Knowl Inf Syst (2014) 41:531–557
DOI 10.1007/s10115-013-0705-z

REGULAR PAPER

Tuple MapReduce and Pangool: an associated implementation

Pedro Ferrera · Ivan De Prado · Eric Palacios · Jose Luis Fernandez-Marquez · Giovanna Di Marzo Serugendo

Received: 20 March 2013 / Revised: 13 September 2013 / Accepted: 17 October 2013 /
Published online: 24 December 2013

Abstract This paper presents 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 shows 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. Additionally, this paper shows: pseudo-codes for relational joins, rollup, and the PageRank algorithm; a Pangool's code example; benchmark results comparing Pangool with existing approaches; reports from users of Pangool in industry; and the description of a distributed database exploiting Pangool. These results show 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.

Keywords MapReduce · Hadoop · Big Data · Distributed systems · Scalability

1 Introduction

During the last years, the amount of information handled within different fields (i.e., web, sensor networks, logs, or social networks) has increased dramatically. Well-established approaches, such as programming languages, centralized 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 organizations that need to extract meaningful information from huge volumes of data and multiple sources. Even though many new technologies have been recently

P. Ferrera · I. De Prado · E. Palacios
Datasalt Systems S.L., Barcelona, Spain

J. L. Fernandez-Marquez (✉) · G. Di Marzo Serugendo
CUI, University of Geneva, 1227 Carouge, Switzerland
e-mail: joseluis.fernandez@unige.ch

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 [10]—a programming model introduced to support distributed computing on large datasets on clusters of computers. MapReduce has been proposed in 2004 [9] 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 parallelization, 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 [14], Hive [20], Cascading,[1] FlumeJava [5]). 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 be built on top of it in a straightforward way.

In a previous work [13], we proposed Tuple MapReduce, which targets those applications that perform batch-oriented distributed computation. Moreover, an implementation of Tuple MapReduce–Pangool—was provided and compared with existing implementations of MapReduce, showing that Pangool maintains Hadoop's performance. In this paper, we extend our previous work with (1) pseudo-codes for relational joins and the PageRank algorithm; (2) details about how Pangool was implemented on top of Hadoop; (3) examples of use of Pangool in the industry; and (4) the role of Pangool in the implementation of the distributed database Splout SQL.

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 its existing implementations in the literature. Section 3 identifies 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 Sect. 4. Additionally, this section shows and analyses different design patterns for data analysis, such as joins, relational joins, and rollup. Section 5 proposes and analyses a Tuple MapReduce implementation called Pangool, built on top of Hadoop. Moreover, we compare Pangool with existing approaches and show a Pangool's code example. Section 6 reports on the use of Pangool in industry as well as the development of a distributed database with Pangool. Finally, conclusions and future work are explained in Sect. 7.

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

[2] http://www.cloudera.com/.

2 Related work

In this paper, we concentrate on MapReduce, a programming model that was formalized by Google in 2004 [9] and on Hadoop, its associated de-facto open-source implementation. We briefly describe the main idea behind MapReduce and its Hadoop implementation, and 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 [15], machine learning [7], business intelligence [8], bioinformatics [19], 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 [17]. 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 [6]. 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 optimization are presented, leading to optimized 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 [14], 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 [20], 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.

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

Jaql [2] 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 execution.

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 has its own particularities, and there is also literature on the performance discrepancies among them [4,12,18].

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 have been studied by Yang et al. [21]. The authors illustrate MapReduce lack of direct support for processing multiple related heterogeneous datasets and for performing 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.

3 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 and secondary sorting), are not directly provided by MapReduce, or, even worst, they are complex to implement with it. In this section, we analyze and summarise the main existing problems arising when using MapReduce for solving common distributed computation problems:

1. **Compound records:** In real-world problems, data are 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

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

MapReduce abstraction. To overcome this complexity, it has become common practice to create custom datatypes, whose scope and usefulness are confined to a single MapReduce job. Frameworks for the automatic generation of datatypes have then been developed, such as Thrift [1] and Protocol Buffers [11], 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.

4 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.

4.1 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 for each individual partition. The reduce function receives as input a list of key/value pairs, all of them sharing the same

[5] http://pangool.net.

key, and converts them into zero, one or more key/value pairs. The set of pairs generated by the application of the reduce function to every partition is the final output.

Equations (1a) and (1b) summarize the behavior map and reduce functions. 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 datatypes. In other words,

- All key and value items received by the map function have the same datatypes k_1 and v_1, respectively.
- All key and value items emitted by the map function have the same datatypes k_2 and v_2, respectively.
- Reduce input key and values datatypes are k_2 and v_2, respectively. Map output and reduce input must then 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, and 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. This process is executed transparently in a distributed way by the MapReduce implementation.

4.2 Tuple MapReduce

The fundamental idea introduced by Tuple MapReduce is the usage of *tuples* within its formalization. Tuples have been widely used in higher-level abstractions on top of MapReduce (e.g., FlumeJava [5], Pig [14], 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 datatypes 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 field 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 (2a) and (2b) summarize the new map and reduce functions contract and the datatypes for each tuple field.

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

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

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

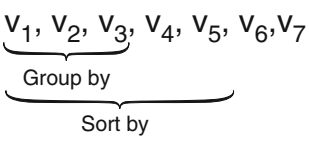


Fig. 1 Group-by and sort-by fields

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 Fig. 1). For clarity, group-by fields and sort-by fields are defined to be a prefix of the emitted tuple, group-by being 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 datatypes.

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 share the same first g fields. Additionally, tuples in the input list are sorted by their 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 us say that tuple A has the same schema as tuple B if A and B have the same number of field 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:

- 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.

4.3 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.

Algorithm 1: Cumulative visits

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

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

    end

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

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.

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.

4.4 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.

Equation (3a–3c) summarizes the functions contract and the datatypes 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.

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

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

$$\begin{aligned} reduce(v_1, v_2, \ldots, v_g), list((v_1, v_2, \ldots, v_n)_A), \\ list((v_1, v_2, \ldots, v_m)_B) \rightarrow \\ list((k_1, k_2, \ldots, k_q)) \\ (g \leq n)(g \leq m) \end{aligned} \tag{3c}$$

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).

4.5 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

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 then return
    foreach payment ∈ rest(tuples) do
        emit(Tuple(client.get("clientName"),
          payment.get("amount")))

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

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; 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 not be scalable when there are a lot of payments per client so that they do not fit in memory. Tuple MapReduce allows for reduce-side inner, left, right, and outer joins without memory consumption in the case of a 1-to-*n* join, and minimal memory consumption for *n*-to-*n* relational joins, as we will see in the next section.

4.6 Implementing relational joins in Tuple MapReduce

Relational joins are widely used in data analysis systems, and it is common to use abstractions that implement them. In fact, such abstractions have already been implemented on top of MapReduce, but the cost of implementing them is much higher than that of formalizing them on top of Tuple MapReduce. As an illustrative example, we show how generic relational joins are implemented on top of Tuple MapReduce. In Algorithm 3, we see the pseudo-code of an inner join between two sets of tuples (named left and right after traditional join name conventions). Hereafter, we refer to those sets as tables for convenience.

In the reduce method, we use a memory buffer for tuples belonging to the left side of the join in order to perform the cross-product with the right side afterward. In this case, intra-reduce sorting allows us to use only one buffer because, after we have gathered all left tuples, we are sure that the remaining tuples are going to belong only to the right side of the join. Moreover, when using this abstraction, we could leverage the fact that the left table tuples are saved in a buffer to perform the join in the most memory-efficient way: that is, using the smallest table on the left and the biggest one on the right.

On the other hand, the joinTuple method receives both tuples from left and right tables in order to be able to return a compound, joint tuple. The semantics of this method are not fixed by this pseudo-code and can be implemented in various ways.

In Algorithm 4, we change the reduce method so that it implements a left join instead of an inner join. The code becomes somewhat more complex. Indeed, we need to handle the case where a group of tuples have no corresponding set of tuples in the right side of the join. In this case, we use a state boolean variable (rightTuples) and call joinTuple with a null value as right tuple, meaning that the method will return the compound tuple with the appropriate fields set to null. So we are assuming that this method is aware of both left and right schemas.

Algorithm 5 shows the pseudo-code of a right join reducer. Compared to the inner join, we just need to cover the case where the left tuple buffer is empty, in which we will emit joint tuples with nulls corresponding to the left part of the join. Finally, Algorithm 6 implements an outer join, which indeed covers both left and right join cases at the same time.

Algorithm 3: Relational inner join

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

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

reduce(groupTuple, tuples):
    leftTuples = []
    foreach tuple ∈ tuples do
        if tuple.getSourceId()==0 then
            addToArray(tuple,leftTuples)
        else
            foreach leftTuple ∈ leftTuples do
                emit(joinTuple(leftTuple, tuple))
            end
        end
    end
```

4.7 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 formalization. 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.

Let us 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
```

Algorithm 4: Relational left join

```
reduce(groupTuple, tuples):
    leftTuples = []
    rightTuples = false
    foreach tuple ∈ tuples do
        if tuple.getSourceId() == 0 then
            addToArray(tuple, leftTuples)
        else
            rightTuples = true
            foreach leftTuple ∈ leftTuples do
                emit(joinTuple(leftTuple, tuple))
            end
        end
    end
    if rightTuples == false then
        foreach leftTuple ∈ leftTuples do
            emit(joinTuple(leftTuple, null))
        end
    end
```

Algorithm 5: Relational right join

```
reduce(groupTuple, tuples):
    leftTuples = []
    foreach tuple ∈ tuples do
        if tuple.getSourceId() == 0 then
            addToArray(tuple, leftTuples)
        else
            if leftTuples.size > 0 then
                foreach leftTuple ∈ leftTuples do
                    emit(joinTuple(leftTuple, tuple))
                end
            else
                emit(joinTuple(null, tuple))
            end
        end
    end
```

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*

Algorithm 6: Relational outer join

```
reduce(groupTuple, tuples):
    leftTuples = []
    rightTuples = false
    foreach tuple ∈ tuples do
        if tuple.getSourceId() == 0 then
            addToArray(tuple, leftTuples)
        else
            if leftTuples.size > 0 then
                foreach leftTuple ∈ leftTuples do
                    emit(joinTuple(leftTuple, tuple))
                end
            else
                emit(joinTuple(null, tuple))
            end
        end
    end
    if rightTuples == false then
        foreach leftTuple ∈ leftTuples do
            emit(joinTuple(leftTuple, null))
        end
    end
```

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 $location$. This is possible by 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 previously mentioned, 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 7 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.

Algorithm 7: Rollup example

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

onOpen(field, tuple):


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

reduce(groupTuple, tuples):
    dateCount = 0
    foreach tuple ∈ 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")
```

The rollup API simplifies the implementation of efficient multi-level aggregations by the automatic detection of group changes.

4.8 PageRank example

PageRank [16] is an algorithm, currently used by Google,[7] that analyzes the relevance of Web pages based on graphs where nodes are Web pages and hyperlinks are edges.

In Algorithm 8, we observe the pseudo-code of a simple implementation of PageRank in Tuple MapReduce. The code implements the iterative step, which is core to the calculation and which updates the PageRank value for each authority by adding the contributions of each of its supporting authorities. In this example, authorities are URLs, and therefore, supporting authorities are modeled by "outlinks".

For implementing this step in Tuple MapReduce, we use a tuple with three fields: "*url*", "*pagerank*", and "*outlink_list*". Because of the nature of the parallel implementation of PageRank, in the map phase, we do not emit all three fields at once, and thus, we emit Tuples with some null fields instead. We assume that the associated implementation of Tuple

[7] www.google.com.

MapReduce handles nulls efficiently and serializes the rest of the Tuple as needed. In the reduce function, we receive all the contributions for the same url and add them up using the PageRank formula, which takes into account a "$damping\_factor$" in order to model the stochastic nature of the model—in this case, human browsing.

Compared to an implementation on top of plain MapReduce, this PageRank implementation requires minimal effort in reasoning about the nature of the intermediate data structures, which are emitted between the map and the reduce phase. In plain MapReduce, we would need to use a compound record in the value in order to emit both an outlink list and a pagerank value. Additionally, this compound record must be efficiently serialized in the absence of one of those two fields for the computation to be efficient.

Algorithm 8: PageRank example

```
map(tuple):
    outlinkList = tuple.get("outlink_list")
    foreach outlink ∈ outlinkList do
        emit(Tuple(outlink, tuple.get("pagerank") /
        size(outlinkList), null))
    end
    emit(Tuple(tuple.get("url"), null, outlinkList))

reduce(groupTuple, tuples):
    outlinkList = []
    pagerank = 0
    foreach tuple ∈ tuples do
        if notNull(tuple.get("outlink_list")) then
            outlinkList = tuple.get("outlink_list")
        else
            pagerank += tuple.get("pagerank")
        end
    end
    pagerank = 1 - DAMPING_FACTOR +
    (DAMPING_FACTOR * pagerank)
    emit(Tuple(tuple.get("url"), pagerank, outlinkList))
```

4.9 Tuple MapReduce as a generalization of classic MapReduce

Tuple MapReduce as discussed in this section can be seen as a generalization of the classic MapReduce. Indeed, the MapReduce formalization is equivalent to a Tuple MapReduce formalization with tuples constrained to be of size two, group-by being done on 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. Indeed, the architecture needed for parallelizing Tuple MapReduce is exactly the same as the one needed for MapReduce—the only substantial changes needed lay in the serialization and API parts. Nothing on the distributed nature of the MapReduce architecture has to be changed. 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 on top of the existing MapReduce implementation.

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

5 Pangool: Tuple MapReduce for Hadoop

In March 2012, we released an open-source Java implementation of Tuple MapReduce called Pangool[8] ready to be used in production. We 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 the 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's guide[9] is the reference for learning how to use it. Additionally, many examples showing how to use Pangool are provided on-line.[10]

5.1 Implementing Pangool on top of Hadoop

The cost and complexity of implementing a distributed version of Tuple MapReduce from scratch would have been prohibitive because of the intrinsic complexity of distributed systems. However, Tuple MapReduce can be developed on top of existing MapReduce implementations without the need of important architectural changes, which is an important feature. Pangool is a fully functional production-ready implementation of Tuple MapReduce built on top of Hadoop. No architectural changes were needed. Indeed, Pangool is just a library to plug in Hadoop developments. This library converts Tuple MapReduce jobs into simple MapReduce jobs by providing some custom classes that overload the default MapReduce main components: Serialization, Sort Comparator, Group Comparator, and Partitioner.

The main tasks carried out during Pangool's implementation were as follows:

- Schema definition language for Tuples.
- Nulls fields.
- Intermediate serialization based on the Schema.
- Custom Partitioner based on the group-by clause.
- Custom SortComparator based on the sort-by clause.
- Custom GroupComparator based on the group-by clause.
- New "mapper", "reducer," and "combiner" classes ready to deal with Tuples.
- TupleInputFormat and TupleOutputFormat for Tuples' persistence.

We discuss here some of them, namely schemas, nulls, and serialization.

[8] http://pangool.net.

[9] http://pangool.net/userguide/schemas.html.

[10] https://github.com/datasalt/pangool/tree/master/examples/src/main/java/com/datasalt/pangool/examples.

5.1.1 Schemas

By obtaining the control over the intermediate serialization through our own serialization classes, we have built an API on top of Hadoop, which allows the user to specify Tuple schemas, which are then obtained by the serialization classes through the use of the Hadoop Configuration object. The schemas define the datatypes that each Tuple has and the order in which they are serialized. Because the schema is defined before running the MapReduce job, we can then serialize the Tuples efficiently, using just as many bytes as needed for each datatype, and concatenating them according to the order in which they were defined. If one job uses more than one schema (i.e., a join), then a variable-length integer (VInt) is added in front of the serialized data, in order to identify the tuple schema. Equation 4 shows an example declaration of a Pangool schema. The schemas can be formed by primitive datatypes (INT, LONG, FLOAT, DOUBLE, STRING, BOOLEAN, BYTES), Java enums (ENUM), or custom objects (OBJECT). Whichever object that can be serialized using one of the registered serializations in the Hadoop serialization registry is supported as a custom object field. This makes the system fully backwards-compatible with Hadoop, allowing users to reuse their own serialization methods inside Pangool Tuples as well as other serialization libraries such as Avro or Thrift.

$$client : int, average\_salary : double, position : int, \\ title : string, fired : boolean \tag{4}$$

5.1.2 Handling nulls

Initial versions of Pangool did not have support for null values, just as the Key and the Value of the standard Hadoop MapReduce API cannot be null either. This supported the idea of Tuple MapReduce being a generalization of MapReduce. However, the convenience of having nulls in a Tuple becomes obvious when integrating Pangool with other Extract, Transform, and Load (ETL) systems. The user may frequently import datasets, which naturally contain "null" values into Pangool Tuples. An example of this process could be importing data from a relational database for processing them using Tuple MapReduce.

Pangool adds support for "null" values as an optional feature. So it is still possible to use Pangool in a "non-null values mode," which will obviously serialize data more efficiently. But if "nulls" are enabled, then Pangool will serialize them as efficiently as it can, by using variable-length byte arrays. Each bit in the byte array acts as a "mask" for the values that can be "null" in a Tuple. So if a Tuple contains 3 fields with possible "null" values, only a single extra byte is needed for indicating the possible presence of a null in any of those 3 fields. This optimization can be done because, as explained in the previous subsection, the order of the datatypes inside the tuple is known beforehand.

Equation 5 shows an example declaration of an augmented Pangool schema, which declares two possible null fields with the sign "?".

$$client : int, average\_salary : double, position : int?, \\ title : string?, fired : boolean \tag{5}$$

5.1.3 Serialization and data routing

At the lower level, Pangool data sit on top of "datum wrappers" that are finally serialized as binary data according to the schemas used, as explained in previous subsections.

Table 1 Lines of code reduction

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

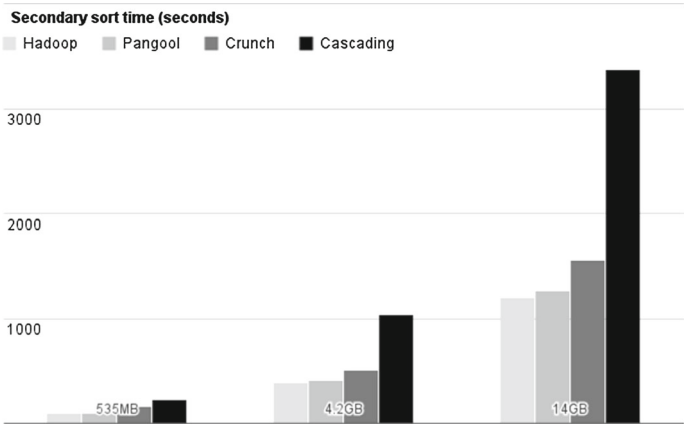


Fig. 2 Secondary sort time (seconds)

In connection with the Hadoop API, the MapReduce "key" are those wrappers, and the "value" is just null. By wrapping all the data in the key, and because Hadoop sorts and groups the data by this key, we implemented our own sorting and grouping comparators, which allows for flexible secondary sorting over the Tuple fields. Obviously, a custom partitioner was implemented so that tuples are assigned to reducers properly, based on the group-by clause. We also implemented our own Hadoop "mappers," "reducers," and "combiners," which unfold the Tuples from the binary data and provide those Tuples to the final API user.

All these systems have been implemented by reusing existing object instances and buffers when possible in order to keep a high efficiency (i.e., creating a minimum overhead over the existing MapReduce API).

5.2 Benchmark performance

Table 1 shows the differences in number of lines when implementing two different tasks in both Hadoop and Pangool. The first task involves a secondary sort, while the second task involves joining two datasets. Both tasks use compound records. These tasks can be seen in Pangool's examples[11] and benchmark[12] 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,[13] Cascading 1.2.5[14]). 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

[11] https://github.com/datasalt/pangool/tree/master/examples.

[12] https://github.com/datasalt/pangool-benchmark.

[13] https://github.com/cloudera/.

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

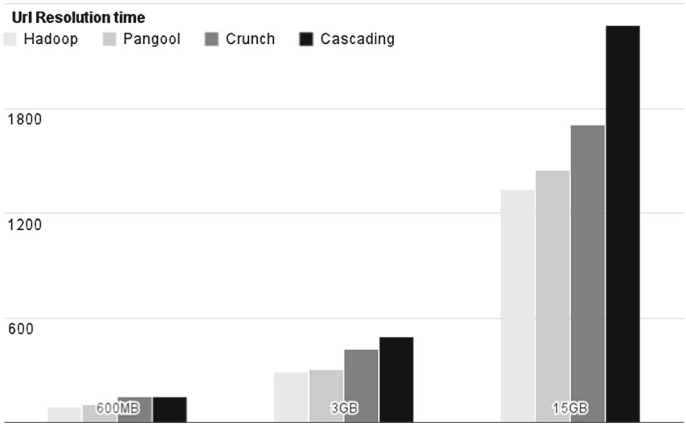


Fig. 3 URL resolution time (seconds)

compared in Table 1. It involves grouping compound records of four fields grouping by two of them and performing secondary sort on a third field.

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 5–8 % 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.[15] The full benchmark consists of a comparison of three tasks, one of them being the well-known word count task.

6 Pangool in the industry

This section describes the use of Pangool by the industry in the development of applications as well as the use of Pangool in the development of higher-level systems, such as a distributed database. Besides, this section intends to show the prevalence of the common design patterns described in Sect. 4 in real-life problems, highlighting the utility of Tuple MapReduce.

6.1 Reports from the use of Pangool in industry

Pangool is currently being used with success in Datasalt[16] projects, simplifying the development and making code easier to understand, while still keeping efficiency. Datasalt has successfully deployed applications in banking, telecommunications, and Internet sectors.

[15] http://pangool.net/benchmark.html.

[16] http://www.datasalt.com.

So far, Datasalt implemented a full pipeline with Pangool for analyzing credit card transactions for the BBVA bank.[17] It included the computation of statistical analysis over the credit card transactions over different time periods, including different aggregations. This is a typical case where compound records are needed because records are not just key/value pairs but a set of fields instead. A recommender system based on co-occurrences that made intensive use of intra-reduce sorting and joins was also implemented. The use of Pangool allowed the development of the system maintaining the highest efficiency without having to deal with the complexities that classic MapReduce imposes.

Some other use cases have been gathered from the Pangool users community.[18] As an example, a Pangool user reported the migration of an existing log loading pipeline to Pangool in a telecommunications company. As reported by him, the legacy system was implemented by the use of plain MapReduce jobs mixed with Pig scripts for joins, aggregations, rollups, deduplication, etc. Oozie[19] was used to put all the pieces together in a flow. The user described the system as difficult to use and debug. Additionally, he specified that the management of the flow was cumbersome.

They decided to move the whole pipeline to Pangool because it has some useful features similar to those in Pig (i.e., joins and compound records), still providing a low-level API that allows developers to retain full control.

Therefore, testing, developing, and having control became easier by the use of Pangool.

6.2 Splout SQL: a distributed system made on top of Tuple MapReduce

Splout SQL[20] is an open-source, distributed, read-only database. It was released in 2012 and developed mainly using Pangool. Its main purpose is that of making large files residing on the Hadoop Distributed File System (HDFS) easily accessible and queryable through a low-latency SQL interface that can be used to power websites or mobile applications.

Splout SQL scales through the use of partitioning. Data in tables must be partitioned by a column or a set of columns. Queries are then routed to the proper partition by the use of an explicit key that must be provided when submitting queries. Each partition is just binary data that is used by SQL engines to serve the queries. By default, Splout SQL uses SQLite[21] as underlying database engine to serve the partition data but other engines can be plugged as well.

Data in Splout SQL are stored in tablespaces. Each tablespace is a set of tables all of them sharing the same partitioning schema. Figure 4 shows the example tablespace *CLIENTS_INFO* that is built from two tables, *CLIENTS* and *SALES*, and that is partitioned by the column *CID*. The resulting tablespace contains two partitions, each of them containing a subset of the source tables data. Tablespace partitions are generated in Hadoop from HDFS data.

Once a tablespace has been generated in Hadoop, it can be deployed atomically into the serving cluster. Figure 5 shows clearly the two main phases of the Splout SQL database:

1. **Generation:** This phase that runs entirely in Hadoop is responsible for building the tablespaces from source data, creating different partitions of approximately equal sizes.

[17] http://highscalability.com/blog/2013/1/7/analyzing-billions-of-credit-card-transactions-and-serving-l.html.

[18] https://groups.google.com/forum/#!forum/pangool-user.

[19] http://oozie.apache.org/.

[20] http://sploutsql.com/.

[21] http://www.sqlite.org/.

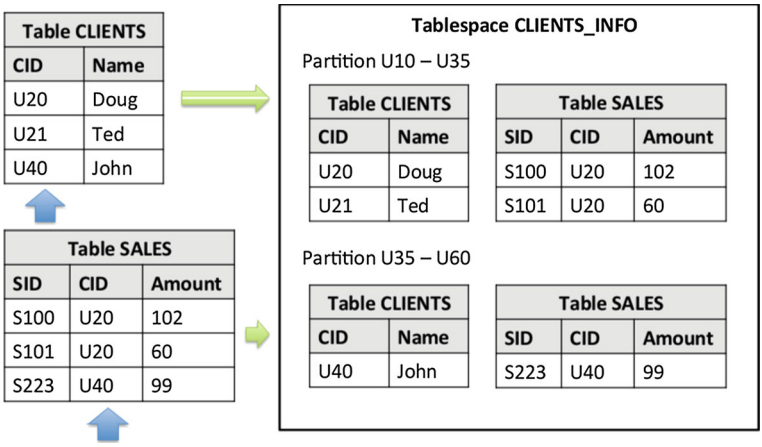


Fig. 4 Example of a generated partitioned tablespace

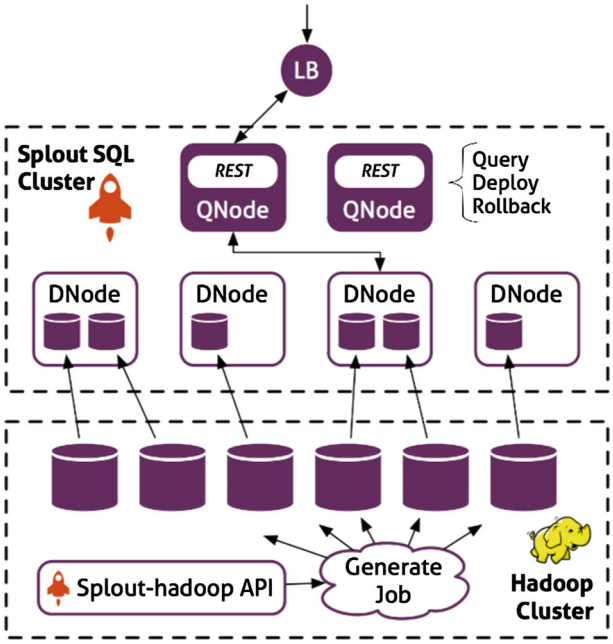


Fig. 5 Splout SQL architecture

Each partition is a SQLite database with its tables and indexes. Once this phase is finished, the tablespace is ready to be deployed into the serving cluster.

2. **Serving:** The responsibilities of the serving phase are routing user queries to the proper tablespace and partition, deploying new tablespaces that were previously generated, and other management tasks such as tablespace versioning and rollback. This phase runs in its own cluster that is independent from the Hadoop cluster. The cluster is coordinated by two services: the QNode, responsible for routing the queries to the proper tablespace

and partition; and the DNode, responsible for running the particular queries over the partitions.

By isolating the generation of query-optimized data structures from their serving, Splout SQL can assure almost no impact on query serving when the data are being updated. The deployment of new tablespace versions is just a copy of files, minimizing the impact on serving and therefore ensuring stable response times.

Data partitioning ensures scalability for those applications where queries can be restricted to particular partitions (for example, to a particular client or a particular location). The use of SQL provides a rich query language and fast query responses with the help of indexes.

Splout SQL is a suitable solution for web or mobile applications that need to serve Big Data tasks ensuring millisecond response times. It provides what can be thought of "SQL materialized views over HDFS data".

Splout SQL is nowadays being used in production by client companies of Datasalt for serving terabytes of statistics from advertising networks.

6.2.1 Splout SQL and Tuple MapReduce

In this section, we will explain how Pangool (an therefore Tuple MapReduce) was a key piece for the development of Splout SQL. Splout SQL benefited from using Pangool as opposed to the plain MapReduce API in several ways:

1. **Datatypes and parsing:** Splout SQL needed to support heterogeneous file types, from CSV files to arbitrary HCatalog tables. The tuple abstraction offered a clean bridge between any data source and Splout SQL. Indeed, because Splout SQL generates SQL tables, these can be seen as a collection of tuples with a predefined schema for each table, which is exactly the abstraction found in Tuple MapReduce. An alternative would have been to use an auxiliary serialization library like Thrift on top of classic MapReduce, but that would have left open other issues we will describe next.
2. **Joining and sorting:** Because data locality in SQL engines influences query response times, the order in which SQL "insert" statements are issued is very important. As we will explain, a special intra-reduce sorting was needed in order to obtain control on data insertion order. The Tuple MapReduce support for intra-reduce sorting and joins covers this requirement. Implementing such a requirement in plain MapReduce would have been quite complex: it would have included the creation of a custom Sort Comparator, Group Comparator, and Partitioner; none of them are needed if Pangool is used.
3. **Code conciseness:** The simplicity and expressiveness of the Pangool API allowed the code of the Tuple MapReduce jobs to be much shorter (and thus easily understandable and maintainable) to what their Java MapReduce versions would have looked like.

We will now explain the advantages that Pangool offered when implementing the generation phase of Splout SQL, thus highlighting the relevance of Tuple MapReduce for real-life applications.

In order to obtain efficiently indexed binary files out of any file, Splout SQL executes an indexation process as a series of Hadoop jobs, with the desired data files as input to that process. As explained previously, the output of this process is a set of binary SQLite files that can be then deployed to the Splout SQL cluster.

The generation process is composed of two jobs, namely (1) the Sampling Job and (2) the Data Indexation Job. The Sampling Job is in charge of determining how the data are distributed. The data distribution will then be used by the data indexation job for creating

evenly distributed partitions. The main responsibilities of the indexation job are parsing the different input sources, routing data to the proper partition, and parallelizing the generation of the different partitions.

A more detailed description of each job is given as follows:

1. **The Sampling Job.** The Sampling Job is a distributed Reservoir Sampling implementation. Because this job can be implemented using a *map-only job* (with norReducers), a special Pangool wrapper called MapOnlyJobBuilder was used. Thanks to Pangool's instance-based configuration (which removes the need for defining a separate Java class for every mapper, reducer, and so forth), the Reservoir Sampling code was shortened and placed conveniently.
2. **The Data Indexation Job.** The Indexation Job is a complex MapReduce process, which encompasses all the features that make programming in plain MapReduce difficult: the use of compound records, intra-reduce sorting, and joins. With its main part done in barely 200 lines[22] using Pangool, it would be quite difficult even for a MapReduce expert to estimate how its pure plain MapReduce version would have looked like.

This job needs to execute a specific business logic for every input file to the database creation process, dealing with data which are potentially heterogeneous (multiple data entities may be merged into the same database and that is achieved by creating a SQL table for each of them). Here, as mentioned before, the use of tuples was an appropriate choice for abstracting the codebase from the possible derived complexities arising from dealing with arbitrary compound datatypes. Even more, a tuple-based metadata service for Hadoop (HCatalog) was leveraged for making it possible to read data from any standard data source in the "Hadoop Ecosystem" (namely Hive, Pig, and such).

The overall indexation process looks as follows.

- Firstly, a specific mapper is defined for every input file, which will parse it as needed and will emit a tuple with the proper schema for each input record. A field with the proper "partition id" will be injected in each tuple. The partition id is set according to the distribution of the data computed by the Sampling Job.
- Secondly, all tuples are joined by "partition id": every reducer receives all tuples belonging to the same "partition id" and streams them into an embedded SQLite database in the form of SQL statements. The order in which tuples reach the reducer is crucial for optimizing query resolution times. Because of that, the order in which records are inserted into the database can be managed by the end user. It is a well-known fact that this order is crucial for query performance, as it allows the user to co-locate data close in disk and make scan queries faster.
- Finally, each reducer executes the creation of database indexes requested by the user, closes the database, and uploads the resultant SQLite database file to the HDFS.

To sum up, Pangool (and therefore Tuple MapReduce) simplified the development of such a complex and low-level application without losing performance. In this section, we have shown that the design patterns described in Sect. 4 (namely *compound records*, *intra-reduce sorting* and *joins*) arise in many real-life problems. The MapReduce limitations on this field are covered properly by the proposed Tuple MapReduce paradigm. The associated implementation, Pangool, shows that Tuple MapReduce has a performance level similar to that of MapReduce, while providing a richer foundation to the developer.

[22] https://github.com/datasalt/splout-db/blob/master/splout-hadoop/src/main/java/com/splout/db/hadoop/TablespaceGenerator.java.

7 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. Moreover, we have shown several real-life uses of Pangool that certify their utility in production scenarios. 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.

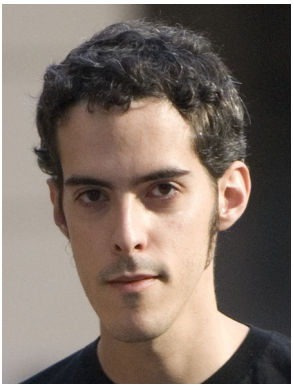
Future work includes the development of new abstractions that simplify the task of chaining MapReduce jobs in a flow. We believe that there is room for improvement in this field. Concretely, we plan to build an abstraction for easing running flows of many parallel jobs, incorporating ideas from tools such as Azkaban[23] and Cascading.

References

1. Agarwal A, Slee M, Kwiatkowski M (2007) Thrift: scalable cross-language services implementation, technical report, Facebook. http://incubator.apache.org/thrift/static/thrift-20070401.pdf
2. Beyer KS, Ercegovac V, Gemulla R, Balmin A, Eltabakh MY, Kanne CC, Özcan F, Shekita EJ (2011) Jaql: a scripting language for large scale semistructured data analysis. PVLDB 4(12):1272–1283
3. Borthakur D (2007) The hadoop distributed file system: architecture and design. The Apache Software Foundation, Los Angeles. https://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf
4. Byambajav B, Wlodarczyk T, Rong C, LePendu P, Shah N (2012) Performance of left outer join on hadoop with right side within single node memory size. In: 26th international conference on advanced information networking and applications workshops (WAINA), 2012, pp 1075–1080
5. Chambers C, Raniwala A, Perry F, Adams S, Henry RR, Bradshaw R, Weizenbaum N (2010a) FlumeJava: easy, efficient data-parallel pipelines. SIGPLAN Not 45(6):363–375
6. Chambers C, Raniwala A, Perry F, Adams S, Henry RR, Bradshaw R, Weizenbaum N (2010b) Flumejava: easy, efficient data-parallel pipelines. In: Proceedings of the 2010 ACM SIGPLAN conference on programming language design and implementation. PLDI '10, ACM, New York, NY, USA, pp 363–375
7. Chu CT, Kim SK, Lin YA, Yu Y, Bradski GR, Ng AY, Olukotun K (2006) Map-reduce for machine learning on multicore. In: Schölkopf B, Platt JC, Hoffman T (eds) NIPS. MIT Press, Cambridge, MA, pp 281–288
8. Dayal U, Castellanos M, Simitsis A, Wilkinson K (2009) Data integration flows for business intelligence. In: Proceedings of the 12th international conference on extending database technology: advances in database technology. EDBT '09, ACM, New York, NY, USA, pp 1–11

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

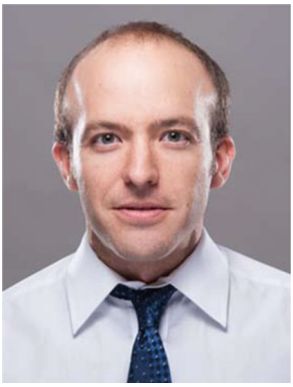
9. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on symposium on operating systems design & implementation, vol 6. OSDI'04, ACM, USENIX Association, pp 10–10
10. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
11. Dean J, Ghemawat S (2010) MapReduce: a flexible data processing tool. Commun ACM 53(1):72–77
12. Deligiannis P, Loidl H-W, Kouidi E (2012) Improving the diagnosis of mild hypertrophic cardiomyopathy with mapreduce. In: In the third international workshop on MapReduce and its applications (MAPREDUCE'12)
13. Ferrera P, de Prado I, Palacios E, Fernandez-Marquez J, Di Marzo Serugendo G (2012) Tuple MapReduce: beyond classic MapReduce. In: IEEE 12th international conference on data mining (ICDM), pp 260–269
14. Gates AF, Natkovich O, Chopra S, Kamath P, Narayanamurthy SM, Olston C, Reed B, Srinivasan S, Srivastava U (2009) Building a high-level dataflow system on top of Map-Reduce: the Pig experience. Proc VLDB Endow 2(2):1414–1425
15. Grossman R, Gu Y (2008) 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. KDD '08, ACM, New York, NY, USA, pp 920–927
16. Page L, Brin S, Motwani R, Winograd T (1998) The PageRank citation ranking: bringing order to the web, technical report, Stanford Digital Library Technologies Project
17. Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: parallel analysis with Sawzall. Sci Program J 13:277–298. http://research.google.com/archive/sawzall.html
18. Stewart RJ, Trinder PW, Loidl H-W (2011) Comparing high level mapreduce query languages. In: Proceedings of the 9th international conference on advanced parallel processing technologies. APPT'11, Springer, Berlin, pp 58–72
19. Taylor R (2010) An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. BMC Bioinformatics 11(Suppl 12):S1+
20. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive: a warehousing solution over a map-reduce framework. Proc VLDB Endow (PVLDB) 2(2):1626–1629
21. Yang HC, Dasdan A, Hsiao R-L, Parker DS (2007) Map-reduce-merge: simplified relational data processing on large clusters. In: Proceedings of the 2007 ACM SIGMOD international conference on management of data. SIGMOD '07, ACM, New York, NY, USA, pp 1029–1040



Pedro Ferrera is CTO of Datasalt, a company offering Big Data consulting and which is building innovative open-source products around the Hadoop eco-system such as Pangool or Splout SQL. Pere works now in Berlin, influencing its Big Data scene. He has worked in the past in several Barcelona start-ups, facing early scalability problems and helping companies build their engineering solutions.

Ivan De Prado is CEO of Datasalt, a company offering Big Data consulting and which is building innovative open-source products around the Hadoop eco-system such as Pangool or Splout SQL. Ivn has worked on several start-ups such as Strands and Enormo in a wide variety of roles, posing several challenges and therefore gaining broad experience in the Big Data space.

Eric Palacios works currently in Barcelona as a backend software engineer in Trovit Search S.L, one of the leading search engines in classified ads in Europe. Eric started to work with Hadoop and several Big Data technologies in their early stages and has been gaining experience since then. Back in 2011, he joined Pere and Ivan in Datasalt that offered him the possibility of leading the design and development of Pangool.



Jose Luis Fernandez-Marquez is currently working as full-time researcher at the Institute of Services Science at the University of Geneva, with a research contract funded by SAPERE EU project. In 2011, he defended his Ph.D. Thesis titled Bio-inspired Mechanisms for Self-Organizing Systems. His expertise is mainly focused on highly distributed, self-organizing, and multi-agent systems. He has a wide experience implementing large-scale simulations and engineering self-organizing systems. Additionally, his research interests are also focused on swarm intelligence, high performance computing, and pervasive computing. Along his research carrier, he has participated in the organization of many conferences and workshops, such as Symposium on Applied computing (SAC2012, SAC2013), Self-Adaptive and Self-Organizing systems (SASO2012), or Adaptive Service Ecosystems: Nature and Socially Inspired Solutions (ASENSIS2012). He is also involved as reviewer in high impact international journals and conferences.

Giovanna Di Marzo Serugendo joined the University of Geneva in August 2010. Since January 2011, she is the Director of the Institute of Services Science (ISS) at the University of Geneva and has extensive experience in the field of self-organizing and self-adapting systems. She received a Ph.D. in Software Engineering from the Swiss Federal Institute of Technology in Lausanne (EPFL) in 1999. Her research interests are related to the engineering of decentralized software with self-organizing and emergent behavior. She is exploring the use of appropriate architectures for developing trustworthy, dependable and controllable self-organizing and self-managing systems, including systems involving large-scale information dissemination and flows. She is part of the editorial board of several international journals in the areas of grid and intelligent systems. She co-founded the IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO) and the ACM Transactions on Autonomous Adaptive Systems (TAAS), for which she served as EiC from 2005 to 2011.