NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Constraint Satisfaction Problems in Hadoop MapReduce

Emmanouil N. Ntoulias
Kyriakos Vlasios E. Tharrouniatis

Supervisor : Panagiotis Stamatopoulos, Assistant Professor

ATHENS

OCTOBER 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Προβλήματα Ικανοποίησης Περιορισμών στο Hadoop MapReduce

Εμμανουήλ Ν. Ντούλιας
Κυριάκος Βλάσιος Ε. Θαρρουνιάτης

Επιβλέπων : Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2016

BSc THESIS

Constraint Satisfaction Problems in Hadoop MapReduce

Emmanouil N. Ntoulias
S.N.: 1115200900162

Kyriakos Vlasios E. Tharrouniatis
S.N.: 1115200500250

Supervisor : Panagiotis Stamatopoulos, Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Προβλήματα Ικανοποίησης Περιορισμών στο Hadoop MapReduce

Εμμανουήλ Ν. Ντούλιας
Α.Μ.: 1115200900162

Κυριάκος Βλάσιος Ε. Θαρρουνιάτης
Α.Μ.: 1115200500250

Επιβλέπων : Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής

ABSTRACT

In this thesis we examine the effectiveness of using Hadoop MapReduce join algorithms to solve Constraint Satisfaction Problems. We start by presenting the Map Reduce framework and continue by making a brief summary of the CSPs. We take advantage of the fact that CSPs and database techniques overlap, by modeling a CSP as a database schema. We describe some of the join algorithms and then use the aforementioned schema as input to them. Some modification and preprocessing is done to these algorithms to support the specifications of CSPs as joins. We finally use them to conduct a set of experiments and conclude that it is not effective to use the map reduce framework for CSPs. One suggestion is to remake the experiments on a more suitable environment, i.e. a better cluster, because the one that was used is proven to be inefficient.

SUBJECT AREA: Big Data, Artificial Intelligence

KEYWORDS: Hadoop, MapReduce, join, Constraint Satisfaction Problem (CSP)

ΠΕΡΙΛΗΨΗ

Σκοπός της παρούσας πτυχιακής εργασίας είναι να εξετάσουμε την αποτελεσματικότητα των αλγορίθμων ένωσης πινάκων μέσω της τεχνολογίας Hadoop Map Reduce για την επίλυση Προβλημάτων Ικανοποιήσης Περιορισμών (ΠΙΠ). Ξεκινάμε παρουσιάζοντας το πλαίσιο Map Reduce και συνεχίζουμε συνοψίζοντας τα ΠΙΠ. Εκμεταλλευόμαστε το γεγονός ότι τα ΠΙΠ υπερκαλύπτονται από τις τεχνικές βάσεων δεδομένων, όπως η μοντελοποίηση ενός ΠΙΠ ως ένα σχήμα βάσης δεδομένων. Περιγράφουμε μερικούς από τους ήδη υπάρχοντες αλγόριθμους ένωσης πινάκων και χρησιμοποιούμε το προαναφερθέν σχήμα ως είσοδο σε αυτούς. Τροποποιούμε τους αλγορίθμους αυτούς ώστε να υποστηρίζουν τα ΠΙΠ ως σχήματα βάσεων δεδομένων. Τέλος, πραγματοποιούμε μια σειρά από πειράματα και συμπεραίνουμε ότι δεν είναι ιδανικό να χρησιμοποιήσουμε το πλαίσιο Map Reduce για επίλυση ΠΙΠ.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μεγάλα Δεδομένα, Τεχνητή Νοημοσύνη

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Hadoop, Map Reduce, ένωση πινάκων, Προβλήματα Ικανοποίησης Περιορισμών

To Hector and Liza

AKNOWLEDGMENTS

We would like to thank our supervisor, Dr. Panagiotis Stamatopoulos, for guiding us through this project.

We would also like to thank Ph.D. Candidate, Nikolaos Pothitos, for setting up the cluster and assisting us throughout the experiments.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Constraint programming is a widely known technique used for solving combinatorial search problems in fields such as operations research and artificial intelligence. For years multiple strategies have been developed in an effort to optimize the search process of constraint programming. A lot of those strategies are programmed for single machines. In this thesis we try to develop a CSP solving strategy by using a cluster of machines within the Hadoop Map Reduce framework. One of the capabilities of this framework is to perform joins between large datasets. So, by modeling a CSP as a join we are able to perform such algorithms and solve our problem. The idea might not be as sophisticated as the already existing constraint programming systems that solve CSPs, but it hopes to take advantage of the magnitude of computer power a cluster offers to do the same job. This obviously means that results of this approach vary depending on the cluster one uses.

The structure of this thesis is as follows. In the second chapter we write an extended overview of the Hadoop Map Reduce framework and its capabilities. In the third chapter we take a glimpse of Constraint Satisfaction Problems and some of the already existing solving techniques, which is important to have as a point of comparison. Then we make a connection between the CSPs and databases by modelling the CSP as a join. Such transformation is important for the next chapter, in which we use such database schemas as input to the jobs. A few of the already existing map reduce join algorithms [1] are presented in chapter 4, albeit there are some modifications performed on them to match the needs of a CSP problem. In our research we first tried to develop algorithms such as the one-shot-join that perform the join in only one job. We quickly came to the conclusion that no such technique is possible when it comes to CSPs because CSPs have every variable as part of the join key. Instead, all of the algorithms that are presented here are of cascaded nature. In the fifth chapter we conduct a set of experiments based on the algorithms presented before. The conclusions are presented in the final chapter, chapter six.

2. MAPREDUCE AND HADOOP

2.1 What is MapReduce

"MapReduce is a programming model and an associated implementation for processing and generating large data sets". [2]

It is a framework [3] for writing applications that process large amounts of data. The term MapReduce refers to two separate and distinct tasks. First comes the map job then the reduce job as the sequence of the name - MapReduce implies.

"As an analogy, you can think of map and reduce tasks as the way a census was conducted in Roman times, where the census bureau would dispatch its people to each city in the empire. Each census taker in each city would be tasked to count the number of people in that city and then return their results to the capital city. There, the results from each city would be reduced to a single count (sum of all cities) to determine the overall population of the empire. This mapping of people to cities, in parallel, and then combining the results (reducing) is much more efficient than sending a single person to count every person in the empire in a serial fashion." [3]

Programs written in MapReduce are automatically parallelized: programmers do not need to be concerned about the implementation details of parallel processing. They only should override map and reduce functions. The **map** function processes a key/value pair to generate a set of intermediate key/value pairs. The **reduce** function then merges all intermediate values associated with the same intermediate key. This model is inspired by the map and reduce functions commonly used in functional programming [2]. Map's and reducer's functions domains and ranges are as shown below:

map (k1,v1) → list(k2,v2)

reduce (k2,list(v2)) → list(v2)

The domain of k2 is usually extracted from v1. The intermediate keys and values are from the same domain as the output keys and values. The **list(k2, v2)** to **(k2,list(v2))** transformation is part of the shuffle process which takes place in-between the map and reduce phase.

What's remarkable is the code written inside **map** and **reduce** functions as well as the Driver code (whatever code is in the main function configuring the MapReduce job) doesn't need to change at all to be migrated and executed from a single machine (develop and debug) to a Hadoop (in our case) cluster.

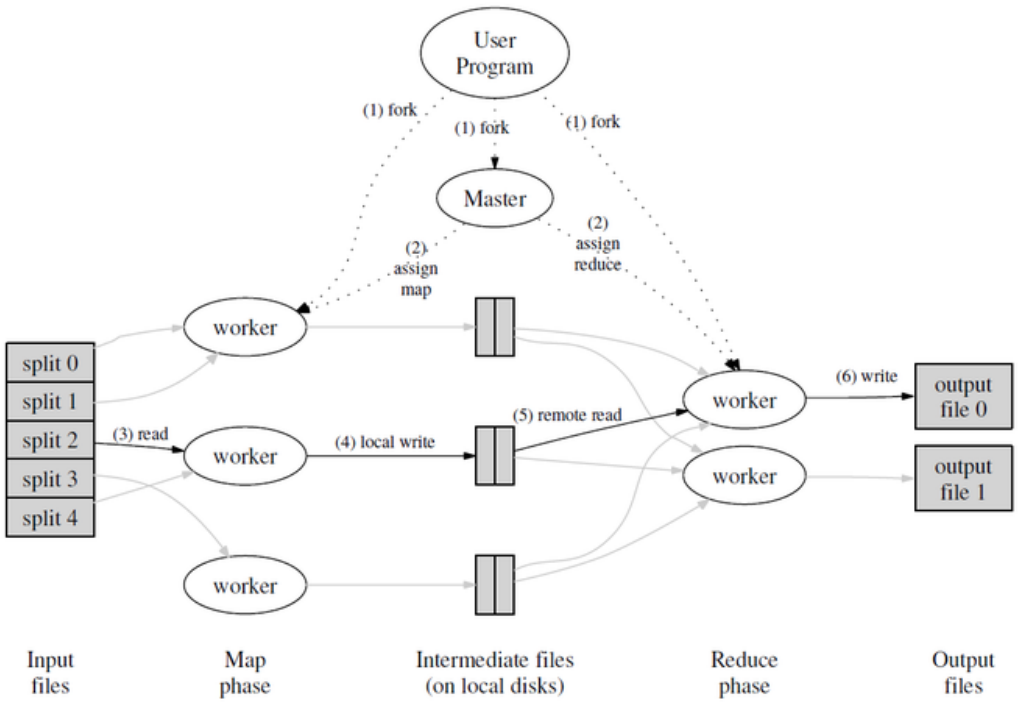

Figure 1 - Execution overview [2]

2.2 MapReduce Execution Overview

When the user program calls the MapReduce function, the following sequence of actions(reproduced here from [2] with a few extensions):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). Practically every one of these pieces will be assigned to a mapper. It then starts up many copies of the program on a cluster of machines. We noticed, on the cluster we were given, that, every time at the beginning of a MapReduce execution there was a noticeable delay and that should be the reason why.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-

defined map function. Hadoop wise, these key/value pairs refer to the offset of the current line read from the beginning of the file/the entire line in String form. The intermediate key/value pairs produced by the map function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. Default Partitioner function can be overridden by the user for allowing custom partitioning.The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. Default Comparator will decide how will this grouping be done and once more can be overridden. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

2.3 MapReduce Example

The following example will help in understanding MapReduce further. Consider we have numbers in a large collection of documents. Lets try to implement the MapReduce framework to solve a somewhat FizzBuzz problem:

1. If a number is dividable by 5 and 3, list them as FizzBuzz.
2. If a number is dividable by 5, list them as Fizz.
3. If a number is dividable by 3, list them as Buzz.
4. Numbers that do not belong to any of the previous sets, list those as Rest.

Given below are the **map** and **reduce** functions in pseudocode:

```
map(String key, String values){
    //key     : offset from the beginning of the file
    //values : assume we transform values into a list of integers
    for each i in values :
        if (i%5 == 0 && i%3 == 0)
            EmitIntermediate("FizzBuzz", i)
        else if (i%5 == 0)
            EmitIntermediate("Fizz", i)
        else if (i%3 == 0)
            EmitIntermediate("Buzz", i)
        else
            EmitIntermediate("Rest", i)
}

reduce(String key, Iterator values){
    //key     : "FizzBuzz" or "Fizz" or "Buzz"
    //values : a list of numbers
    String nums = ""
    for each v in values
        nums = nums + "," + v.toString()
    Emit(key, nums)
}
```

Given the list of numbers below:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

The output would be like:

FizzBuzz 15

Fizz 5,10

Buzz 3,6,9,12

Rest 2,4,7,8,11,13,14

Lets have a quick look on the nature of the problem. One could run through the array forward or backwards and get the same result. What's more he/she could pick random numbers from the list, and still get the same result. In problems like this, doing something on some data doesn't affect the rest of the data and vice versa.In fact if we had two CPUs, we could split the input having each CPU process half of the elements which means a **map** phase twice as fast.

2.4 Hadoop and HDFS

2.4.1 What is Hadoop

Hadoop (also known as Apache Hadoop) [4] is a framework that allows the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

The core [5] of Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part called MapReduce. Every node inside the Hadoop cluster has its own data. The map code is distributed to every node. Thus, every node process its own data locally. One problem would be the overhead, sending data (pairs of key/values as we saw) across the network to the reducers (shuffle phase). However, Hadoop has its solution to reduce this overhead via the Combiner class (discussed in chapter 2.4.5).

The term Hadoop has come to refer not just to the base modules above, but also to the ecosystem [5], or collection of additional software packages that can be installed on top of or alongside Hadoop like Apache Pig, Apache Hive, Apache Spark and more. (for details on the packages see [4], [5])

2.4.2 HDFS

Data in a Hadoop cluster is broken down into smaller pieces (called blocks) and distributed throughout the cluster. HDFS is a file system that is designed for MapReduce jobs that process those pieces and write their output. Data must be available and Hadoop must be reliable in case of a node failure. And that is exactly why data is replicated to multiple nodes. As long as at least one replica of a data chunk is available, the consumer of that data will not know of storage server failures. [6]

2.4.3 The Hadoop Cluster

The main components of a Hadoop cluster are:

ResourceManager : master that arbitrates all the available cluster resources and thus helps manage the distributed applications.

NameNode: It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It does not store the data of these files itself. Manages the filesystem metadata.

NodeManagers: take instructions from the ResourceManager and manage resources available on a single node.

DataNode: stores the actual data. More than one DataNodes will have replicated Data as we mentioned already. DataNode instances can communicate to each other, which is what they do when they are replicating data.

2.4.4 Mapper

The mapper class [15] is one of the two classes the user almost always overrides in order to execute an MR process. Its job is to map input key/value pairs into a set of intermediate key/value pairs. The input pairs don't have to share the same types with the output pairs.

When the execution starts the framework spawns one map task for each input split the input format has generated. It is important to not confuse the map task with the map function of the mapper. The first is a process that executes the whole mapper class, the second is a method of the mapper class. The map function is called by the framework

as many times as the number of input records of an input split and the one responsible for creating the intermediate records.

Additionally, there's the setup method that is called once before the iteration of map function, usually to perform some preprocessing. There is also the cleanup method that is called after the iteration of the map function. All these function are called within the run method of the Mapper, which can also be overridden for more advanced handling of the framework.

```
public void run(Context context) throws IOException, InterruptedException {
  setup(context);
  try {
    while (context.nextKeyValue()) {
      map(context.getCurrentKey(), context.getCurrentValue(), context);
    }
  } finally {
    cleanup(context);
  }
}
```

Table 1 - Mapper run method

2.4.5 Reducer

The reducer class [15] is the other class the user almost always overrides in order to execute an MR process. Reducer receives one or more sets of intermediate records which share a key and then reduces those values. The reduction is done by the reduce function of the class.

As with the mapper, one should not confuse the reduce function with the reducer tasks that are spawned. The number of reducers tasks is explicitly stated by the user via the Job.setNumReduceTasks(int) method, while the number of times the reduce method will be called throughout the whole job is equal to the number of unique keys that exists in the intermediate records. Typically, a reducer task will call the reduce function one or more times if it has been assigned with at least one key or it may not call it at all if the number of Reducer tasks is larger than the number of unique keys.

Again, there are also the setup and the cleanup methods of the reducer that the user can override for his own preferences. These methods are called in the run method of the reducer before and after the reduce function similarly to the mapper.

```
public void run(Context context) throws IOException, InterruptedException {
  setup(context);
  try {
    while (context.nextKey()) {
      reduce(context.getCurrentKey(), context.getValues(), context);
```

```
      // If a back up store is used, reset it
      Iterator<VALUEIN> iter = context.getValues().iterator();
      if(iter instanceof ReduceContext.ValueIterator) {
        ((ReduceContext.ValueIterator<VALUEIN>)iter).resetBackupStore();
      }
    }
  } finally {
    cleanup(context);
  }
}
```

Table 2 - Reducer run method

Each reducer task writes one part file to the output. So, if we have N reducer tasks there will be N part files to the output.

2.4.6 Combiner

The combiner class is an optional class that the user specifies if he wants to use it. It takes place just after the mapper and in the same machine with it. Essentially it is a local reducer and hence overrides the reducer implementation. It takes as input the intermediate records of a single mapper and reduces them before they are sent across the network. For this its key/value input types must match the key/value output types of the mapper and its key/value output types the input types of the reducer. The reason the combiner class exists is to minimize the records and decrease the load across the network. Unfortunately we were not able to utilize this class in our thesis.

2.4.7 Input Split and Input Format

An Input split [15] represents the data to be processed by a single mapper. It presents a byte-oriented view of the input. The default size of a split is usually 64 MB. The user has the option to modify the size depending on his needs.

Input Format is the class responsible for splitting a file into the input splits described above. If some files are too small the input format won't merge them together and hence the number of input splits in a job is even or great to the number of input files. The input format is also responsible for providing the RecordReader implementation which is used to convert the byte-oriented view of the input split into a record oriented view.

The default input format is the text input format. For every input split of a file it creates records with input key an offset to the start of a line and input value the line itself. Text input format is the one we will override in our own algorithms.

2.4.8 Shuffle Phase

With the term shuffle we mean the whole process from the point where a mapper produces output to where a reducer consumes input [16]. During this period the intermediate key/value pairs are transferred from the mapper to the reducer, but there's more than that happening.

First, there's the partitioner class that takes the output pairs of a mapper and specifies in which reducer each one is going to end up. This is done by performing a hash function on the key of every pair. At the same time a sorting phase takes place on these output pairs based on the key. If there's a combiner specified then the sorted output will be processed by the combiner before being sent over to the reducers.

When the intermediate records reach the reducers, each reducer task receives records from multiple mappers because the keys are spread within the input splits. A second sort on the key takes place that merges all those records properly. The reduce methods can now be safely started.

Note that during the sorting phases of Hadoop, the comparator class is very important since it is the one makes the comparison between two values happen.

In the image below the whole shuffling process is summarized.

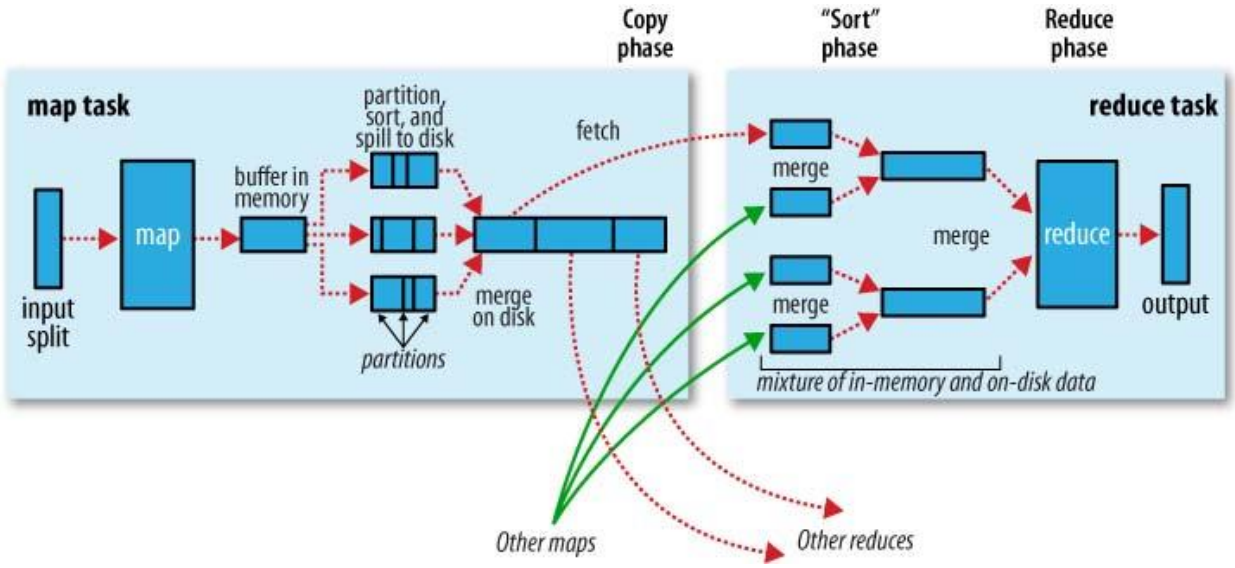

Figure 2 - Shuffle Phase

2.4.9 Job

The job class [15] is responsible for describing the parameters of a map reduce job to the framework so that it can be executed properly. This customization takes place on the driver code (the code of the main class of our program). The classes such as input format, mapper, reducer, combiner, comparator and partitioner that we described above are all defined here by an instance of the job class. After that the mr job starts running in the cluster while the code in the main function waits for its completion.

One of the assets of the job class is that it can, with the help of the configuration class, create user defined parameters and pass them to the mappers and the reducers. These user defined parameters can be used during the execution of the mr job.

3. CONSTRAINT SATISFACTION PROBLEMS

3.1 Definition

According to F Rossi, P Van Beek and T Walsh [9] a Constraint Satisfaction Problem (CSP) is defined as follows. A CSP P is a triple (V, D, C), where V is an n-tuple of variables V = {v_1, v_2, …, v_n}, D is a corresponding n-tuple of domains D = {D_1, D_2, …, D_n} such that $v_i \in D_i$ and C is a t-tuple of constraints C = {C_1, C_2, …, C_t}. A constraint C_j is a pair {R_{Sj}, S_j} where S_j is a subset of V and R_{Sj} is a relation on the variables in Sj. In other words, R_j is a subset of the Cartesian product of the domains of the variables in S_j. A solution to the CSP P is an n-tuple A = {a_1, a_2, ..., a_n} where $a_i \in D_i$ and each C_j is satisfied in that R_{Sj}. In a given task one may be required to find the set of all solutions, sol(P).

A simpler way to define a CSP is to describe it as “a problem where one has to find a value for a (finite) set of variables satisfying a (finite) set of constraints.” [10]

3.2 CSP Example

One of the most well-known CSP is the n-queens problem [11]. The problem by itself isn’t of major importance but its simplicity makes it a good introduction to constraint programming [10].

Assuming we have n queens, the goal of the problem is to place the queens on an n×n chess board so that no queen can attack any other queen on the board. A queen in chess can move an infinite amount of squares either vertically, horizontally or diagonally. In the 4 queens version of the problems there are 2 solutions as seen in the tables below.

Table 3 - Queens 4 solutions

	Q1		
			Q2
Q3			
		Q4	

		Q2	
Q1			
			Q3
	Q4		

Any other way of placing the queens results in conflict with the constraints of the problem. For example the following board is not a solution to the problem since queens 2 and 4 can attack each other.

Table 4 - 4 Queens Invalid

	Q1		
			Q2
Q3			
			Q4

In order to model the above problem according to our CSP definition we define the 4 queens as the 4-tuple of the variables V= {Q1, Q2, Q3, Q4} – one queen for every row. The 4-tuple of the domains D = {D_1, D_2, D_3, D_4} is defined such that all domains are equal $D_1 = D_2 = D_3 = D_4$. Each D_i consists of values from 1 to 4 and each number represents a column that the queen can be placed on. The number of the constraints is the number of the unique queen pairs because the values that each queen takes are affected by where the other queens are placed. This number is effectively n*(n-1)/2, in our case 4*(4-1)/2 = 6 constraints. We hence define a 6-tuple for the constraints C = {C1, C2, C3, C4, C5, C6}. Each constraint $C_j = \{R_{Sj}, S_j\}$ has a unique pair of queens S_j = {Qx, Qy} and a relation R_{Sj} over this pair such that an attack between those two queens either vertically, horizontally or diagonally is not possible.

3.3 Solving a CSP

One of the two classes of strategy for solving constraint satisfaction problems is the systematic search strategies [10]. The other is the repair strategies but we will focus on the first one on this paper.

The basic idea of systematic search strategies is to assign values to the variables one by one. The values come from the corresponding domain of every variable. If a value that we've just assigned doesn't break any of the constraints, we move on to the

next variable. If, however it does break one of the constraints we choose another value until a valid one is picked. If there aren't any values left to pick for this variable we remove the value that we chose for the previous variable and continue searching for the previous variable. This is the simplest form of a process known as backtracking and it continues until all possible combinations of values have been picked if we want to find all the solutions of the problem. Alternatively, we can stop on the first solution we find, unless there are not any. In that case the problem is unsatisfiable.

In order to improve the above method, we introduce the concept of lookahead [10]. Instead of running into a dead end, i.e. all values of a variable that we try to assign go against the constraints, we can make extra analysis during previous stages of the algorithm to prevent the dead end beforehand. For example, the simplest thing to do is whenever we assign a value to a variable we invalidate all the values of all variables that are affected by this assignment - because of constraints. If a variable is then left with an empty domain, it means that we have already found the dead end and we have just saved time that we would have otherwise spent assigning to other variables before reaching the dead-end variable. All the above is the basic idea of the forward checking strategy [12] which of course can become a lot more complicated. What's important is to maintain a balance between the effort trying to analyze every situation and the potential gain from such analysis.

Another improvement we can make for the backtracking strategy is the first fail principle [10] [12]. Up until now we never mentioned the order in which we decide to assign variables, which in fact can greatly affect the efficiency of a search. The concept of first fail is to first try to assign variables with the smallest domain. Remember that when using the forward checking strategy described above we are essentially diminishing the domain of a variable during the execution of the algorithm which is why first fail works very well with forward checking. For example, assuming there is a variable with only one value available and we choose to assign it first, one saves time by automatically discarding all the values of all variables that are in conflict with that assignment and in case of another assigning order these values would have been assigned first and lead us to a dead end.

3.4 Modeling a Join as a CSP

Provided that there is a CSP P = {V, D, C}, we can map it into a database schema the following way [13]:

Each variable in V is represented by an attribute (column). Since however we are referring to equi-joins one variable corresponds to all attributes with the same name regardless of the relation they belong to. For example if we have relations A, B and attributes A.x, B.x, these two attributes represent the same CSP variable.

Second, the domains D_i of the variables V_i are represented by the domains of the attributes of the database schema. In equi-joins the domain of an attribute is a subset of the domain of the CSP variable it corresponds to and the union of the domains of all attributes that correspond to the same CSP variable is equal to that variable's domain.

Finally, each constraint $C_j = \{R_{Sj}, S_j\}$ is represented by a database relation. The subset Sj of variables V is represented by the attributes of the database relation and the relation R_{Sj} is essentially the tuples in the relation that are produced by a subset of the Cartesian product of the domains of the relation's attributes.

The following table maps the CSP terminology to the database terminology we will be using from now on. Note that instead of "attribute" we will be mostly using "column names" or just "columns" which is usually used for matrixes in general.

CSP Terminology	**Our Database Terminology**
Constraint	Table, relation
CSP variable	Attribute, column name, column
Value of a CSP variable	Value of an attribute
Domain of a CSP variable	Domain of an attribute
Tuple in a constraint	Tuple in a table

Table 5 - Terminologies

After the connection between a CSP and a database has been made we can formulate a join query to solve the CSP. For example in our 4-queens problem the join in a database related language would look like this:

```
SELECT C1.Q1, C1.Q2, C2.Q3, C3.Q4
FROM C1, C2, C3, C4, C5, C6
```

WHERE C1.Q1 = C2.Q1 AND C2.Q1 = C3.Q1 AND C1.Q2 = C4.Q2 AND C4.Q2 = C5.Q2 AND C2.Q3 = C4.Q3 AND C4.Q3 = C6.Q3 AND C3.Q4 = C5.Q4 AND C5.Q4 = C6.Q4

Where {Q1, Q2}, {Q1, Q3}, {Q1, Q4}, {Q2, Q3}, {Q2, Q4}, {Q3, Q4} attributes of C1, C2, C3, C4, C5, C6 respectively. Q stands for Queen and C for Constraint.

4. JOIN ALGORITHMS IN MAPREDUCE

In this section we will briefly describe some of the already existing join algorithms in the MapReduce architecture. At the same time we will propose a new join algorithm that works specifically for CSPs.

4.1 Headers and Join key

One of the things we need to remember before moving to the presentation of the algorithms is the fact that when it comes to CSPs as joins every column in our schema is part of the join key. For this, some preprocessing regarding the header of each file needs to be done.

To be more specific, whenever we join two files we don't know in advance on which columns these two are going to be joined. What we do is get their headers and compare them with each other in order to find which columns exist in both files. The columns that do so represent (almost) the join key and from now on will be referenced as common columns.

In order to form the join key from the common columns further processing might be needed. The columns in each file aren't sorted by their names which causes extra problems. For example there are occasions where the header of one file is X1 X3 X2 and the header of the other X2 X3. Simply omitting the values from column X1 would leave us with the common columns but in reverse order!

In the driver code of our program we take care so that along with the position of every common column in a file we also pass to the job configuration the position this column needs to have in the join key. If the mapper is processing a record from the first file, the order of the values of the common columns is remained unchanged. If however it is processing the second file, the values of the common columns are reordered to match the order of the first file.

One more obvious thing that we should never forget when forming the join key is to always use a separator character between the common columns (usually a tab). In cases where the values are more than one character long the key 1 11 would be considered the same with the key 11 1 since the separator is missing and both keys become 111 instead.

Finally, we need to find the header file for the output of the job which is also done by comparing the two headers of our input files. This header is the first thing we write to the

output of the job. We keep the header of the first file unchanged and we concatenate it with the unique columns that appear only in the second file. Again, it is imperative that this order is kept within the records as well. Whenever we are about to join two records we must put the values originating from the unique columns of the second file at the end of the concatenation. At the start we put the join key and at the middle the values of the unique columns of the first file.

The preprocessing above takes place in the Repartition, Semi and Map join and hence won't be restated explicitly in those sections.

4.2 Repartition Join

Before we jump into the algorithm we need to explain some class extensions of the Hadoop framework.

First, there is the Header Input Format that extends the File Input Format. It's the same as the class it extends with the only difference that whenever a mapper gets the first split of a file it tosses the first line because it is the header – it contains the column names of the file.

Second, there is the Tagged Key class [8]. This class is used within the framework and hence needs to implement the writable and writable comparable interfaces. Apart from the join key that is used for the join this class also has a tag field which tells us from which file this key comes from. Tagged Key also has an overridden compare method that when join keys are equal it compares the tags.

Because the Tagged Key is a composite class the partitioner needs to know the field of the class that the partition will take place. For this we implement the Joining Partitioner that overrides the default one and partitions the Tagged Key by checking the join key field of the class [8].

Finally, we implement the Joining Grouping Comparator in a similar way [8]. The comparator needs to know which field to compare and our class does exactly that by again specifying the join key field.

With the classes above we are able to achieve what is known as secondary sorting. Secondary sorting sends each key to a reducer by partitioning the keys on the join key field of our Tagged key class then proceeds to again group the keys by checking the join key field of the class. However, when all the values of a key are

grouped and sorted in a reducer they will be differentiated based on their tags. Those with a lexicographically smaller tag will be placed on the top of the reducer's iterable list.

Repartition join is the most common way of performing joins in Hadoop Map Reduce. The framework partitions the two files into multiple smaller splits and each mapper takes one as input. In the setup we get can learn which file we are processing by getting the name of the file from the input split.

The map function extracts the join key from the common columns reordering them if necessary. We then combine the join key and the name of the file (used as a tag) into a Tagged Key object which is written as the output key. The rest of the columns represent the output value. An extra byte – the tag byte – is written in front of the value in order to know from which file it comes from. The Joining Partitioner proceeds to partition the key-value pairs based on the join key. The code of the map function is as follows

```
@Override
       protected void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
         String key_str = "";
         String value_str = "";
         int counter=0, i=0, col_counter=0;
         StringTokenizer itr = new StringTokenizer(value.toString());
         String[] key_array = new String[key_length];
         if (joinOrder == 0) // specified by the name of the file. 0 for
first file, 1 for second
         {
            for (Integer raw_number: key_raws)
            {
               if (raw_number >= 0)
                  key_array[raw_number] = itr.nextToken(); // order of key
columns changes to match the order of the other file
               else
                  itr.nextToken();
            }
            data.set(reverseOrder + value.toString().trim());
         }
         else
         {
            for (Integer raw_number: key_raws)
            {
               if (raw_number >= 0)
                  key_array[col_counter++] = itr.nextToken(); //order of key
columns remains the same
               else
                  value_str += itr.nextToken() + separator;
            }
            while (itr.hasMoreTokens())
               value_str += itr.nextToken() + separator;
            data.set(reverseOrder + value_str.trim());
         }
         Out_Key.set(Arrays.toString(key_array).replaceAll(",|\\[|\\]", ""),
```

```
reverseOrder);

            context.write(Out_Key, data);
      }
}
```

Table 6 – Repartition join map method

In the reducer all the values of the same join key are grouped together by the Joining Grouping Comparator. By using Tagged Key class that we mentioned above we achieve a secondary sort in the reducer. We check the values of the iterable one by one and add them into a list. While the tag byte in front of every value remains the same it means that we read values from the first file since the values are sorted by their tag. When that byte changes it means that all the values following the change belong to the second file. We join every remaining value of the iterable - values of the second file – with the values we have previously added on the list – values of first file – and write them on the output of the reducer. Reduce function code is shown below.

Table 7 - Repartition Jon Reduce method

```
    public void reduce(TaggedKey key, Iterable<Text> values, Context context)
throws IOException, InterruptedException
   {
      aList.clear();
      char flag = 0;
      value = values.iterator().next().toString();
      file_descriptor = Integer.parseInt(value.substring(0,1));

      if (file_descriptor == 1) //no entries from first file. descending
order
         return;
      aList.add(new Text(value.substring(1)));

      for (Text val : values) {

         value = val.toString();

         if (Integer.parseInt(value.substring(0,1)) != file_descriptor)
         {  flag = 1; //tag byte has changed. records from file 2 from now on
            break;
         }
         aList.add(new Text(value.substring(1)));
      }
      if (flag == 1)
      {  right_val.set(value.substring(1));
         for (Text list_val : aList)
            context.write(right_val, list_val);
      }
      for (Text val : values) {
         right_val.set(val.toString().substring(1));
         for (Text list_val : aList)
         {
            context.write(right_val, list_val);
         }
      }
```

```
    }
}
```

Since the algorithm is cascaded we repeat the process until all files have been joined. The input of the next job is the output of the previous one plus a file of our choosing.

4.3 Semi Join

Semi Join consists of three phases, one map reduce job followed by two map only jobs [7].

In the first phase we select the smaller out of the two files and use it as input for our job. The mapper takes the records of the file one by one and writes to its output the values of the key columns as the key and null as the value. The reducer receives those pairs and tosses the null values writing to its output only the key. In this way we now have the unique values of the key columns that appear in this file.

In the second job we cache in the distributed cache the output of the previous job and use as input the second file. In the mapper we load the cached file in a map structure and for every record we encounter we check whether the values of its key columns exists in the structure. If they do we write this record in the output. Since it is a map only job, no reducer is used. We have now tossed some of the records of the second file that for sure can't be joined with the first file because the values of their key columns don't appear in the first file.

The third job is also a map only job that is known as broadcast join. Broadcast join selects the smallest out of the two files and puts it into the distributed cache. In our case the two files are the output of the second phase and the first file (that was used as input in the first phase). Normally, the algorithm stores the smaller of the two in the distributed cache buts since we have already compared the two files in the first phase we already know we need to choose the first file. There is an extreme case where the second file becomes smaller than the first after filtering it during the second phase but we did not take it into consideration. In the mapper, during the setup method we load the cached file on a map structure. The key of the structure are the common columns while the value is a collection of the records that are associated with that key. In the map function we again formulate the key from the common columns and probe the map structure with it. We join every record that the structure returns with the current line of the input file.

Because the algorithm is cascaded we redo the whole process by joining the output of the previous job with the next file of our choosing until all files are joined. The high level code of the three semi join phases is presented below.

```
for (int i=0; i < num_of_joins; i++)
{  String temp_name;
   boolean swapped = false;

   // CACHE THE SMALLER FILE
   // swap files
   if (InputOptimization.compare_files(filename2, out_namefile, fs) == true)
   {  temp_name = filename2;
      filename2 = out_namefile;
      out_namefile = temp_name;
      swapped = true;
   }

   String header1 = StringManipulation.get_header(out_namefile, fs);
   String header2 = StringManipulation.get_header(filename2, fs);
   //GET THE COMMON COLUMNS OF THE TWO FILES. THEY ARE THE KEY FOR THE JOIN
   String[] keys = StringManipulation.Intersection(header2, header1,
separator);
   String new_header = StringManipulation.new_header(header2, keys[3],
separator); //the header of the output file

   //1st phase
   //MR-JOB TO GET THE VALUES OF THE KEY COLUMNS OF FILENAME2
   hash_semi_join.hash_semi_join(filename2, "./" + i + "-mid-output",
keys[0], num_of_reducers, separator);
   //2nd phase
   //MR-JOB TO KEEP THE RECORDS OF OUT_FILENAME THAT ONLY HAVE THE VALUES
ABOVE
   semi_join_phase2.semi_join_phase2(out_namefile, "./" + i + "-mid-output",
"./" + i + "-mid-output2", header1, keys[1], ",", num_of_reducers,
separator);

   fs.delete(new Path("./" + i + "-mid-output"), true);
   if (i != 0 && swapped == false) // if swapped path is still needed. delete
it later
      fs.delete(new Path(input_path + current_num_of_files), true);

   //OUTPUT OF 3RD PHASE
   current_num_of_files++;
   out_namefile = input_path + current_num_of_files + "/";

   //3rd phase
   //MAP ONLY JOB. JOIN THE OUTPUT OF 2ND PHASE WITH FILENAME2
   broadcast_join.broadcast_join(filename2, "./" + i + "-mid-output2",
out_namefile, new_header, keys[0], "m", ",", separator);


   //FIND WHICH FILE IS BEST TO JOIN WITH THE OUTPUT OF THE LAST JOB
   filename2 = InputOptimization.Inner(filenames , out_namefile , fs,
separator);

   fs.delete(new Path("./" + i + "-mid-output2"), true);
   if (i != 0 && swapped == true)
      fs.delete(new Path(input_path + (current_num_of_files-1)), true);
}
```

Table 8 - Semi Join

4.4 Constructive Join

Constructive join is the new algorithm proposed by this paper. Its name comes from the fact that it constructs the outcome of the join column by column – one column per job. This also explains why it only works for CSPs: It is imperative that every column in our schema is part of the join key.

4.4.1 Preprocessing

Since the algorithm uses every column in our schema the first thing we need to do is gather all the column names in a collection. To achieve this we scan through the headers of every input file tossing the duplicate names wherever they exist. It is this data structure that we iterate through during the execution of the algorithm and get one column name per job. Note that the collection must be sorted lexicographically. We will explain later why.

Next, we need another structure where we map every file name to a counter. Each counter represents how many columns we have already iterated through in this file. If for example we have two files, one with header “X1 X2” and another with “X1 X3” and we have iterated through the column names X1 and X2 the counter for the first file is currently 2, while the counter for the second file is 1 (because X3 hasn’t been iterated yet!). These counters serve as indexes for the mapper – to know where each record of a specific file needs to be split. For all this to work it is necessary that the headers of all files are ordered lexicographically.

Finally, we need a third structure to map every column name to all the files it exists in. This structure helps us find the input paths of every job, since the input of a job is exactly what this structure returns when we probe it with the current column name.

4.4.2 The Algorithm

We start by getting the first column name from our ordered collection as we described above. Then we probe the column name to our second map structure and get the input files for the job. Files that don’t contain this column name don’t add any constraints to this column and hence aren’t needed in the current job. After that, we probe all the input file names to our first map structure and increase the corresponding counters by one. We pass the affected counters to the MR job configuration. The key to get them in the mapper is again their corresponding filename. Now, each file split in the mapper can use this counter as an index to find the value of the current column name

for all of its records. Remember that each file split has access to the name of the original file which enables it to collect the proper counter.

4.4.3 Distributed Cache

Whenever a job terminates output is stored into the hdfs. The output has as many columns as the column names we have already iterated. For example, assuming a schema has three unique column names (X1, X2 and X3) after the first job the output will have one column – X1. After the second iteration the records of the output will have two columns – X1 X2. Finally after the last iteration the output will have three – X1 X2 X3 – and at this point we have found the solutions of our CSP problem. For this, those intermediate files are named solution files and they are cached in the distributed cache of the hdfs every time a mr job takes place.

Apart from the position of the current column in an input file we also need to know the columns of the solution file that also exist in an input file – the common columns. To achieve this, we just cycle through the header of the solution file and the header of the input file. The comparison is done linearly because both headers are sorted lexicographically. We pass the result to the job configuration by using again the input filename as a key – albeit a bit statically changed to differentiate the two values.

Last but not least we overwrite the HeaderInputFormat so that input files are not splitable. We will see later why.

4.4.4 The Mapper

We start with the setup method where we get the name of the input file from the input split. We use this to get the index of the current column name in this input file. We also read our cached files and populate a map structure. We again use the filename to get the common columns. For each record in the cached file we use the values of the common columns as a key to our structure and the whole record as a value.

In the map function of the mapper we split each line on the current column by using its index. Every column before the current column consists the common columns. Every column after the current column is not needed and left to be analyzed by the next jobs. We use the columns prior the current one as a key to the map structure that we populated in the setup method. We concatenate every record of the solution file that the structure returns with the value of the current column and write the result of the concatenation –only if we haven’t already- to the mapper output as the key. We write null to the output value. The code of the map method can be viewed below.

```
@Override
protected void map(Object keyin, Text valuein, Context context) throws
IOException, InterruptedException
{
   StringTokenizer itr = new StringTokenizer(valuein.toString(), separator);
   String build_str = ""; // the common columns
   int i;
   for (i=0; i<index_of_var-1; i++)
   {
      build_str += itr.nextToken() + separator;
   }
   if (i == index_of_var - 1)
      build_str += itr.nextToken();

   String last_token = itr.nextToken(); //current column

   if ((a_sol = current_sls.get(build_str)) != null) //true when build_str ==
""
   {  if (a_sol.size() != 0)
      {
         if (already_constructed.contains(a_sol.get(0) + separator +
last_token)) //remove duplicates
            return;
         else
            already_constructed.add(a_sol.get(0) + separator + last_token);
         for (i=0; i< a_sol.size(); i++) {
            key_out.set(a_sol.get(i) + separator + last_token);
            context.write(key_out, null_value);
         }
      }
      else // enter here when current key column is the first column on this
file, i.e no common columns
      {  if (!already_constructed.contains(last_token))
         {  already_constructed.add(last_token);
            key_out.set(last_token);
            context.write(key_out, null_value);
         }
      }
   }
}
```

Table 9 - Constructive Join map method

4.4.5 The Reducer

We leave the Hadoop framework to do its shuffling, grouping and sorting stages. If the size of the iterable in a reducer is equal to the number of the input files then the key of the reducer is an -up until now- valid solution and is written to the reducer output – the solution file for the next job. We repeat the same procedure for the next column name until all of the unique column names have been iterated. The code of the reduce functions can be viewed below.

```
@Override
protected void reduce(Text keyin, Iterable<NullWritable> values, Context
```

```
context) throws IOException, InterruptedException
{  int count = Iterables.size(values);
   if (count == num_of_files)
   {
      context.write(keyin, null_value);
   }
}
```

Table 10 - constructive Join reduce method

The reason why this works is because each mapper gets a whole input file (files are not splitable). Then in the map function if an output key is already written to the output it won't be duplicated. This means that each file produces unique keys which are transferred to the reducers. Assuming we have N input files which equals to N mappers, if a reducers receives N times a certain key, it means all N files produced this key and that this key/solution complies with the constraints of all the input files.

By this time, it is easier to explain why both our collection of column names and the headers of the input files are sorted lexicographically. It is done this way to have less complicated structures in our driver code. In this way we only need to keep one counter per file in order to find the position of the current column in a record. Otherwise we would need to keep the location of every column in every file. Most importantly it allows us to find the common columns of the input file in linear time since all of them are before the current column.

4.4.6 An Alternative Approach

Whenever the input files are larger than the solution file our algorithm choses to cache the input files instead and use the solution file as the input to the job. In this case, the two files are processed in the same way but since the solution file is cached, it is the one that will be loaded to the map structure in the setup method.

The main difference this time is that we have as many map structures as there are input files – each one for each input file. In order of a line from the solution file to be considered valid the values of its common columns must exist in every one of those structures. Note that the common columns differ for every map structure.

4.5 Map Side Join

In contrast to reduce side joins, map side join is an algorithm that evaluates the join during the map phase of the framework [6]. However, there are a few extra constraints

that the input files must follow in order to execute it. The following table summarizes these constraints.

Limitation	Why
All datasets must be sorted using the same comparator.	The sort ordering of the data in each dataset must be identical for datasets to be joined.
All datasets must be partitioned using the same partitioner.	A given key has to be in the same partition in each dataset so that all partitions that can hold a key are joined together.
The number of partitions in the datasets must be identical.	A given key has to be in the same partition in each dataset so that all partitions that can hold a key are joined together.

Table 11 - Map Join Limitations

Luckily, all these constraints are satisfied by a Hadoop MapReduce job. So, in order to execute a map join we first run an extra MR job to sort the datasets by the same partitioner and comparator.

4.5.1 The Algorithm

Our version of the map side join is cascaded and us such it takes two constraint files as input each time, until all files have been processed. One of the two files is always the output of the previous map join job.

Normally, one can use one sorting job for each one of the input files. Instead we decided to sort the two files in a single job. For this, all the classes that were used in previous algorithms for tagging each file are also used here. The way the join key is extracted from the common columns is also the same. The difference is that we don't want our sorting reducer to mix the two files.

```
    @Override
   protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
```

```
      List<String> values =
Lists.newArrayList(splitter.split(value.toString()));
      String joinKey ;
      builder.setLength(0);
      for(String keyIndex : indexList){
         builder.append(values.get(Integer.parseInt(keyIndex))+"\t");
         values.set(Integer.parseInt(keyIndex),"");
      }
      builder.setLength(builder.length() - 1);
      joinKey = builder.toString();
      values.removeAll(emptyList);
      String valuesWithOutKey = joiner.join(values);

      taggedKey.set(joinKey, joinOrder);
      fatValue.set(valuesWithOutKey,joinOrder);
      context.write(taggedKey, fatValue);
   }
}
```

Table 12 - Map Join sorting map method

The Multiple Output Format Class is a built in Hadoop class that helps us differentiate the reducer output based on the file a record belongs to. That is records with the same join key will still end up in the same reducer but each reducer has two different output paths – one for each file – so that the records are kept separated.

```
@Override
protected void reduce(TaggedKey key, Iterable<FatValue> values, Context
context) throws IOException, InterruptedException {
   Iterator<FatValue> iter = values.iterator() ;
   while(iter.hasNext()){
      fatvalue = iter.next();
      combinedText.set(fatvalue.toString());
      if( fatvalue.getJoinOrder() == 1  ){
         keyOut.set(key.getJoinKey());
         context.write(keyOut,combinedText);
      }else{
         keyOut.set(key.getJoinKey());
         mos.write("file2",keyOut,combinedText);
      }

   }
}
```

Table 13 - Map Join reduce sorting method

Notice that the two first criteria for the Map join are met by the fact that both files are sorted in the same job and hence use the same partitioner and comparator. They are also sorted by the same key via the reordering of their common columns. The third criterion is also met by the fact that the sorting job uses a set number of reducers. Remember that the number of output partitions is the number of the reducers used and

since each reducer writes two separate files if we have N reducers we will get N partitions for the one files and N for the other.

Now that the two files are sorted properly and obey the map join constraints we can move on to the second part of the algorithm which is the actual join [14]. Fortunately, Hadoop provides us with a built in implementation for performing map side joins. All we need to do is specify the type of join we want –in our case the inner join- and then let the Composite Input Format class do the work for us. This class gets as input the type of join that will be executed and then proceeds to join the records of the two files before they reach the mapper. For this, the mapper that we specify is a simple one. We just make sure the values of the already joined records are properly formatted and that there is a match between the order of the columns in the header and the order of their values in the records just as with our other algorithms.

```
@Override
protected void map(Text key, TupleWritable value, Context context) throws
IOException, InterruptedException {
   valueBuilder.append(key).append(separator);


   for (Writable writable : value) {
      String test1 = writable.toString();
      if(!test1.equals(""))
         valueBuilder.append(writable.toString()).append(separator);
   }
   valueBuilder.setLength(valueBuilder.length() - 1);
   outValue.set(valueBuilder.toString());
   context.write(nullKey, outValue);
   valueBuilder.setLength(0);
}
```

Table 14 - Map join combined values map method

We repeat the process (sorting and joining) until all files have been joined.

4.6 Optimizations

We can improve all of the above cascaded two-way algorithms by joining first the pair of files with the least output cardinality [1], which is the number of output records of a join. In order to find the output cardinality of a join we need to sum the products of tuples from file A with tuples from file B that have the same join key. The problem with CSPs is that every pair of files has different columns as join keys, which means that each time we want to count a record we need first to rearrange these columns to form the key and get the proper value. Something like that might not be cost-efficient

especially when it comes to the first join pair that is chosen among all pair possibilities (O(n^2) where n is the number of input files).

Instead we decided to do something simpler that is not optimal but makes an educated guess of the output cardinality with less effort. Assume we have two files represented by the two tables below.

A	B	C
1	2	3
1	4	6
1	7	8
2	3	6
2	4	8

A	D	E
1	2	6
1	5	9
2	3	7
2	1	8

Table 15 - Input files

In this case the join key is A and the output cardinality of the join is:

3*2 + 2*2 = 10 records

However if the header of the second file was A B E (instead of A D E) then the join key would be A B and the cardinality would be:

1*1 + 1*0 + 1*0 + 0*1 + 1*1 + 1*0 + 0*1 = 2 records

The idea is that the more common columns there are between two files the less likely it is to have larger output cardinality. Note that if two files have 0 common columns between them, then we are guaranteed to have a*b output records where a, b are the number of records of the two files, which is the largest output cardinality we can have. So instead of processing all the records of the files we just check their headers and pick the pair that has the most common columns. If some pairs have the same number of common columns we just pick the smaller files among them. Implementing this

optimization, instead of choosing random files, has helped us not run out of memory for size of problems that previously resulted so.

5. EXPERIMENTS

Experiments we conducted on a single virtual machine cluster. Its specifications are as follows:

No. of virtual machines	8
CPU	2 GH
Memory	8 GB
Disk Space	340 GB
OS	Ubuntu 14.04.3 LTS
Linux Kernel	3.13.0.62-generic

Table 16 - Cluster Specifications

The virtual machine was running hadoop's 2.7.2 version. However, programs were compiled using the 2.7.1 version

5.1 N Queens

The table below shows the execution time of our algorithms. Time is calculated in minutes.

	N_Queens_12	N_Queens_13	N_Queens_14	N_Queens_15	N_Queens_16
Repartition Join	44.5	56	102	(2)	(2)
Constructive Join	9.4	17	72.8	(3)	(3)
Semi Join	104	(2)	(2)	(2)	(2)

Map Join	75.3	93.4	158.8	(2)	(2)
Using ECLiPSe CPS	0.07	0.37	2.2		

Table 17 - N queens results

(0): Error: DiskErrorException: Could not find any valid local directory for map output

(1): Error: GC overhead limit exceeded

(2): Error: Java heap space

(3): Unknown Error

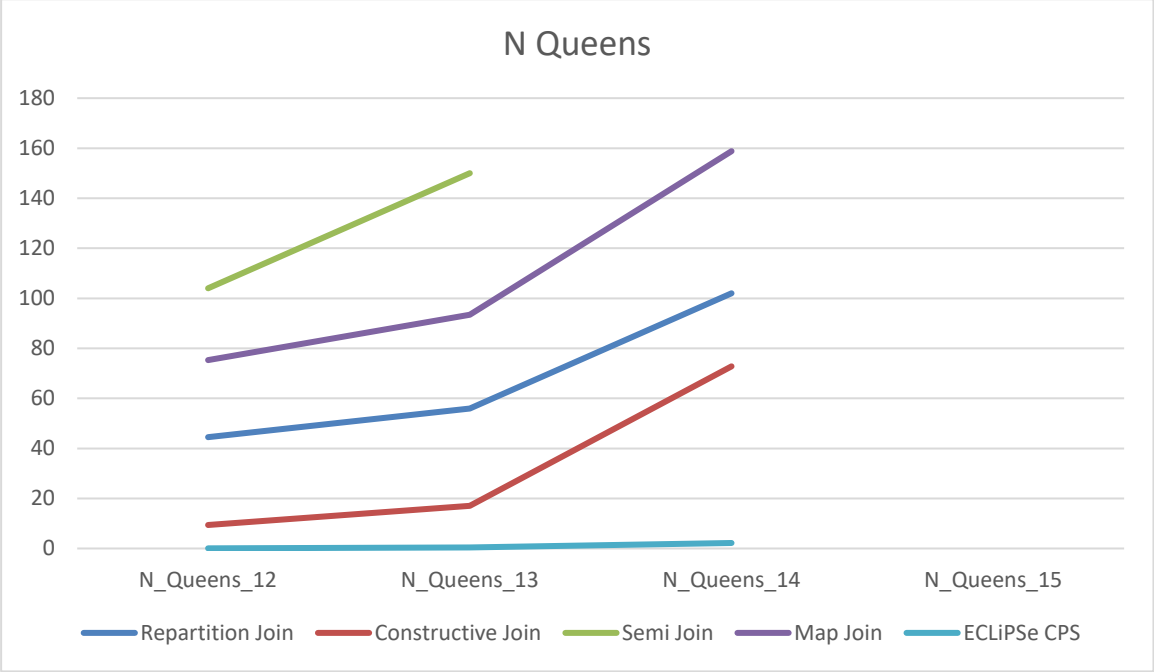

Figure 3 - N queens results

5.2 Spatially Balance Latin Squares

	SBLS_8	**SBLS_9**
Repartition Join	28	(0)
Constructive Join	52	(1),(2)

Semi Join	45.2	(2)
Map Join	38.8	(0)

Table 18 - SBLS results

(0): Error: DiskErrorException: Could not find any valid local directory for map output

(1): Error: GC overhead limit exceeded

(2): Error: Java heap space

(3): Unknown Error

6. CONCLUSION

Unfortunately, as seen by the experiment section we were not able to provide sufficient results to support the case of using the Hadoop MapReduce framework as a way of efficiently solving Constraint Satisfaction Problems. The main problem we faced is the lack of memory in the cluster we used which resulted in our programs terminating abruptly with the java heap space exception. We would suggest reperforming the experiments in a more optimal environment – a more scalable cluster.

ABBREVIATIONS - ACRONYMS

CSP	Constraint Satisfaction Problem
MR	MapReduce
HDFS	Hadoop Distributed File System

REFERENCES

[1] Chandar, J. Join Algorithms using Map/Reduce, Magisterarb. University of Edinburgh, 2010
[2] Jefrey, D., and S. Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM (2008).
[3] IBM - https://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/ 6-10-2016
[4] Apache Hadoop - http://hadoop.apache.org.
[5] Wikipedia - https://en.wikipedia.org/wiki/Apache_Hadoop 6-10-206
[6] J. Venner. Pro Hadoop. Apress, 1 edition, June 2009.
[7] Blanas, S. A comparison of join algorithms for log processing in MapReduce. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM.
[8] http://codingjunkie.net/mapreduce-reduce-joins/ 2-10-2016
[9] Rossi, Francesca, Peter Van Beek, and Toby Walsh, eds. Handbook of constraint programming. Elsevier, 2006.
[10] Tsang, Edward. "A glimpse of constraint satisfaction." Artificial Intelligence Review 13.3 (1999): 215-227.
[11] E. J. Hoffman et al., "Construction for the Solutions of the m Queens Problem". Mathematics Magazine, Vol. XX (1969), pp. 66–72
[12] Haralick, R.M. & Elliott, G.L., Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence, Vol.14, 1980, 263-313
[13] Lal, Anagh, and Berthe Y. Choueiry. "Constraint Processing Techniques for Improving Join Computation: A Proof of Concept." International Symposium on Constraint Databases and Applications. Springer Berlin Heidelberg, 2004.
[14] http://codingjunkie.net/mapside-joins/ 6-10-2016
[15] MapReduce tutorial - https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html 7-10-2016
[16] T. White. Hadoop: The Definitive Guide. O'Reilly Media, 1 edition, October 2010.