The Design of the Efficient Theta-Join in Map-Reduce Environment

Maciej Penar and Artur Wilczek

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

Abstract. When analysing the data, the user often may want to perform the join between the input data sources. At first glance, in Map-Reduce programming model, the developer is limited only to equi-joins as they can be easily implemented using the grouping operation. However, some techniques have been developed to leverage the joins using non-equality conditions. In this paper, we propose the enhancement to cross-join based algorithms, like Strict-Even Join, by handling the equality and non-equality conditions separately.

Keywords: Map-Reduce, Hadoop, Join algorithms, Theta join, Inequality join, Big Data

1 Introduction

In recent years Map-Reduce systems have proven their effectiveness to process large volumes of the data. The developers all over the world praise them for the simplicity of expressing *Jobs* - parallel programs run on the Map-Reduce cluster. Therefore, many companies have incorporated open-source version of Google's original Map-Reduce system called Hadoop [1] to leverage their analytic requirements. In such systems the data is often loaded from transactional databases without any pre-processing being done. Thus, the data originating from a single table can span across multiple files. This forces the Map-Reduce developer to express not only some computations leading to the desired final result, but also some logic to perform the join operation which ties together many separate files.

This becomes the good starting point for the Job optimization, as the join operation can be perceived as time-consuming. Whether or not the companies utilize their own cluster, it is always beneficial for Map-Reduce programs to achieve the lowest possible *execution time*. It is even more important when the cloud-based Map-Reduce clusters are concerned.

Some of the existing join algorithms could be in our opinion enhanced to ensure better performance. Our main contributions are as follow:

– We introduce the concept of *pruning* - the enhancement to the existing Map Reduce theta join algorithms.

– We conduct the experiment which confirms the potential drop of the execution time of the join routine in Map Reduce.

The paper is organized as follows: section 2 contains a short overview of Map-Reduce programming model. In section 3 the idea of *Load Balancing Approach* is presented. Chapter 4 contains our contribution to the *Load Balancing Approach*. Experimental evaluation is presented in the section 5. Last section is a short conclusion of the topics covered in this paper.

2 Joins in Map-Reduce

Map-Reduce programming model was designed to leverage and simplify the computations in the distributed environment consisting of the commodity hardware ([7] and [11]). In fact, the philosophy of simplification in the Map-Reduce systems is often the subject of the controversy and is often criticized [8].

The idea of Map-Reduce programming model is as follows: the developer implements two (or in some cases only one) methods/functions used for the data processing - obligatory *map* function and the optional *reduce* function. In so-called *Map* phase, the system calls the *map* function for every record which has been read, so that key/value pair is emitted. If the *reduce* function was delivered, after the *Map* phase has finished, the intermediate *Shuffle* phase starts - which copies over the network the emitted key-value output to the nodes responsible for *Reduce* phase - this happens in such fashion that every record with a common key is placed on a single node. After that, the *Reduce* phase begins and the *reduce* function is applied for every set of the values with a common key. The detailed information about the Hadoop architecture and its anatomy can be found in [18].

Although many join algorithms for Map-Reduce environment were proposed, they always had limited application which varies for every algorithm. Their fundamental problem (which also is a problem in other Map-Reduce Jobs) is that the programming model can group data based only on the equality of the key.

Formally speaking, this makes only two types of join viable in Map-Reduce programming model:

Equi-joins - as they fit into equality based groupings.
Band-joins - as pointed in [16], they can be "naively" expressed if we assume that values of the foreign key are always non-negative integers. Basically, this means transforming the Inequality-join into the Equi-join.

Many methods for solving the equi-joins have been developed with notable examples of: Repartition Join, Improved Repartition Join, Bloom Filtering, Broadcast Join and etc. In general, join algorithms in Map-Reduce are divided into two categories Map-Side Joins, where tuple is reconstructed in Map phase, and Reduce-Side Joins, where tuple is reconstructed in Reduce phase. These algorithms can be found under numerous sources, for instance: [9], [18], [6] and [14]. The effectiveness of these algorithms have been deeply analysed which resulted in many cost models ([4], [3] and [17]) backing up the Map-Reduce job execution.

More unorthodox approaches aim to amortize existing drawbacks of the Map-Reduce Jobs, especially data skews ([13] and [5]). Among these, the SAND Join is worth mentioning [10] as well as joins using the technique called anti-combining [15].

3 Theta Join in Map-Reduce

Calculating the θ-join in the plain Map-Reduce programming model can be considered a difficult task, as no explicit groups can be found at first sight. This is the obstacle which *Load Balancing* approach helps to overcome. The name refers to the fact that each Reducer in the cluster receives similar number of the records for every data source, thus same number of iterations are required to produce the subset of the final cross join. The output of every Reducer is disjunctive to each other, so their union is equivalent to cross product.

This approach can be found in [19], [16] and [12]. The Reader should be aware that this algorithm can be found under many names like: *Strict-Even Theta*, *1-N Bucket Theta*, etc., but for the convenience, we will refer to it as the *Load-Balancing Approach*.

As the algorithm calculate the cross product, it is very convenient to perceive the θ-join as the operation of subsetting on the cross-product given the input condition. That way, the two-way θ-join can be expressed as $S_1 \bowtie_\theta S_2 \equiv \sigma_\theta(S_1 \times S_2)$ where S means input source (i.e. a relation) and θ is a join condition.

Idea

To demonstrate the idea of the *LoadBalancingApproach* we perfom the N-way theta-join on the data sources $S_1, ...S_N$ using the condition θ. One of the parameters of the Map-Reduce Job which can be specified by user is the number of used Reducers R. Before the Job execution, one must establish the number of so-called *partitions* (or fragments [4]) for every input data source - this can be done arbitrarily or be incorporating the cost-based optimization (which will be discussed further). The number of the partitions p_x for x^{th} data source means that data set S_x is divided into p_x disjunctive parts, for example if $p_1 = 3$ then $S_1 = S_{1,1} \cup S_{1,2} \cup S_{1,3}$, so that $S_{1,1} \cap S_{1,2} = \emptyset$, $S_{1,2} \cap S_{1,3} = \emptyset$ and $S_{1,1} \cap S_{1,3} = \emptyset$.

One should be aware that the number of the partitions is constrained by the number of the Reducers, so that $R = \prod_{i=1}^{N} p_i$. For example, if two data source are joined, $R = 8$ and $p_1 = 4$, then $p_2 = 2$.

Every Reducer is assigned to process only a single part of every data source in such way that no other Reducer processes the exact same combination of parts. Then, Reducer uses the received parts to produce the Cross Product of them. Thus, if some Reducer was set up to receive the parts $S_{1,3}, S_{2,2}, ..., S_{N,1}$, the $S_{1,3} \times S_{2,2} \times ... \times S_{N,1}$ will be calculated and no other Reducer will process the same combination. This can be presented graphically as in the figure 1.

Given the facts that: (1)single combination of parts enables to produce subset of the cross-join, (2) every Reducer is assigned the unique partition combination

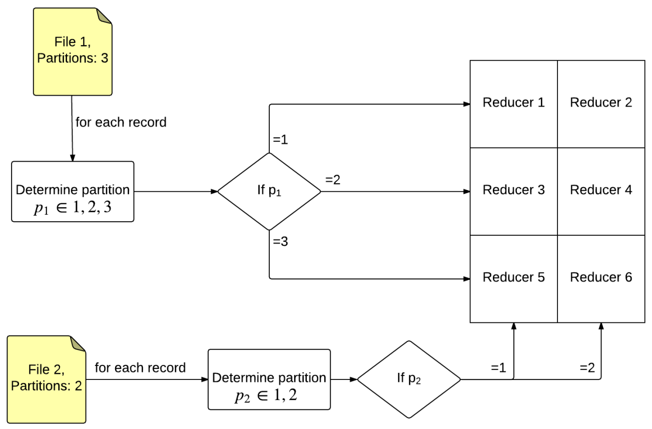

Fig. 1. Idea of Load-Balancing join algorithms for two files with 3 and 2 partitions

and (3) every combination of parts has been assigned to Reducer, thus globally all Reducers perform the Cartesian product.

Two main disadvantages of such approach exist. Firstly, the shuffle phase is more expensive compared to classical Map-Side or Reduce-Side Joins as each record should be replicated to a number of the Reducers which increases the complexity of Shuffle phase. This can be easily measured as the records from the i^{th} source are required to be sent to exactly $\frac{R}{p_i}$ Reducers. Secondly, the algorithm is problem-independent as the developer actually solves the even data distribution which, from some point of view, can be perceived as rewriting the functionality of *Partitioner* and *Mapper* classes.

Cost-Based Optimization

The Load Balancing Approach is often considered in terms of cost-based optimization as one can easily predict the impact of different combinations of partitioning on the Shuffle phase. The most common cost model ties the size of the input file, which can be denoted by F_i - for the i^{th} input file, with the partitioning. The expression is presented below.

$$f(p) = R\sum_{i=1}^{N} \frac{F_i}{p_i} \tag{1}$$

The idea of this cost model is based on the assumption that every single Reducer will receive same amount of data originating from the same file. The problem is that if big input files are concerned, then Shuffle phase may be dominated by Reduce phase cross-join calculation which this model does not take into the account [16].

4 Enhancement

Given the fact that calculating the all-to-all comparisons is difficult and time consuming even for a very small files, our contribution concerns the methods of decreasing the Reducer workload.

We introduce the concept of *pruning* - lowering the complexity of the Reduce phase by skipping the unnecessary comparisons. One may distinguish three methods of pruning:

Equi-join pruning - when data sets are joined using the equality condition between common foreign keys (the foreign key referencing the same data source)

Theta-join pruning - when data sets are joined using any condition between foreign keys

Hybrid pruning - method combining the Equi-join pruning and Theta-join pruning - when data sets are joined using many conditions, one of which is equality between the common foreign keys and second can be any condition

Pruning introduces more invocations of the *reduce* method in the Load Balancing Approach program, as originally every Reducer executes *reduce* only once. More invocations often lower the requirement for the memory, as the scope of some of the buffered records is limited to a single *reduce* which is shorter than in the non-pruned version of the algorithm.

To describe our solution, let us introduce the following notation:

R - index of the Reducer where the record should be sent
F - index of the input file
$Equi$ - foreign key values used for equi-join
$Theta$ - foreign key values used for theta-join

Equi-join pruning

This method requires appending the information about the foreign key during the Map phase. Thus, the record is emitted using the $(R, Equi)$ key. The buffering of the last data set can be avoided by changing the key to $(R, Equi, F)$ and using the additional Comparator forcing the *reduce* on each of the $(R, Equi)$(as in the Improved Repartition Join).

This method does not differ much from the Repartition Join algorithm, apart from the fact that it is resistant to abnormal foreign key distributions in expense of more complex shuffle phase.

Theta-join pruning

Pruning based on theta-join condition is more complicated as no explicit grouping augmenting the join itself exists. As no assumptions can be made which records should be buffered in the memory, in the end one is forced to buffer every record. Therefore, the philosophy of the theta-join pruning differs from the previous method as the *reduce* invocation optimize the process of buffering.

The motivation is simple: many records with same foreign key values may exist, thus one may check if the condition is satisfied only on the distinct foreign key values.

To implement this solution, one is required to emit the records with a following key $(R, Theta, F)$. Then in Java implementation one can use array of Multimaps to handle the buffer: $Multimap < String, String > []\ buffer$, such that $buffer[i]$ is the record buffer for i^{th} data source and the records lists in the buffer are associated with the foreign keys.

In this manner, the join can be computed either in the end of the *reduce* invocation or in the *cleanup* method - in both cases every record has to be buffered. It is worth mentioning though, that during *cleanup* Hadoop will not update the Job progress, which may be inconvenient for the user.

Hybrid pruning

The last optimization incorporates both previously presented so that the scope of buffering is limited only to the invocation of *reduce* - the equality conditions force the groupings - and conditions are checked only on distinct values of foreign keys used for theta join.

The method is complex, as it requires emitting $(R, Equi, Theta)$ and implementing the Grouping Comparator on $(R, Equi)$. Additionally, it may be advisable to attach redundant information about the theta-foreign key to the record - if one does not want to dynamically retrieve this information on the side of the Reducer.

Also, hybrid-pruning can be used to exploit the equi-join conditions when not all of the data sources have common foreign key. In such case, some of the records shall be emitted under the $(R, null, Theta)$ key which should be processed at the first *reduce* invocation. These records should be buffered throughout the whole Reduce phase.

Unfortunately, it is difficult to build the cost model for this method as theta join conditions are evaluated inside the groupings - such selectivity cannot be easily measured without any statistics.

5 Experimental evaluation

The data which was used in the research was generated using the TPC-H data generation tool DBGen. Default behaviour of this tool forces it to generate the data which values are drawn from the uniform distribution. In all experiments, the two-way join is calculated between the two biggest files generated by TPC-H benchmark: *orders.tbl* and *lineitem.tbl* . Originally these tables are tied by one-to-many relationship. As for some of the queries we require compound keys, and the TPC-H model does not have them, we create artificial ones by generating the extra columns based on dates (for query no.3) and by rewriting the values of primary key. Additionally, as the theta-pruning approach efficiency depends on (1) the skewed data and (2) many-to-many relationship between data sources, the original primary key columns have been transformed to present the optimistic and pessimistic use cases.

As a result, we may distinguish four data sets in our experiment: 1-N (one-to-many relationship between data sets), 1-1(one-to-one relationship between data sets), Zipf-0.5 (foreign keys drawn from the Zipf distribution with exponent equal to 0.5) and Zipf-1(foreign keys drawn from the Zipf distribution with exponent equal to 1).

The test was performed using the Amazon Elastic MapReduce cluster. The master node was running on m1.large instance while the four worker nodes were running on m1.medium. The detailed specification of the Amazon Elastic Compute Cloud instances can be found on the Amazon site [2]. All nodes were running on Hadoop 2.4.0 with the default configuration (The configuration for the worker node can be seen in the Table 1).

Table 1. Configuration of Amazon m1.medium node

Type	General Purpose
Storage space	410 GB
Cores	1
Memory	3.75 GB
Network performance	Moderate

5.1 Assumptions

The experiment measuring the performance of the Load Balancing approach was designed with the following assumptions:

1. The following approaches were compared: joins using the Hive framework, joins using the Pig framework, Standard Cross Join, Load Balancing Approach with the equi-pruning and the hybrid-pruning
2. In case of the queries (1-3) the Scale Factor was set to 1 ($6 * 10^6$ records versus $1,5 * 10^6$ records)
3. In case of the query 4 the Scale Factor was set to 0.01 due to the compute intensive nature of the all-to-all comparisons
4. Every test case was repeated three times and the average of the measured execution time was calculated
5. The number of the Reducers and the partitioning were established using the optimizer, with the maximum number of the Reducers was 4
6. The Standard Cross Join, Pig queries and Improved Repartition Join had the number of the Reducers set to 4

7. The single test case was interrupted after 15 minutes

Additionally, the case of the equi-join was compared with the Improved Repartition Join and Standard Cross Join implementation was based on *MapReduce Design Patterns* [14].

5.2 Evaluation

In the chart for the query no.1 (fig. 2) we may observe that Improved Repartition Join algorithm topped in almost every case except from the data set $1-1$. This may be contrary to ones expectations as the data skew in data sets $Zipf - 0.5$ and $Zipf - 1$ should force some Reducers to finish up later. The problem of achieving such observation is that the data skew may be amortized by the good hashing function. In every case of equi-join, our solution, which is Load Balancing approach with hybrid-prune, finished as a runner-up. This is interesting phenomenon as both in equi-prune and hybrid-prune approaches no inequality condition was specified, thus one may think that the execution time of these approaches would be the same - or in favour of equi-prune.

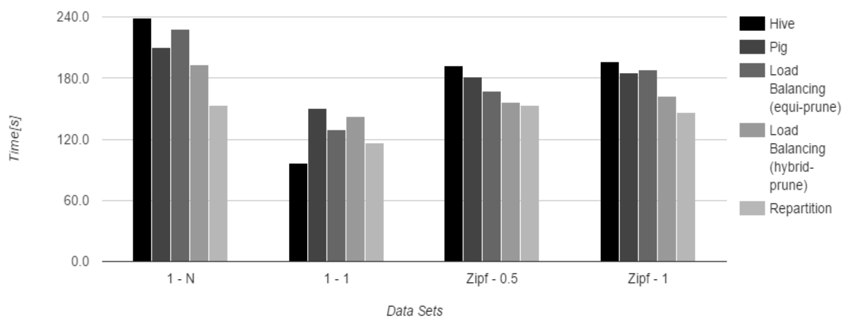

Fig. 2. The execution time for the query no.1: equality condition

The case of query no.2 (fig. 3) proved that the hybrid-pruning imposes some overhead which may slow the execution time. This can be observed for the data set $Zipf - 1$ where evaluating the inequality conditions on the attributes with skewed values distribution does not provide any boost in the execution time. What is more, the standard Load Balancing approach with equi-prune achieves better results, although the differences in execution time between hybrid and equi pruning are marginal. Nonetheless, due to the more complex implementation of hybrid-pruning technique, the standard approach can be considered better.

Next case does not vary much from the second query (fig. 4). The main difference is that the inequality condition is expressed using the User Defined Function (UDF) - comparing if two dates are within the same month. This implies that

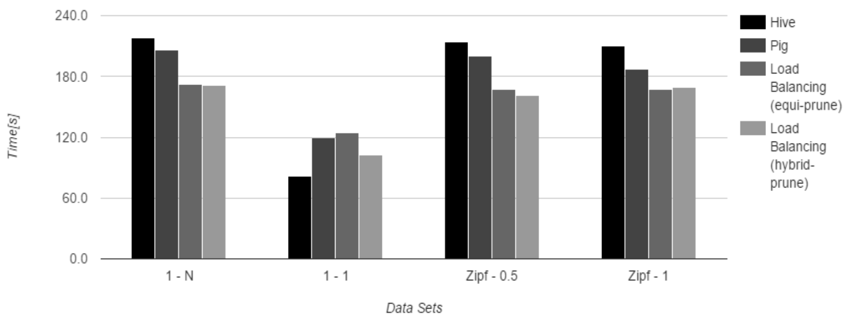

Fig. 3. The execution time for the query no.2: equality + inequality condition

cost of checking it is much higher than in the previous cases, therefore we expect that our method which evaluates the condition only for unique combinations of foreign keys will achieve the best results.

The results showed the poor performance of the Pig as opposed to other solutions. Again our method was the best in cases when data sets $1-N$, $Zipf-0.5$ and $Zipf-1$ were processed.

The last query is expressed using inequality conditions only and does not fit well in the Map-Reduce programming model as the data cannot be grouped to ease the execution. In HiveQL such queries has to be expressed using the *CROSS JOIN* as *JOIN* syntax allows only equality conditions to be passed.

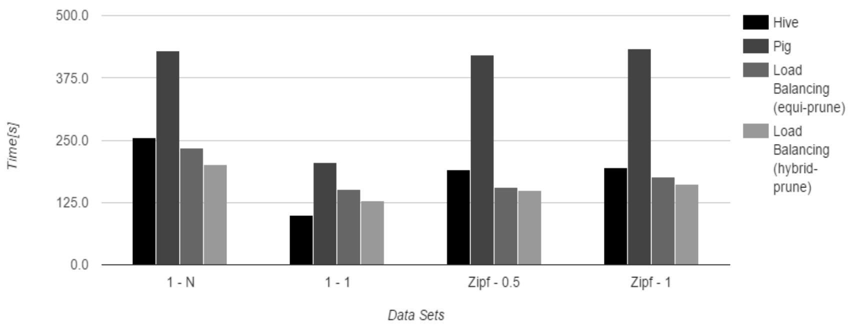

Fig. 4. The execution time for the query no.3: equality + inequality condition based on User Defined Function

It should be noted that queries with only inequality conditions are the main reason behind the Load Balancing algorithms, thus this is the case where these approaches are expected to shine. Firstly, as the complexity of such queries tend to grow exponentially we run the test on a small volume with Scale Factor equal to 0.01. Therefore, our data sets contains $6 * 10^4$ records and $1,5 * 10^4$ records and requires performing $9 * 10^8$ iterations in total. The results for the aforementioned volume is presented in the figure 5.

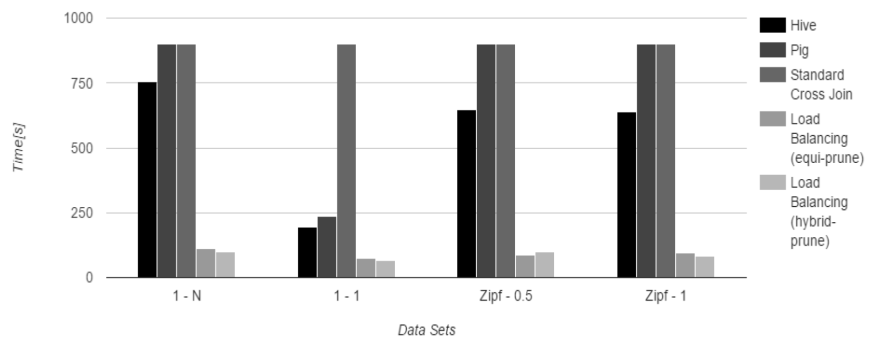

Fig. 5. The execution time for the query no.4: inequality join (data volume 0.01)

Firstly, one should note that in any case the Standard Cross Join had not finished the execution. The same can be said about the Pig framework, as only in case of processing the $1 - 1$ data set it managed to finish the execution. Although the Hive finished in every scenario, its performance is poor comparing to Load Balancing approaches and comparing these is difficult due to the scale imposed by the other algorithms. Nonetheless, one can see that query run on the $Zipf - 0.5$ data set was calculated faster when using the equi-pruning technique. In order to examine the Load Balancing approaches more deeply, we increased the data volume to 0.025, thus the problem on the side of the Reducer scaled by the factor of 6.25, The evaluation of this case for the Load Balancing algorithms is presented in the figure 6.

The results of this experiment show that Load Balancing algorithm with hybrid pruning achieves better results than its counterpart without any pruning (as in this case equi-pruning had no effect) when processing joins on the data sources in which the values of the foreign keys reappear.

6 Conclusion

Many research was conducted concerning join in the Map-Reduce programming model. Recently, much effort was put into solving the inequality joins. This resulted in finding out the algorithms like Strict-Even Join [12] or 1-Bucket Theta [16]. Although the approach seems to break the programming model, it is easy to implement and provide a general approach for calculating joins.

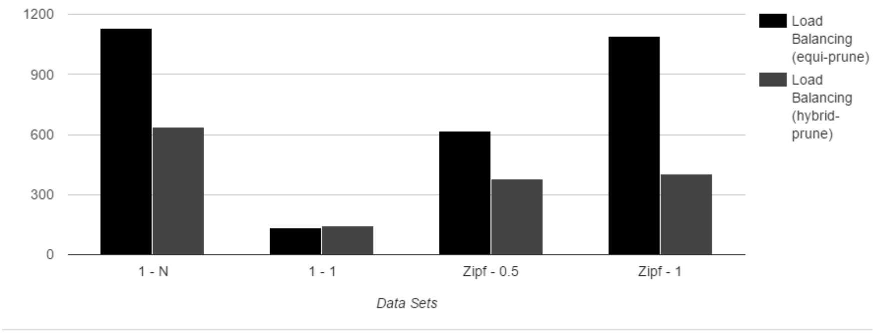

Fig. 6. The execution time for the query no.4: inequality join (data volume 0.025)

We contribute with a modification to existing algorithms, also providing the various emission key formats which leverage Job ability to skip unnecessary calculations. The further research may include building the cost models which will estimate the execution cost. The problem with such estimations is that they are heavily depending on collected statistics. This can make them hard to implement as developers are often reluctant to execute additional computations beside the main Job. What makes the situation even more challenging is the fact that data processed by Map-Reduce systems are often sorted, thus simple remote streaming of top N records can result in the far-fetched estimations.

As our research was conducted on a relatively small cluster and data volume, additional test could be run using the real-world queries to back up obtained results. Other issue which can be tackled is extending the cost models and algorithms for other relational algebra operators - as operators may influence each other, combining them may require new cost models.

References

1. Apache hadoop reference. `http://hadoop.apache.org/`.
2. Easy amazon ec2 instance comparison. `http://www.ec2instances.info/`.
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. Dean J. and Ghemawat S. *MapReduce: simplified data processing on large clusters. Magazine Communications of the ACM - 50th anniversary issue: 1958 - 2008*, 51, 2008.

8. Dewitt D.J. and Stonebraker M. *Map-Reduce: A major step backwards.* `http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html`.
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. Li J. and Wu L. and Zhang C. *Optimizing Theta-Joins in a MapReduce Environment. International Journal of Database Theory and Application*, 6, 2013.
13. Bamha M., Hassan A.H., and Loulergue F. Handling data-skew effects in join operations using mapreduce. In Journées nationales du GdR GPL, Paris, France, June 2014.
14. Miner D. and Shook A. MapReduce Design Patterns : [building effective algorithms and analytics for Hadoop and other systems]. O'Reilly, Beijing, Kln, u.a., 2013. DEBSZ.
15. Alper Okcan and Mirek Riedewald. Anti-combining for mapreduce. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 839–850, New York, NY, USA, 2014. ACM.
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. 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.