool[5], that is compared against well-known approaches.

IV. Tuple MapReduce

In order to overcome the common problems that arise when using MapReduce, we introduce Tuple MapReduce. Tuple MapReduce is a theoretical model that extends MapReduce to improve parallel data processing tasks using *compound-records*, optional in-reduce *ordering*, or inter-source datatype *joins*. In this section, we explain the foundational model of Tuple MapReduce, and show how it overcomes the existing limitations reported above.

A. Original MapReduce

The original MapReduce paper proposes units of execution named jobs. Each job processes an input file and generates an output file. Each MapReduce job is composed of two consecutive steps: the map phase and the reduce phase. The developer's unique responsibility is developing two functions: the map function and the reduce function. The rest of the process is done by the MapReduce implementation. The map phase converts each input key/value pair into zero, one or more key/value pairs by applying the provided map function. There is exactly one call to the map function for each input pair. The set of pairs generated by the application of the map function to every single input pair is the intermediate dataset. At the reduce phase MapReduce makes a partition of the intermediate dataset. Each partition is formed by all the pairs that share the same key. This is the starting point of the reduce phase. At this point, exactly one call to the reduce function is done per each individual partition. The reduce function receives as input a list of key/value pairs, all of them sharing the same key, and converts them into zero, one or more key/value pairs.

[5]http://pangool.net

The set of pairs generated by the application of the reduce function to every partition is the final output.

Equations 1(a) and 1(b) summarize the behaviour map and reduce function must follow. Remember that the particular implementation of these functions must be provided by the developer for each individual job.

$$map \quad (k_1, v_1) \rightarrow list(k_2, v_2) \tag{1a}$$

$$reduce \quad (k_2, list(v_2)) \rightarrow list(k_3, v_3) \tag{1b}$$

The map function receives a pair of key/value of types (k_1, v_1) that it must convert into a list of other data pairs $list(k_2, v_2)$. The $reduce$ function receives a key data type k_2 and a list of values $list(v_2)$, that it must convert into a list of pairs $list(k_3, v_3)$. Subscripts refer to data types. In other words:

- All key and value items received by the map function have the same data types k_1 and v_1 respectively.
- All key and value items emitted by the map function have the same data types k_2 and v_2 respectively.
- Reduce input key and values data types are k_2 and v_2 respectively. That is, map output and reduce input must share the same types.
- All key and value pairs emitted by the reducer have the same types k_3 and v_3 respectively.

The summary of the process is the following: the map function emits pairs. Those pairs are grouped by their key, thus those pairs sharing the same key belong to the same group. For each group, a reduce function is applied. The pairs emitted by applying the reduce function to every single group constitute the final output. All this process is executed transparently in a distributed way by the MapReduce implementation.

B. Tuple MapReduce

The fundamental idea introduced by Tuple MapReduce is the usage of *tuples* within its formalisation. Tuples have been widely used in higher-level abstractions on top of MapReduce (e.g. FlumeJava [6], Pig [4], Cascading[6]). Nonetheless, the innovation of Tuple MapReduce lies in revisiting the foundational theoretical MapReduce model by using as a basis a tuple-based mathematical model. Namely, we substitute key/value records as they are used in traditional MapReduce by a raw n-sized tuple. The user of a Tuple MapReduce implementation, instead of emitting key and value data-types in the map stage, emits a tuple. For executing a particular MapReduce job, the Tuple MapReduce implementation has to be provided with an additional *group-by* clause declaring the fields on which tuples must be grouped on before reaching the reduce stage. In other words, the group-by clause specifies which fields tuples emitted by the map function should be grouped on. By eliminating the distinction between key and value and by allowing the user to specify one or more fields in the group-by clause, the underlying Tuple MapReduce implementation easily overcomes the difficulties imposed by the original MapReduce constraint that forces to use just pairs of values.

Equations 2(a) and 2(b) summarize the new map and reduce functions contract and the data types for each tuple field.

$$map \quad (i_1, \ldots, i_m) \rightarrow list((v_1, \ldots, v_n)) \tag{2a}$$

$$reduce \quad ((v_1, \ldots, v_g), list((v_1, \ldots, v_n))_{sortedBy(v_1,\ldots,v_s)}) \rightarrow list((k_1, \ldots, k_l)) \quad g \leq s \leq n \tag{2b}$$

In Tuple MapReduce, the map function processes a tuple as input of types $(i_1, \ldots, i_m)$ and emits a list of other tuples as output $list((v_1, \ldots, v_n))$. The output tuples are all made up of n fields out of which the first g fields are used to group-by and the first s fields are used to sort the fields (see Figure 1). For clarity, group-by fields and sort-by fields are defined to be a prefix of the emitted tuple, being group-by a subset of the sort-by ones. It is important to note that the underlying implementations are free to relax this restriction, but for simplifying the explanation we are going to consider that fields order in tuples is important. As for MapReduce, subscripts refer to data types.

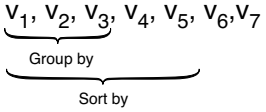

Figure 1. Group-by and Sort-by fields

The $reduce$ function then takes as input a tuple of size g, $(v_1, \ldots, v_g)$ and a list of tuples $list((v_1, \ldots, v_n))$. All tuples in the provided list share the same prefix which correspond exactly to the provided tuple of size g. In other words, each call to the reduce function is responsible to process a group of tuples that shares the same first g fields. Additionally, tuples in the input list are sorted by theirs prefixes of size s. The responsibility of the reduce function is to emit a list of tuples $list((k_1, \ldots, k_l))$ as result.

Therefore, the developer is responsible for providing:

- The map function implementation
- The $reduce$ function implementation
- g, s with $g \leq s \leq n$

Let's say that tuple A has the same schema as tuple B if A and B have the same number of fields n and the type of field i of tuple A is the same as the type of field i of the tuple B for every i in $[1 \ldots n]$. The main schemas relations in Tuple MapReduce are the following:

[6]http://www.cascading.org

- All map input tuples must share the same schema
- All map output tuples and reduce input tuples in the list must share the same schema
- All reduce output tuples must share the same schema

By using tuples as a foundation we enable underlying implementations to easily implement intra-reduce sorting (the so called "secondary sort" in Hadoop terminology). The user of a Tuple MapReduce implementation may specify an optional sorting by specifying which fields of the tuple will be used for sorting $(v_1, v_2, \ldots, v_s)$. Sorting by more fields than those which are used for group-by $(v_1, v_2, \ldots, v_g)$ will naturally produce an intra-reduce sorting. Intra-reduce sorting is important because the list of tuples received as input in the reducer can be so long that it does not fit on memory. Since this is not scalable, the input list is often provided by the implementations as a stream of tuples. In that context, the order in which tuples are retrieved can be crucial if we want some problems such as calculating moving averages for very long time series to be solved in a scalable way.

As MapReduce, Tuple MapReduce supports the use of a $combiner$ function in order to reduce the amount of data sent by network between mappers and reducers.

C. Example: cumulative visits

As mentioned above, many real-world problems are difficult to realize in traditional MapReduce. An example of such a problem is having a register of daily unique visits for each URL in the form of compound records with fields (url, date, visits) from which we want to calculate the cumulative number of visits up to each single date.

For example, if we have the following input:

```
yes.com, 2012-03-24, 2
no.com,  2012-04-24, 4
yes.com, 2012-03-23, 1
no.com,  2012-04-23, 3
no.com,  2012-04-25, 5
```

Then we should obtain the following output:

```
no.com,  2012-04-23, 3
no.com,  2012-04-24, 7
no.com,  2012-04-25, 12
yes.com, 2012-03-23, 1
yes.com, 2012-03-24, 3
```

The pseudo-code in Algorithm 1 shows the map and reduce function needed for calculating the cumulative visits using Tuple MapReduce.

The map function is the identity function: it just emits the input tuple. The reduce function receives groups of tuples with the same URL sorted by date, and keeps a variable for calculating the cumulative counting. Group-by is set to "url" and sort-by is set to "url" and "date" in order to receive the proper groups and with the proper sorting at the reduce function.

Algorithm 1: Cumulative visits

```
map(tuple):
    emit(tuple)

reduce(groupTuple, tuples):
    count = 0
    foreach tuple in tuples do
        count += tuple.get("visits")
        emit(Tuple(tuple.get("url"),
          tuple.get("date"), count))

groupBy("url")
sortBy("url", "date")
```

Tuple MapReduce will call 5 times the map function (one per each input tuple), and twice the reduce function (one per each group: $no.com$ and $yes.com$)

For performing the above problem we used some of Tuple MapReduce key characteristics: on one side, we used the ability of working with compound records (tuples) and on the other side we used the possibility of sorting the intermediate outputs by more fields than those that are needed for grouping.

D. Joins with Tuple MapReduce

Finally, we incorporate to the foundational model the possibility of specifying heterogeneous data source joins. The user of a Tuple MapReduce implementation needs only to specify the list of sources - together with a data source id - and the fields in common among those data source tuples in order to combine them into a single job. The user will then receive tuples from each of the data sources in the reduce groups, sorted by their data source id. This predictive, intra-reduce sorting enables any type of relational join to be implemented on top of a Tuple MapReduce join.

Equations 3(a) to 3(c) summarize the functions contract and the data types for each tuple field in a two data sources join. The map_A and map_B functions are map functions of Tuple MapReduce that map tuples from two different sources A and B. The $reduce$ function in the case of multiple data sources, receives an input tuple of types $(v_1, v_2, \ldots, v_g)$ representing the common fields in both data sources on which we want to group-by, and two lists of tuples, the first with tuples emitted by map_A, and the second with tuples emitted by map_B. The output of the $reduce$ function is a list of tuples.

$$map_A(i_1, i_2, \ldots, i_o) \rightarrow list((v_1, v_2, \ldots, v_n)_A) \quad (3a)$$

$$map_B(j_1, j_2, \ldots, j_p) \rightarrow list((v_1, v_2, \ldots, v_m)_B) \quad (3b)$$

$$reduce(v_1, v_2, \ldots, v_g), list((v_1, v_2, \ldots, v_n)_A), list((v_1, v_2, \ldots, v_m)_B) \rightarrow list((k_1, k_2, \ldots, k_q)) \quad (3c)$$

$$(g \leq n)(g \leq m)$$

It is easy to extend this abstraction to support multiple input sources, not just two.

For simplicity reasons, the above model only contemplates the possibility to perform intra-reduce sorting by source id A or B. But it would be possible to set a different intra-reduce sorting, including using source id at different positions. For example, it would be possible to perform a sort-by (v_1, v_2, id) with a group-by clause of (v_1).

E. Join example: clients and payments

Algorithm 2 shows an example of inner join between two datasets: clients $(clientId, clientName)$ and payments $(paymentId, clientId, amount)$. For example, if we have client records like:

```
1, luis
2, pedro
```

And payment records like:

```
1, 1, 10
2, 1, 20
3, 3, 25
```

Then we want to obtain the following output:

```
luis, 10
luis, 20
```

Because we want to join by $clientId$, then we group by $clientId$. Each client can have several payments, but each payment belongs to just one client. In other words, we have a 1-to-n relation between clients and payments. Because of that, we are sure that in each reduce call we will receive as input at most 1 client, and at least one tuple (reduce groups with zero tuples are impossible by definition). By assigning the $sourceId$ 0 to clients and 1 to payments and configuring an inner-sorting by $sourceId$ we ensure that clients will be processed before payments in each reducer call. This is very convenient to reduce the memory consumption, because otherwise we would have to keep every payment in memory until we retrieve the client tuple, then perform the join with the in memory payments and then continue with the rest of payments already not consumed (remember that reduce input tuples are provided as a stream). This would be not scalable when there are a lot of payments per client so that they don't fit in memory. Tuple MapReduce allows for reduce-side inner, left, right and outer joins without memory consumption.

Algorithm 2: Client-payments inner join example

```
map(client):
    client.setSourceId(0)
    emit(client)

map(payment):
    payment.setSourceId(1)
    emit(payment)

reduce(groupTuple, tuples):
    client = first (tuples)
    if client.getSourceId() != 0
      return
    foreach payment in rest(tuples) do
      emit(Tuple(client.get("clientName"),
        payment.get("amount")))

groupBy("clientId")
sortBy("clientId", sourceId)
```

F. Rollup

Dealing with tuples adds some opportunities for providing a richer API. That is the case of the $rollup$ feature of Tuple MapReduce, which is an advanced feature that can be derived from the Tuple MapReduce formalisation. By leveraging secondary sorting, it allows to perform computations at different levels of aggregation within the same Tuple MapReduce job. In this section we will explain this in more detail.

Lets first see the case of an example involving tweets data. Imagine we have a dataset containing $(hashtag, location, date, count)$ and we want to calculate the total number of tweets belonging to a particular hashtag per location and per location and date by aggregating the partial counts. For example, with the following data:

```
#news, Texas, 2012-03-23, 1
#news, Arizona, 2012-03-23, 2
#news, Arizona, 2012-03-24, 4
#news, Texas, 2012-03-23, 5
#news, Arizona, 2012-03-24, 3
```

We would like to obtain the totals per hashtag per location (that is, how many tweets have occurred in a certain location having a certain hashtag):

```
#news, Arizona, total, 9
#news, Texas, total, 6
```

And the totals per hashtag per location and per date (that is, how many tweets have occurred in a certain location having a certain hashtag within a specific date):

```
#news, Arizona, 2012-03-23, 2
#news, Arizona, 2012-03-24, 7
#news, Texas, 2012-03-23, 6
```

The straightforward way of doing it with Tuple MapReduce is creating two jobs:

1) The first one grouping by $hashtag$ and $location$
2) The second one grouping by $hashtag$, $location$ and $date$

Each of those jobs just performs the aggregation over the $count$ field and emits the resultant tuple. Although this approach is simple, it is not very efficient as we have to launch two jobs when really only one is needed. By leveraging secondary sorting, both aggregations can be performed in a job where we only group by $hashtag$ and $location$. How is that possible? The trick consists in sorting each group by $date$ and maintaining a state variable with the count for each $date$, detecting consecutive $date$ changes when they occur and thus resetting the counter.

The idea is then to create a single job that:

- groups by $hashtag$, $location$
- sorts by $hashtag$, $location$, $date$

As we said, the reduce function should keep a counter for the location total count, and a partial counter used for date counting. As tuples come sorted by date it is easy to detect changes when a date has changed with respect to the last tuple. When detected, the partial count for the date is emitted and the partial counter is cleaned up.

The proposed approach only needs one job, which is more efficient, but at the cost of messing up the code. In order to better address these cases, we propose an alternative API for Tuple MapReduce for supporting rollup.

The developer using rollup must provide a $rollup-from$ clause in addition to $group-by$ and $sort-by$ clauses. When using rollup, the developer must group by the narrowest possible group. Every aggregation occurring between $rollup-from$ and $group-by$ will be considered for rollup. The additional constraint is that all rollup-from clause fields must be also present in the group-by clause. The developer provides the functions $onOpen(field, tuple)$ and $onClose(field, tuple)$. These functions will be called by the implementation on the presence of an opening or closing of every possible group.

The pseudo code presented in Algorithm 3 shows the solution with the proposed rollup API for the counting of tweets hashtags. There is a global counter $locationCount$ used for counting within a location. Each time a location group is closed, the aggregated location count is emitted and the $locationCount$ variable is reset. The reduce function updates the $locationCount$ and the $dateCount$ with the counts from each tuple, and is responsible for emitting the counts for dates.

The rollup API simplifies the implementation of efficient multilevel aggregations by the automatic detection of group changes.

Algorithm 3: Rollup example

```
locationCount = 0
map(tuple):
    emit(tuple)

onOpen(field, tuple):


onClose(field, tuple):
    if field == "location":
        locationCount += tuple.get("count")
    emit(Tuple(tuple.get("hashtag"),
        tuple.get("location"),
        "total",
        locationCount))
        locationCount = 0

reduce(groupTuple, tuples):
    dateCount = 0
    foreach tuple in tuples do
        locationCount += tuple.get("count")
        dateCount += tuple.get("count")
    emit(Tuple(groupTuple.get("hashtag"),
        groupTuple.get("location"),
        groupTuple.get("date"),
        dateCount))

groupBy("hashtag", "location", "date")
sortBy("hashtag", "location", "date")
rollupFrom("hashtag")
```

G. Tuple Mapreduce as a generalization of classic MapReduce

Tuple MapReduce as discussed in this section can be seen as a generalisation of the classic MapReduce. Indeed, the MapReduce formalisation is equivalent to a Tuple MapReduce formalisation with tuples constrained to be of size two, group-by being the first field (the so called key in MapReduce) and an empty set sorting with no inter-source joining specification. Because MapReduce is contained in Tuple MapReduce, we observe that the latter is a wider, more general model for parallel data processing.

Tuple MapReduce comes with an additional advantage. Implementing Tuple MapReduce in existing MapReduce systems does not involve substantial changes in the distributed architecture. For implementing Tuple MapReduce on top of a classic MapReduce implementation, it is usually sufficient to support custom serialization mechanisms, custom partitioner and low-level sort and group comparators of the existing MapReduce implementation. A proof of

	Hadoop's code lines	Pangool's code lines	% Reduction
Secondary sort	256	139	45.7%
URL Resolution (join)	323	158	51%

Table I
LINES OF CODE REDUCTION

that is Pangool, an open source implementation of Tuple MapReduce on top of Hadoop.

V. PANGOOL: TUPLE MAPREDUCE FOR HADOOP

In March, 2012, we released an open-source Java implementation of Tuple MapReduce called Pangool[7] ready to be used in production. We have developed Pangool on top of Hadoop without modifying Hadoop source code. Pangool is a Java library. Developers just need to add it to their projects in order to start developing with Tuple MapReduce.

Pangool implements Tuple MapReduce paradigm by allowing the user to specify an intermediate tuple schema that will be followed by the map output and the reduce input. This schema defines the tuple data types and fields. Pangool allows to specify group-by and sort-by clauses per each job. Additionally, several data sources can be added by employing different intermediate tuple schemas and specifying a common set of fields that will be used for joining these data sources. Users of Pangool define a map and reduce function similar to how they would do in Hadoop, but wrapping their data into tuple objects. In addition, Pangool offers several enhancements to the standard Hadoop API: configuration by instances and native multiple inputs / outputs. Pangool's user guide[8] is the reference for learning how to use it. Additionally, many examples showing how to use Pangool are provided on-line[9].

Table I shows the differences in number of lines when implementing two different tasks in either Hadoop or Pangool. The first task involves a secondary sort, while the second task involves joining two datasets. Both use compound records. These tasks can be seen in Pangool's examples[10] and benchmark[11] projects. In both examples, we notice a reduction of about 50% in lines of code when using Pangool as opposed to when using Hadoop.

Figure 2 shows a benchmark comparison between Pangool, Hadoop and two other Java-based higher-level APIs on top of Pangool (Crunch 0.2.0[12], Cascading 1.2.5[13]). In the graphic we show the time in seconds that it takes for each implementation to perform a simple MapReduce parallel task. This task is the "Secondary sort example" whose code lines were compared in Table I. It involves grouping compound records of four fields grouping-by two of them and performing secondary sort on a third field.

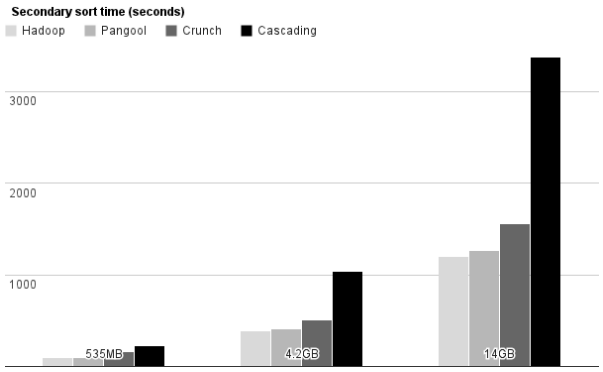

Figure 2. Secondary sort time (seconds)

Figure 3 shows the relative performance between different implementations of a reduce-join on two datasets of URLs. We decided to benchmark Pangool to other higher-level APIs in spite of the fact that Pangool is still a low-level Tuple MapReduce API, mainly for showing that the associated implementation of Tuple MapReduce should still be powerful enough to perform comparably to an associated implementation of MapReduce (Hadoop). In these graphics we can see that Pangool's performance is in the order of 5% to 8% worse than Hadoop, which we think is remarkably good considering that other higher-level APIs perform 50% to (sometimes) 200% worse. We also think that Pangool's performance is quite close to the minimum penalty that any API on top of Hadoop would have. In any case, this overhead would be eliminated with a native Tuple MapReduce implementation for Hadoop, which we believe would be very convenient as Tuple MapReduce has shown to keep the advantages of MapReduce but without some of its disadvantages.

The benchmark together with the associated code for reproducing it is available at the following location[14]. The full benchmark consists of a comparison of three tasks, one of them well-known (the word count task).

Pangool is currently being used with success in Datasalt[15] projects, simplifying the development and making code easier to understand, while still keeping efficiency.

So far we have been using it for building inverted indexes and performing statistical analysis on banking data. On the other hand we have developed some use case examples like

[7] http://pangool.net
[8] http://pangool.net/userguide/schemas.html
[9] https://github.com/datasalt/pangool/tree/master/examples/src/main/java/com/datasalt/pangool/examples
[10] https://github.com/datasalt/pangool/tree/master/examples
[11] https://github.com/datasalt/pangool-benchmark
[12] https://github.com/cloudera/
[13] http://www.cascading.org/
[14] http://pangool.net/benchmark.html
[15] http://www.datasalt.com

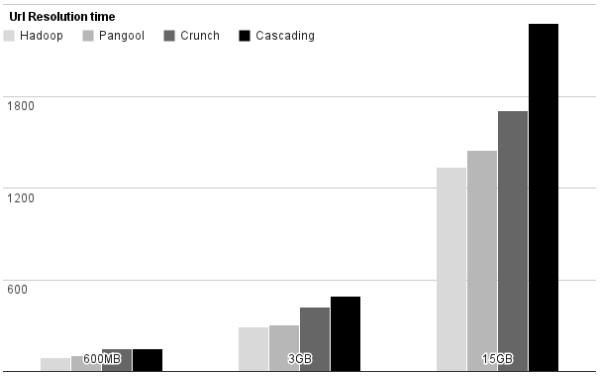

Figure 3. URL resolution time (seconds)

writing a scalable Naive Bayes text classifier[16]. This example illustrates that scalable MapReduce-based data mining algorithm implementations such as the ones in Mahout project[17] would benefit from being implemented on top of Pangool. Any computation that involves the usage of MapReduce can be implemented easily and more conveniently using Pangool because, as we have stated, it will usually involve the usage of compound records, inter-group sorting or joins. A popular example of such a calculation is PageRank. Implementations of PageRank involve the serialization of compound records containing multiple authorities and a score metric. It is also needed to join data from previous steps in order to take into account, for example, dangling nodes.

VI. Conclusions and Future Work

Our theoretical effort in formulating Tuple MapReduce has shown us that there is a gap between MapReduce, the nowadays de-facto foundational model for batch-oriented parallel computation - or its associated mainstream implementation (Hadoop) - and mainstream higher-level tools commonly used for attacking real-world problems such as Pig or Hive. MapReduce key/value constraint has been shown too strict, making it difficult to implement simple and common tasks such as joins. Higher-level tools have abstracted the user from MapReduce at the cost of less flexibility and more performance penalty; however, there is no abstraction in the middle that retains the best of both worlds: simplicity, easy-to-use, flexibility and performance.

We have shown that Tuple MapReduce keeps all the advantages of MapReduce. Indeed, as shown, MapReduce is a particular case of Tuple MapReduce. Besides, Tuple MapReduce has a lot of advantages over MapReduce, such as, compound records, direct support for joins, and intra-reduce sorting. We have implemented Pangool for showing that there can be an implementation of Tuple MapReduce that performs comparably to an associated implementation of MapReduce (Hadoop) while simplifying many common tasks that are difficult and tedious to implement in MapReduce. Pangool also proves that key changes in the distributed architecture of MapReduce are not needed for implementing Tuple MapReduce. For all these reasons, we believe that Tuple MapReduce should be considered as a strong candidate abstraction to replace MapReduce as the de-facto foundational model for batch-oriented parallel computation.

Future work includes the development of new abstractions that simplify the task of chaining MapReduce jobs in a flow. We believe there is room for improvement in this field. Concretely, we plan to build an abstraction for easing running flows of many parallel jobs, incorporating ideas from tools such as Azkaban[18] and Cascading.

[16]http://www.datasalt.com/2012/04/building-a-parallel-text-classifier-in-hadoop-with-pangool/

[17]http://mahout.apache.org/

[18]http://sna-projects.com/azkaban/

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] J. Dean. and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. USENIX Association: ACM, 2004, pp. 10–10.

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

[4] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1414–1425, Aug. 2009.

[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.

[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: easy, efficient data-parallel pipelines," *SIGPLAN Not.*, vol. 45, no. 6, pp. 363–375, Jun. 2010.

[7] R. Grossman and Y. Gu, "Data mining using high performance data clouds: experimental studies using sector and sphere," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '08. New York, NY, USA: ACM, 2008, pp. 920–927.

[8] 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*, B. Schölkopf, J. C. Platt, and T. Hoffman, Eds. MIT Press, 2006, pp. 281–288.

[9] U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson, "Data integration flows for business intelligence," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '09. New York, NY, USA: ACM, 2009, pp. 1–11.

[10] R. Taylor, "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics," *BMC Bioinformatics*, vol. 11, no. Suppl 12, pp. S1+, 2010.

[11] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming Journal*, vol. 13, pp. 277–298, 2005. [Online]. Available: http://research.google.com/archive/sawzall.html

[12] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: easy, efficient data-parallel pipelines," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 363–375.

[13] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. C. Kanne, F. Özcan, and E. J. Shekita, "Jaql: A Scripting Language for Large Scale Semistructured Data Analysis," *PVLDB*, vol. 4, no. 12, pp. 1272–1283, 2011.

[14] R. J. Stewart, P. W. Trinder, and H.-W. Loidl, "Comparing high level mapreduce query languages," in *Proceedings of the 9th international conference on Advanced parallel processing technologies*, ser. APPT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 58–72.

[15] B. Byambajav, T. Wlodarczyk, C. Rong, P. LePendu, and N. Shah, "Performance of left outer join on hadoop with right side within single node memory size," in *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, march 2012, pp. 1075 – 1080.

[16] P. Deligiannis, H.-W. Loidl, and E. Kouidi, "Improving the diagnosis of mild hypertrophic cardiomyopathy with mapreduce," in *In The Third International Workshop on MapReduce and its Applications (MAPREDUCE'12)*, 2012.

[17] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Mapreduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 1029–1040.

[18] A. Agarwal, M. Slee, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," Facebook, Tech. Rep., April 2007. [Online]. Available: http://incubator.apache.org/thrift/static/thrift-20070401.pdf

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