# The Evaluation of Map-Reduce Join Algorithms

Maciej Penar and Artur Wilczek

{artur.wilczek}@pwr.edu.pl
{penar.maciej}@gmail.com

**Abstract.** In recent years, Map-Reduce systems have grown into leading solution for processing large volumes of data. Often, in order to minimize the execution time, the developers express their programs using procedural language instead of high-level query language. In such cases one has full control over the program execution, what can lead to several problems, especially when join operation is concerned. In the literature the wide range of join techniques has been proposed, although many of them cannot be easily classified using old Map-Side/Reduce-Side distinction. The main goal of this paper is to propose the taxonomy of the existing join algorithms and provide their evaluation.

**Keywords:** Map-Reduce, Join algorithms, Hadoop, Taxonomy, Bigdata, Use Cases

## 1 Introduction

Increasing popularity of the Map-Reduce programming model encourages usage of this model to analyse huge sets of data. Unfortunately, unless the frameworks like Pig [2] or Hive [1] are used, the lack of declarative query languages makes the programmer responsible for the Map-Reduce program performance. Therefore, optimization of certain operations, which are analogous to the relational algebra operations, is a top priority in order to achieve the minimal execution time. One of the most expensive operations is the join operation. Due to the fact that often the technologies based on the Map-Reduce model are cloud-based solutions, the optimization task has even more impact as the execution time directly corresponds to the cloud utilization cost. In this paper, the overview of the existing join algorithms is presented, as well as design patterns for the join jobs to help the reader with choosing the best solution for own usages.

Due to the fact that among recent years plethora of Map Reduce join algorithms were designed and many of them seem to be similar and do not fall easily into existing taxonomy, the authors wished to contribute as follows:

- we extend current taxonomy of the Map Reduce joins, search for them among various sources and categorize them using proposed taxonomy
- we discuss commonly used techniques, pointing their advantages and disadvantages and highlighting the most frequent pitfalls concerning the implementation as well as use cases

## 2 Map-Reduce model

The main concept behind Map-Reduce system was to provide an easy way to express queries to large volumes of unconstrained data, which could be effortlessly paralleled and easily partially recomputed ( [8] and [11]). From the developer's point of view, Map-Reduce programs are evaluated in two steps called Map and Reduce, which have following responsibilities:

**Map** - the step executed by system agents called Mappers during which input is processed step by step (which often means line by line), as partial results are decorated with reduce key and are emitted for further processing (preferably in Reduce step)
**Reduce** - the step executed after Map phase by system agents called Reducers during which all partial results with common reduce key are being aggregated to one final result

In addition to this, developer can define quasi-step named Combine. However, one should be cautious when implementing it as the decision about its invocation is made internally by cluster, thus Combine may not be performed at all.

## 3 Taxonomy

The complexity of joining several data files in Map-Reduce model is determined by two separate factors: how many data sources are joined together and the condition itself for joining data structure. Therefore, in terms of the number of files being joined together and the type of join condition we can distinguish four kinds of the problem:

**Two-way theta Join** - denoted $R \bowtie_\theta S$. Given two datasets $R$ and $S$ and the predicate $\theta$, the theta join is defined as a subset of cartesian product of these data sets: $R \bowtie S \subseteq R \times S$, such that for every tuple $t$ in this subset the predicate over the tuple $t$ is true: $\forall_{x \in R \bowtie S} \theta(x) = true$
**Two-way equi-join** - is a special case of two-way theta join, where the predicate $\theta$ is equality comparision between values of one or more attributes from input data sources.
**Multi-way theta-join** - when two or more data sets are joined using any condition
**Multi-way equi-join** - is a special case of multi-way theta join, when the join predicate is an equality comparison between values of one or more attributes from input data sources.

These definitions describe the class of joins called Inner joins. That means that if a record originating from one of the joined table do not fit any record in other table, it will not be visible in the final result. Such records are often referred to as *dangling tuples* and they are bound to exist if they originate from a data source which has little or no referential constraints. Special kind of join named (Full) Outer join expresses the result where dangling tuples are included from both data sources. Specialized versions of the Outer join operator exist named Left/Right Outer Join which add dangling tuples from only one data set.

### 3.1 Joins in Map-Reduce model

In contemporary sources concerning Map-Reduce join algorithms one may come across something we may call *classical* taxonomy - the categorization based on where the actual join takes place [18], [7]:

**Map side joins** - these group of algorithms emits the joined record as a Mappers' output. These often leads to preprocessing or forcing the locality of one of the file used for join in Mappers'.

**Reduce side joins** - these group of algorithms emits the record as a Reducers' output. The general concept is to partition and process the groups of records with a common join key, with a way of distinguishing the source of record, to avoid joining the two records originating from the same file.

This simple taxonomy poses two major drawbacks: (1) it does not capture other algorithms which perform the join before the Map/Reduce phase, (2) it suggests that Map-Side join should always be performed during the Map phase while in some cases it is advisable to defer the tuple reconstruction to the Reduce phase - if the grouping can be performed without the join.

As the names of the algorithms seem to have enrooted and little can be done about that - we propose the extended version of the taxonomy (Figure 1) which amortize the first reason.

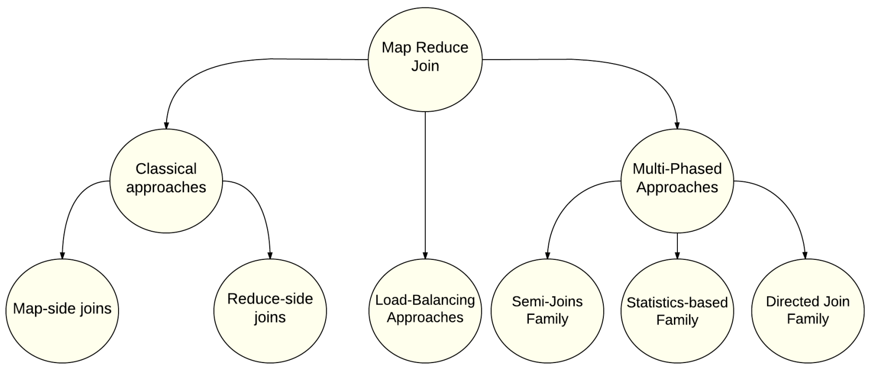

**Fig. 1.** The proposed taxonomy of Map Reduce join algorithms

The first level of our taxonomy contains a set of *Approaches*:

**Classical** which contains the Map/Reduce side taxonomy. These algorithms are mainly deterministic.

**Load-Balancing** which captures the family of multi-way theta-joins. Algorithms in this class are indeterministic.

**Multi-Phased** the techniques in this class require more than one Job to produce final output.

In the table below (Table 1), we list the algorithms found in the literature and categorizing them into our taxonomy.

The following sections provides short overview of chosen join algorithms describing their common use cases.

**Table 1.** Taxonomy of Joins in Map Reduce

| **Name** | **Type** | **Also known as / Similar to** | **Reference** |
|---|---|---|---|
| Repartition Join | Reduce Side Join | General Reduce Side Join, Standard Repartition Join | [7], [18], [9], [17], [5] |
| Improved Repartition Join | Reduce Side Join | Optimized Reduce Side Join, Optimized Repartition Join | [9] |
| Broadcast Join | Map Side Join | Map Side Join, Fragment-Duplicate Join, Fragment-Replicate Join, JDBM Join, Reverse Map Join | [9], [6], [14], [17], [5] |
| Reverse Map Join | Map Side Join | N/A | [14] |
| Map Reduce Cross Join | Map Side Join | N/A | [7] |
| MR Theta Join | Load-Balancing | 1-Bucket-Theta, Strict-Even Join | [13], [16], [19] |
| Semi-Join | Semi-Join Family | Per-Split Semi-Join, MR Join using Bloom Filter | [9], [12], [17] |
| SAND Join | Statistics-based Family | N/A | [10] |
| Directed Join | Directed Join Family | N/A | [9], [7], [6] |

## 4 Repartition Join

Repartition Join is concerned to be the most general approach for solving the joining problem, as it can be easily adapted to work with any number of different input files [6]. The idea of the algorithm is to tag the records with the information about their source, group them by the common joining key and finally compute the actual join. In the figure 2 one can observe the idea of the Repartition Join while the following sections cover the phases in detail.

### 4.1 Map Phase

Mappers responsibility is to identify the source of the record and extract the key. Therefore, if the job is configured so that every input is processed by the same Mapper, some logic may be required to distinguish the record source. If it is not the case, and every distinct input has its own Mapper implementation, class name can be used as a source tag. Nevertheless, this approach may impose some overhead as the class names tend to be quite lengthy. As an emission key, the join attribute from the input record is used. The whole record with additional field containing the source tag is emitted as a value.

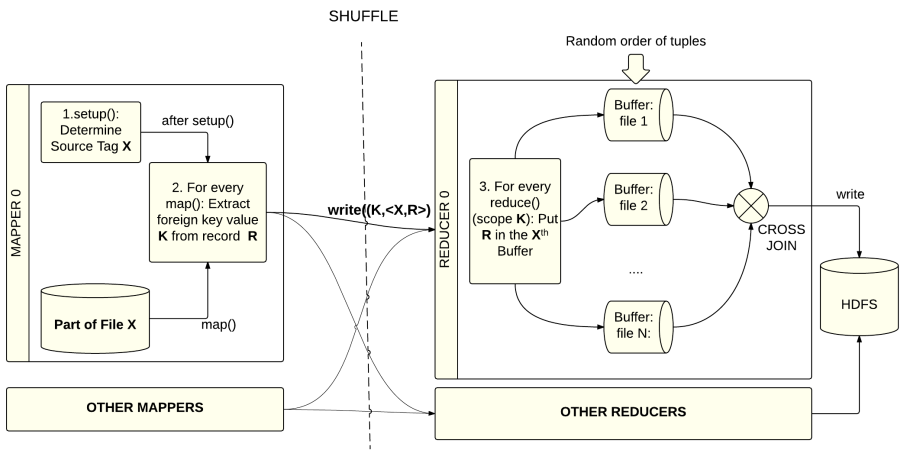

**Fig. 2.** The anatomy of N-Way Repartition Join

## 4.2 Reduce Phase

Reducers need to distinguish the records originating from different sources and produce the final output using any classic joining technique, preferably Nested-Loop Join. As in the current state the records from different files are most likely to come mixed up, we are forced to create as many buffers as the number of input files and defer the production of the final result till every record is buffered.

The main drawback of this solution is that if many records for the given key exist, buffering may lead to *OutOfMemoryException* (OOM), thus one may need to use the non-volatile memory to avoid raising such exception.

## 4.3 One-Shot Multi-Way Join

Although Repartition Join is not the fastest method to calculate join, its simplicity makes it a popular choice among the developers. In addition, its Reduce step can be easily generalized to handle Multi-Way join in one Job, by dynamically creating the buffers using Java *HashMap* collection. The precondition to do this is the fact that join has to be performed on the common key. Such situation is presented on figure 3 where one can notice the source *A* with a join attribute *id* is associated with sources *B1* and *B2* using one-to-many relation, which enables the developer to write one job that joins all sources at once.

If the Standard Repartition Join suffered from OOM exception, the Multi-Way version of it does even more, as the greater number of buffers are required to be maintained.

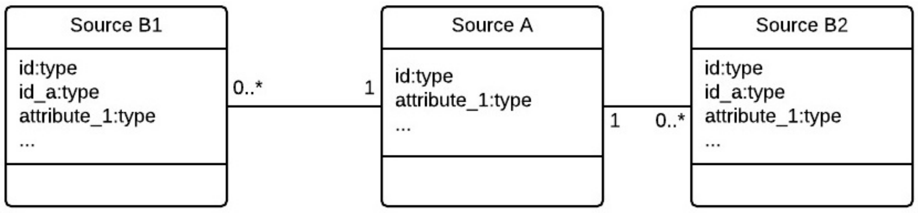

**Fig. 3.** UML diagram presenting the conditions to perform the join on multiple sources at once using Repartition Join

## 5 Improved Repartition Join

As we previously stated, Repartition Join is recognized as a general solution to join problem in Map-Reduce environment. However, it is vulnerable to potential OutOfMemoryExceptions as the Reducers are forced to buffer all the incoming tuples.

The general idea behind Improved Repartition Join is to decrease the memory impact of buffering and enable producing the output as the records from the last data set are read. To minimize the impact on the memory, we wish not to buffer the largest data source. To achieve this, one is required to utilize many Hadoop mechanism, namely: *custom writables*, *Paritioners* and *secondary sorting*.

The execution of the Improved Repartition Join can be seen in the figure 4.

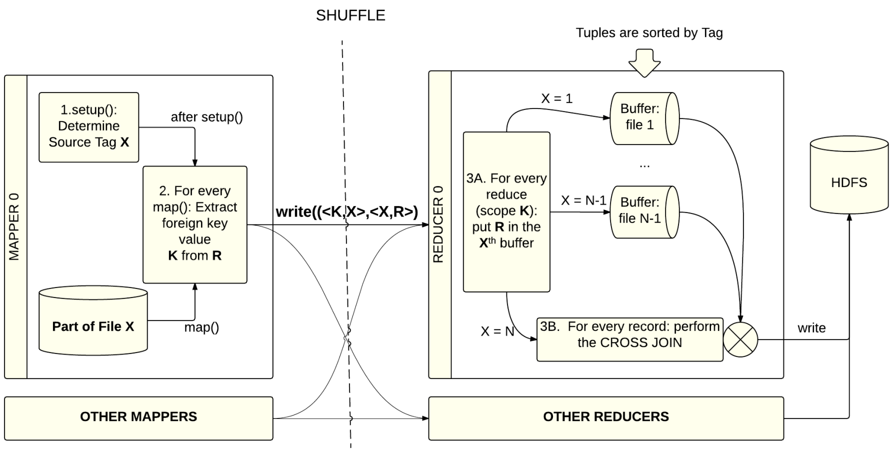
SHUFFLE
Tuples are sorted by Tag
MAPPER 0
1.setup(): Determine Source Tag X
after setup()
2. For every map(): Extract foreign key value K from R
Part of File X
map()
write((<K,X>,<X,R>)
REDUCER 0
X = 1
Buffer: file 1
...
3A. For every reduce (scope K): put R in the X^th buffer
X = N-1
Buffer: file N-1
X = N
3B. For every record: perform the CROSS JOIN
write
HDFS
OTHER MAPPERS
OTHER REDUCERS

**Fig. 4.** The anatomy of N-Way Improved Repartition Join

### 5.1 Preparation

Firstly, we extract the attributes used for joining the record sources, the origin of the data and the record itself. The Mapper should emit the instance of custom writable both for the key and the value. The custom writable serves as a holder of the tag and some payload - attributes used for join for Mapper key and whole record for Mapper value.

Secondly, the default behaviour of *Partitioner* should be overridden so Reducer is determined based on the hash value of source tag and not the hash value of grouping key.

Finally, we provide the custom Grouping Comparator which ensures that on the side of the Reduce records with same payload in the custom writable key are processed in single *reduce()* (the tag is ignored).

As we provide the prerequisites for Improved Repartition Join, we ensure that not only Reducers will posses every record for given attribute used for join, but also that the records will be processed in a predictable way.

### 5.2 Map Phase

Map phase for Improved Repartition Join only consists of retrieving the information about the attributes used for join and emitting the instance of custom writable which serves as a holder of the tag and some value - attributes used for join for Mapper key and whole record for Mapper value.

### 5.3 Reduce Phase

In the Reduce phase the data sorted by the join attribute and the source tag is obtained, therefore we may exploit this fact by not buffering the group of data originating from the last data source. The problem is that there is no mean to automate the process to notify which tag is the last in the obtained data stream. However, in the most common case, which is two-way join, the solution is fairly simple, as we can read the tag from the key parameter of the *reduce* method invocation and in the subsequent calls we may anticipate for the tag change.

### 5.4 Multi-Way Join

To produce the multi-way join output, the developer has to verify that Mappers incorporate the logic of tagging to ensure that the smallest of the data sources will be processed first.

The main problem of multi-way join is that there is no safe mean to calculate how many distinct sources are used to calculate the join. Thus, we do not know if the tag of the currently processed record belongs to the last source. This problem can be easily solved if the number of sources is counted in the job configuration process. Then, we are able to write the auxiliary value to the context and in the reduce phase use it to efficiently process the join.

## 6 Map-Side Joins

The techniques of Mapper side joins are exploiting the *DistributedCache* mechanism of Hadoop API to replicate data sources to the Mappers. This enables them to retrieve the sources as local files, load them into the memory and build the auxiliary hashtable. Finally, the join output is produced as the Mapper reads the input file record by record, which are joined using the built hashtable.

In the figure 5 one can observe the data flow in the Map Side Join.

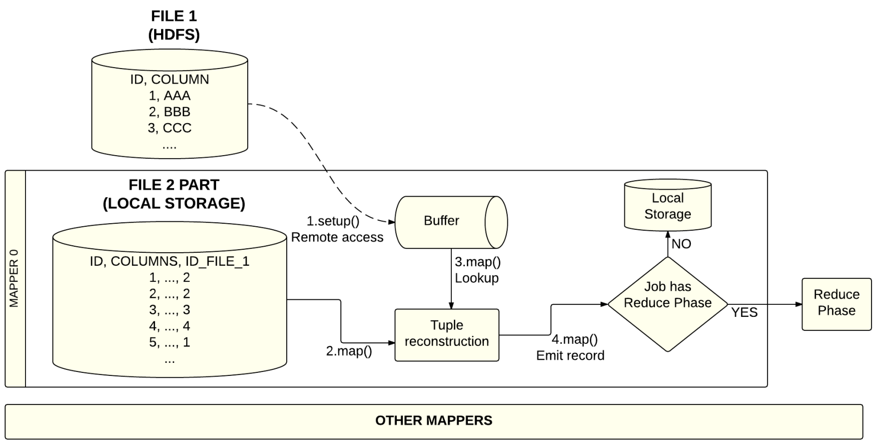

**Fig. 5.** The anatomy of 2-Way Map Side Join

### 6.1 Broadcast join

The most crucial part of the Broadcast join is the proper Job initialization. As we desire to distribute one of the joined sources to all Mappers, we must call proper method of the *DistributedCache* mechanism: *DistributedCache.addLocalFiles(Configuration, String)*. Doing so forces the Hadoop to replicate specified file to all Mappers. Therefore, the two-way Broadcast Join can be considered a single input file Job.

The next step of this approach is processing the file from the local storage to in-memory hash-based structure. This can be effectively implemented during Mapper's *setup(Context)* method, which serves as a constructor, as it is invoked once per Mapper's lifecycle.

After the local file is loaded and hashed in the main memory, one is required to deliver the implementation of the map function which will produce the join output.

Mapper side joins are perfect choice for performing the Star Join given the one source is served as input and rest of the sources are broadcasted via *DistributedCache* mechanism. Therefore, there is no need to perform the job chaining to produce the join output of more than two data sources. As we are able to locally read many different files, one can easily predict that the incorporation of the theta-join conditions is straightforward. What is more, this approach allows the developer to easily produce the cross product of few sources at once.

### 6.2 JDBM join

The execution of standard Map Side Join is threatened by the raise of the OutOfMemory exception. To remove this threat factor, one may use the Java transactional persistence engine library to enable building Java collections limited by the node persistent storage capacity.

Main problem of the JDBM join (and in fact every join exploiting any of the persistent engines) is that it generates additional disk I/O operations which are significantly slower than operating in the memory only. Therefore, if the processed file would fit into the memory, using JDBM Join will require more time to produce the output than standard Map Side Join.

### 6.3 Reverse Map join

The Reverse Map Join [14] method does not remove the risk of raising the OutOfMemory exception, but it lessens the probability of it. As the name suggest, the process of the buffering and producing the final output is somehow reversed. In original Map Side Join we are buffering the broadcasted files and then in Map phase, we emit the join output records. In the Reversed approach however, during the map phase the input data source is buffered and hashed and the *cleanup()* method serves as the phase of emitting the records, as the file obtained from Distributed cache is read line by line and compared.

This approach enables us to predict the amount of the memory required to buffer the records as the raw data should take up the size of the Mapper split, which in most cases is the same as the block in our Hadoop cluster. Another great advantage of this solution is the fact that the emission of the records are somehow deferred, thus the programmer is able to collect some statistics and control the shuffle process. What is even more beneficial is the fact that in the standard implementation of the Reverse Map Join during the buffering phase no disk write operations will occur. Moreover, it is guaranteed that during the write phase, no read operations will be invoked. In the standard Map Side Join, if the output buffer reaches the certain threshold it is spilled to the disk, therefore invocation of the map() may result not only in disk read operation but write also.

## 7 Directed join

Multi-phase join algorithms are designed by chaining more than one Map-Reduce Job. Given the fact that algorithms producing the result in one Job exists, the question rises why one may want to use multi-phase approach. Often the developer posses some knowledge about the processed data which either makes it possible to achieve a shorter execution time or forces the preprocessing as the performance of the 'naive' Map/Reduce side joins may suffer [5].

The most known algorithms of this class are joins using the **bloom filter** which filters the dangling tuples. Similar approach, although using semi-join operation can be found in Map-Reduce Semi Join algorithm and its variation Per-Split Semi Join (both [9]). Also, one may consider the statistic collection as the phase of the algorithm - few such algorithms were proposed, for instance SAND [10].

In next section, we will dwell into the Directed Join - one of the first Multi-phase solution. The approach itself was popular in the early versions of Hadoop not only as its implementation was available in the *contrib.utils.join* package but also as it was considered optimal solution for equi-join problem.

## 7.1 Idea

Not necessarily the algorithm itself, the Directed Join can be considered a set of conditions fulfilling which gives the developer the opportunity to minimize the amount of the data transferred over the network.

The idea behind the Directed Join is simple, as every distinct input file is split into a number of parts. The Job itself works on one file only, as the Mappers determine on which part of the file they are working and are remotely obtaining the adequate part of the second file. Therefore, one may notice that very strict conditions should be met to be able to use this method, as every file should have equal number of parts and same join attributes values sets should be found in the corresponding parts. These constraints are not so strict when one consider that in fact nearly every Hadoop Job fulfils them. That being said, the Directed Join is often preceded by specialized job which groups the data. This preprocessing is the reason why the Directed Join is considered multi-phase join. To partition the data correctly, the preprocessing job should:

– Use the same *setNumReduceTasks()* for every processed file, so that the number of the output files should be equal. In addition, the Reducer should be explicitly set, otherwise Mappers will write the output locally
– The data should be partitioned by the join attributes using the same *Partitioner* class

Additionally, one may use the comparator to sort the files by the values of the join attribute. If the data is sorted, the join can be processed in stream based fashion. If the files are not sorted, the Directed Join still can be implemented, although it will involve buffering.

## 7.2 Performance

As only the files which names contain the same part number are joined it can be easily observed that the number of the input file parts directly influence the performance of the Directed Join. Therefore, one should be aware of the following facts:

– The number of the partitions is limited by the cardinality of the values of the join attributes.
– If the parts are small enough (less than Hadoop *dfs.block.size* parameter), only one Mapper per part will be created, which guarantees that the remote parts are transferred over the network to one node only.

If the data are initially partitioned on the values of the foreign key, the Directed Join algorithm can be perceived as the optimal solution. In case of two-way join when every part is written on one HDFS block, every Mapper is required to perform the number of the remote reads equal to the number of the partitions.

The Directed Join resembles the Broadcast Join, with a notable difference that in the first algorithm only the specific part of the input file is remotely read, while in the latter one, the whole file is obtained.

## 8 Conclusion

Lack of knowledge of Map-Reduce join algorithms often forces the developers to perform the bound-to-be ineffective queries in frameworks like Pig [2] or Hive [1]. Awareness of the limitations of the existing algorithms can contribute to better decisions concerning method to be used. As at current state of art no mechanisms of collecting statistics in Map-Reduce were introduced - so the usefulness of cost-model are limited - the ability to arbitrary choose the suboptimal solution is invaluable.

This paper purpose is to group, categorize and provide the short overview of the plentiful of the Map-Reduce join algorithms which can be found in literature. In recent times, much effort was put to build cost models and optimize the existing join algorithms. The traces of these optimizations can be found in [4], [3] and [17]. Also some progress was made with skew handling in Map-Reduce in general [15] as well as in join algorithms ( [10] and [5]).

Many recent contributions were made to processing the theta-joins by subsetting the cross-join. Algorithms of this class, although break Map-Reduce programming model, are particullary attractive as they are resistant to any abnormal skews. The authors of this article are currently involved in developing optimization leveraging this process. The most known works on the field of theta-join in Map-Reduce are [13], [16] and [19].

## References

1. Apache hive reference. `https://hive.apache.org/`.
2. Apache pig reference. `http://pig.apache.org/`.
3. Pigul A. *Generalized Parallel Join Algorithms and Designing Cost Models.* 2012.
4. Afrati F. N. and Ullman J.D. *Optimizing Joins in a Map-Reduce Environment.* In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110, 2010.
5. Atta F. *Implementation and Analysis of Join Algorithms to handle skew for the Hadoop Map/Reduce Framework.* Master's thesis, University of Edinburgh, 2010.
6. Chandar J. *Join Algorithms using Map/Reduce.* Master's thesis, University of Edinburgh, 2010.
7. Miner D. and Shook A. *MapReduce Design Patterns.* O'Reilly, Beijing, Köln, u.a., 2013. DEBSZ.
8. Dean J. and Ghemawat S. *MapReduce: simplified data processing on large clusters. Magazine Communications of the ACM - 50th anniversary issue: 1958 - 2008*, 51, 2008.
9. Ercegovac V. and Blanas S. *A Comparison of Join Algorithms for Log Processing in MapReduce.* In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 975–986, 2010.

10. Atta F., Viglas S.D., and Niazi S. Sand join - a skew handling join algorithm for google's mapreduce framework. In Multitopic Conference (INMIC), 2011 IEEE 14th International, pages 170–175, Dec 2011.
11. Karloff H. and Suri S. and Vassilvitskii S. *A model of computation for MapReduce*. pages 938–948, 2010.
12. Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim. Join processing using bloom filter in mapreduce. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, RACS '12, pages 100–105, New York, NY, USA, 2012. ACM.
13. Li J. and Wu L. and Zhang C. *Optimizing Theta-Joins in a MapReduce Environment*. *International Journal of Database Theory and Application*, 6, 2013.
14. Gang Luo and Liang Dong. Adaptive join plan generation in hadoop.
15. Balazinska M., Howe B., Kwon Y., and Ren K. *Managing Skew in Hadoop*. *IEEE DATA ENG. BULL., VOL*, 2013.
16. Okcan A. and Riedewald M. *Processing Theta-Joins using MapReduce*. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 949–960, 2011.
17. Palla K. *A Comparative Analysis of Join Algorithms Using the Hadoop Map/Reduce Framework*. Master's thesis, University of Edinburgh, 2009.
18. White T. Hadoop: The Definitive Guide, chapter 8, pages 283–288. O'Reilly, 3rd edition, 2012.
19. Zhang X. and Chen L. and Wang M. *Efficient Multiway Theta-Join Processing Using MapReduce*. In *Proceedings of the VLDB Endowment (PVLDB)*, volume 5 of *11*, pages 1184–1195, 2012.