SmartJoin: a network-aware multiway join for MapReduce

**Kenn Slagter · Ching-Hsien Hsu · Yeh-Ching Chung ·
Gangman Yi**

Received: 30 July 2013 / Revised: 9 January 2014 / Accepted: 20 January 2014 / Published online: 19 February 2014

Abstract MapReduce is an effective tool for processing large amounts of data in parallel using a cluster of processors or computers. One common data processing task is the join operation, which combines two or more datasets based on values common to each. In this paper, we present a network aware multi-way join for MapReduce (SmartJoin) that improves performance and considers network traffic when redistributing workload amongst reducers. SmartJoin achieves this by dynamically redistributing tuples directly between reducers with an intelligent network aware algorithm. We show that our presented technique has significant potential to minimize the time required to join multiple datasets. In our evaluation, we show that SmartJoin has up to 39 % improvement compared to the non-redistribution method, a 26.8 % improvement over random redistribution and 27.6 % improvement over worst join redistribution.

Keywords MapReduce · Hadoop · Multiway join · Workload redistribution

1 Introduction

MapReduce [1] is a flexible programming model proposed by Google for processing and creating data sets over a cluster of computers. The MapReduce model hides extraneous details inherent in distributed programming such as parallelization, fault tolerance, data distribution and load balancing within a library. This simplifies the process of writing distributed programs, which is an advantage MapReduce has over other distributed programming models such as MPI that requires the programmer to explicitly handle the data flow [2].

Programmers who use the MapReduce library need to write two functions, a map function and a reduce function. The purpose of the map function is to take the input key/value pairs from an input source, process it, and then produce a set of intermediate key/value pairs. The intermediate key/value pairs it generates is then sent to a reduce function as input. The reduce function processes this input and then generates its own set of key/value pairs as output.

Parallelism is achieved by running multiple map and reduce functions on multiple processors or machines. The intermediate key/value pairs produced by each of the map functions are partitioned so that intermediate key/value pairs that share the same key are all sent to the same reduce function. The reduce function then processes each key alongside a list of associated values.

Since its conception by Google, MapReduce has had many adopters in industry and academia. One of the more well-known adopters is Yahoo, who developed an open source implementation known as Hadoop [3]. Hadoop is a Java-based implementation, which by default uses its own distributed file system (HDFS). Because Hadoop is open source, well documented and easy to use, the tool has gained prominence in the distributed programming community. For this

K. Slagter · Y.-C. Chung
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, ROC
e-mail: KennSlagter@hotmail.com; kennslagter@sslab.cs.nthu.edu.tw

Y.-C. Chung
e-mail: ychung@cs.nthu.edu.tw

C.-H. Hsu (✉)
Department of Computer Science, Chung Hua University, Hsinchu, Taiwan, ROC
e-mail: chh@chu.edu.tw

G. Yi
Department of Computer Science, Gangneung-Wonju National University, Gangwon, Gangneung, South Korea
e-mail: gangman@cs.gwnu.ac.kr

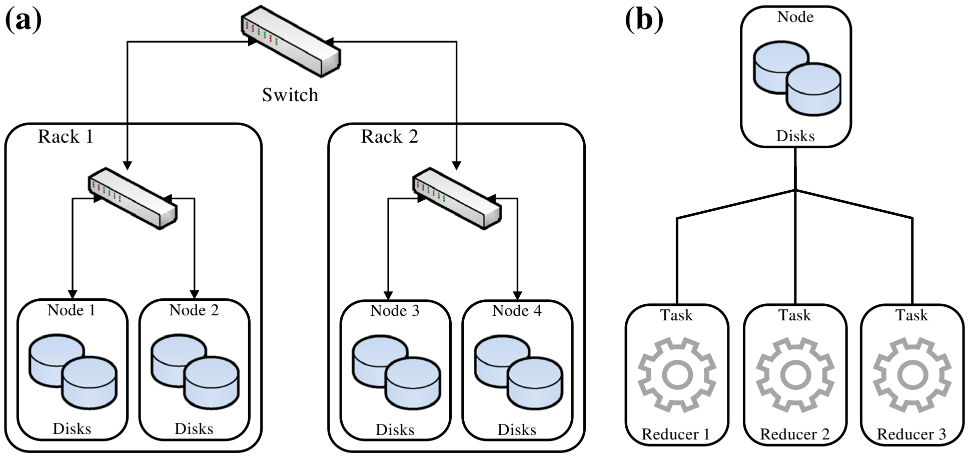


Fig. 1 Network model used in this paper **a** a tree network consisting of racks and nodes. **b** A node running a set of reduce tasks

reason, we use Hadoop as our reference platform for MapReduce in this paper.

The MapReduce model is effective at processing large amounts of data or datasets. A dataset is essentially a set of tuples stored in a file. Most computing devices can generate datasets, and the data can be about almost anything. Examples of data sources include log files [4], sensors [5] and social media [6]. In this paper, we look at one of the more common data processing operations called a join. A join combines two or more datasets together based on some common value. There are many possible ways to implement a join. The efficiency of a join implementation depends on how many data sets there are and how large the data sets are. A MapReduce join can be implemented as a map-side join or a reduce-side join and multiple datasets may be handled using either a cascade join or a multiway join [7].

Cascade Joins handle multiple joins using successive two-way joins. In other words, a cascade join is performed by joining datasets two at a time. Essentially, a cascade join is an iterative version of a two-way join. The main advantages of a cascade join is that it can handle any number of datasets of any size so long as they can all be stored on the HDFS. With n tables, T_1,T_2,$T_{3,...,}$$T_n$, table T_1 and table T_2 are joined in one job. The table created by this join is then joined with T_3. This continues until all the tables are joined. Using relational algebra this can be expressed as:

$$(\ldots(((T_1 \bowtie T_2) \bowtie T_3)\ldots \bowtie T_{n-1}) \bowtie T_n \qquad (1)$$

Multiway joins handle multiple joins simultaneously. Using relational algebra a multiway join can be expressed as:

$$T_1 \bowtie T_2 \bowtie T_3 \bowtie \ldots \bowtie T_{n-1} \bowtie T_n \qquad (2)$$

Multiway joins have certain advantages and disadvantages over cascade joins. First, it avoids considerable overhead since it does not need to setup multiple jobs. Second, it can save space on the network since it does not need to store intermediate results. However, there are some drawbacks to multiway joins. When a multiway join is performed, it needs to buffer tuples. This can lead to memory problems, especially if the data is skewed. Therefore, the number of datasets and the size of datasets are limited by the memory resources available.

The main idea of SmartJoin is to improve processing time of a multiway join by dynamically redistributing the workload between reducers. Furthermore, it does this by considering network topology during workload redistribution.

The main contributions of our work are as follows. First, we present a model to redistribute tuples amongst reducers on the MapReduce framework for a multiway join. Second, we show how the SmartJoin redistribution algorithm can reduce job response times for a multiway join by considering network distance and reducer workload. Third, we compare our method to alternative methods that do not take into account these factors.

The rest of this paper is organized as follows. Section 2 explains our network model. Section 3 presents the proposed techniques on multiway joins and tuple redistribution. In Sect. 4, the simulation results and performance analysis are given to weigh the pros and cons of the proposed method. In Sect. 5, we discuss related work. Finally, the conclusion and future work are presented in Sect. 6.

2 Background

2.1 Network model

The research model for this study is presented in Fig. 1, which shows a network environment consisting of switches, racks and nodes.

The two-level tree topology shown in Fig. 1a is a common network layout used by Hadoop. Each rack contains a set of servers (nodes) all interlinked by a switch. The racks themselves then uplink to a core switch or router. The nodes

Table 1 Network distance

Notation	Distance	Node	Rack
Distance ($N_{(1,1)}$,$N_{(1,1)}$)	0	Same node	Same rack
Distance ($N_{(1,1)}$,$N_{(2,1)}$)	2	Different node	Same rack
Distance ($N_{(1,1)}$,$N_{(3,2)}$)	4	Different node	Different rack

run map or reduce tasks as shown in Fig. 1b. In this paper, map tasks and reduce tasks are also referred to as mappers and reducers respectively.

The purpose of the MapReduce model is to process large amounts of data over a cluster of computers. This requires sending data between nodes. The rate one can send data over the network is restricted by the bandwidth.

In Hadoop, the distance between two nodes is calculated as the sum of the distances to the lowest common ancestor in the network tree. Although the structure of the tree is not predetermined, it is common to assign the levels of the tree based on the data center, rack and node on which a process resides. The purpose of this model is to reflect the way bandwidth decreases between processes running on the same node, nodes on the same rack, nodes on different racks and potentially nodes in different clusters.

Three different network scenarios may occur in a data center. These scenarios can be expressed using the notation $N_{(i,j)}$ which represents node$i$on rack j. Table 1 describes the scenarios that may arise on a network with a two-level tree topology, as shown in Fig. 1, and the subsequent network distances.

Based on the network distance one can determine the relative distance between two processes residing on a network. If a process resides on the same node and same rack as another process its network distance is 0, since both processes reside on the same node. If a process resides on a different node and on the same rack, its distance is 2. If a process resides on a different node and on a different rack its distance is 4. Essentially, the distance metric counts the number of hops that data needs to travel to get from one node to another.

When data is sent by one process to another on the network, the distance the data needs to travel increases. Essentially, network distance is based on the number of switches or devices that data has to traverse to reach its destination (not including the point of origin). If both processes reside on the same node, communication and resources can be shared locally. If both processes reside on the same rack, data has to traverse the switch on that rack, before it can arrive at its destination node. If both processes reside on different racks, data has to traverse multiple switches. Thus, if a network has a two-level tree topology, and if data is being sent between nodes on different racks, there would be three switches for the data to traverse.

The speed of switches on a network can differ in practice. The costs of a fiber optic network used by high speed networks is often prohibitive for many consumers, and often a traditional switch is all that is required on any given rack. If the speed of a switch on one rack is slower than another rack, the effective network bandwidth between those racks is reduced to that of the slowest switch.

2.2 Join algorithms

Join algorithms have been studied extensively over the years, with many different variants existing for each type of algorithm. Many join algorithms in academia predate the invention of MapReduce, due to their ubiquitous use throughout the database community. The multiway join algorithm presented in this paper is a hybrid join and is a combination of a reduce-side join and a hash-join.

2.2.1 Reduce-side join

Reduce-side joins are based on the MapReduce programming model, which is composed of a map phase and a reduce phase. In the map phase, the datasets are read by a map function, which is executed by a map task, one tuple at a time. The purpose of the map function is to pre-process the tuples and sort them by the join key. Before tuples are partitioned based on their join key, they are tagged so that the reduce function can know which table the tuple originated from. In Hadoop, a tag can be performed with a custom TextPair class. As its name suggests, a TextPair class contains two Text values. The purpose of the TextPair class is to bind a tag to the key and value so that the reducer can discern which table they came from.

After the key and value are tagged, the mapper partitions the key-value pairs. In Hadoop, a partitioner class determines how the partitions are divided, and the number of reducers determines the number of partitions. The partitioner class partitions the data based on the join key so that all the tuples that share the same key are sent to the same reducer. A custom partitioner class is then used to override the default partitioner so that partitioning is only performed with the key and not the tag.

Although this ensures all tuples go to the correct reducer, the reducer groups all values based on the key and processes them together in the reduce function. Since the key is a TextPair class containing two values the default grouping function handles this incorrectly as a single entity. Therefore a custom comparator class is also required so that

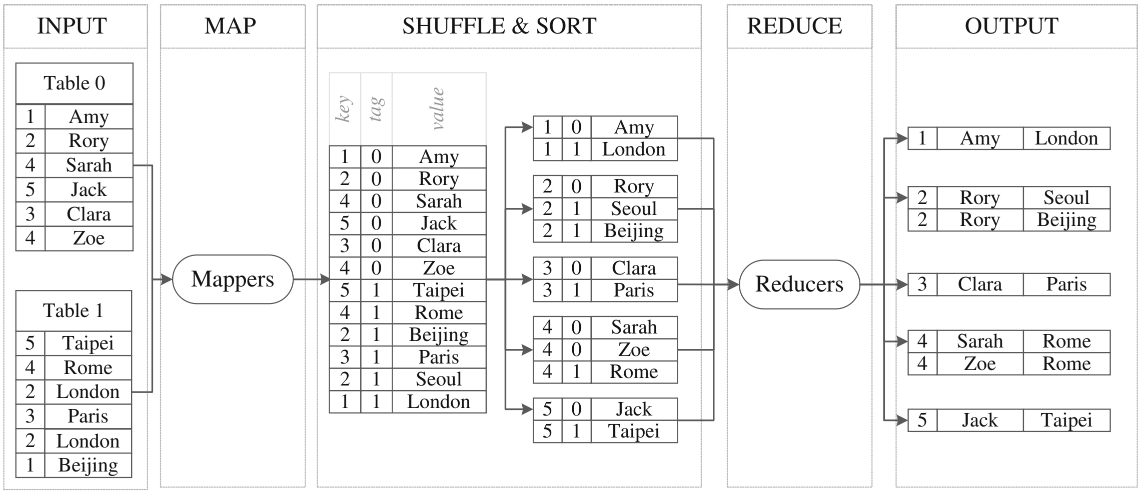


Table 0	
1	Amy
2	Rory
4	Sarah
5	Jack
3	Clara
4	Zoe

Table 1	
5	Taipei
4	Rome
2	London
3	Paris
2	London
1	Beijing

key	tag	value
1	0	Amy
2	0	Rory
4	0	Sarah
5	0	Jack
3	0	Clara
4	0	Zoe
5	1	Taipei
4	1	Rome
2	1	Beijing
3	1	Paris
2	1	Seoul
1	1	London

1	0	Amy
1	1	London

2	0	Rory
2	1	Seoul
2	1	Beijing

3	0	Clara
3	1	Paris

4	0	Sarah
4	0	Zoe
4	1	Rome

5	0	Jack
5	1	Taipei

1	Amy	London

2	Rory	Seoul
2	Rory	Beijing

3	Clara	Paris

4	Sarah	Rome
4	Zoe	Rome

5	Jack	Taipei

Fig. 2 Reduce-side join (2-way)

only the key is considered when the reducer processes these groups. These groups are sorted by the composite key so that tuples are secondary sorted by their tag. In the 2-way join shown in Fig. 2, it means tuples from the first table would arrive before tuples from the second table,

The reducer then calls its reduce function for each group of keys. In Hadoop, tuples are read from the HDFS stream. There is no random data access, so the reduce function buffers the tuples from the first dataset. These are then joined to tuples from the second dataset, which is read directly from the HDFS stream. Only a single key is presented for each group, therefore the tag on the values is used to identify which dataset a tuple came from.

2.2.2 Hash join

A hash join is a traditional algorithm used by databases for joining two datasets together. A hash join consists of two distinct phases a 'build' phase and a 'probe' phase. In the build phase, the smallest dataset is inserted into an in-memory hash table. In the probe phase, the largest dataset is scanned and joined with the appropriate tuples stored in the hash table.

Consider two datasets P and Q. The algorithm for a simple hash join is as follows

```
for all p ∈ P do
Load p into in memory hash table H
end for
for all q ∈ Q do
 if H contains p matching with q then
        add (p,q) to the result
 end if
end for
```

2.2.3 Map-side join

MapReduce joins can be performed on either the map-side or the reduce-side. Map-side joins can occur between large inputs by joining the data prior to the execution of the map function. However, the inputs to each mapper have to be partitioned and sorted in a particular way. Firstly, all inputs must be sorted by the same join-key. Secondly, all records of a specific key must be stored on the same partition. Finally, each input must be divided into the same number of partitions [3]. These requirements can be met if the inputs have been preprocessed and outputted via a MapReduce job.

Consequently, for the Map-side join to work, the data has to have been pre-processed by another job. This is in contrast to the Reduce-side Join, which does not require the input dataset to be structured in any particular way. In this paper, the two largest tables are joined first via a reduce-side join. A reduce-side join is used in order to avoid the various limitations imposed by a map-side join and because a map-side join would require an additional MapReduce job.

Finally, the SmartJoin presented in this paper, joins smaller tables on the reducer-side in order to exploit the fact that the partitioned tables are sorted by their join attribute prior to the reduce phase [8]. Since smaller tables tend to be joined first, this join tends to be more effective when the initial join is highly selective.

3 SmartJoin

In this section, we present our proposed multiway join. The purpose of a multiway join is to join multiple datasets together. Our proposed join improves performance of the multiway join by redistributing the workload amongst reducers. Unlike other schemes that redistribute the workload using a distributed queue [8], our methodology redistributes the

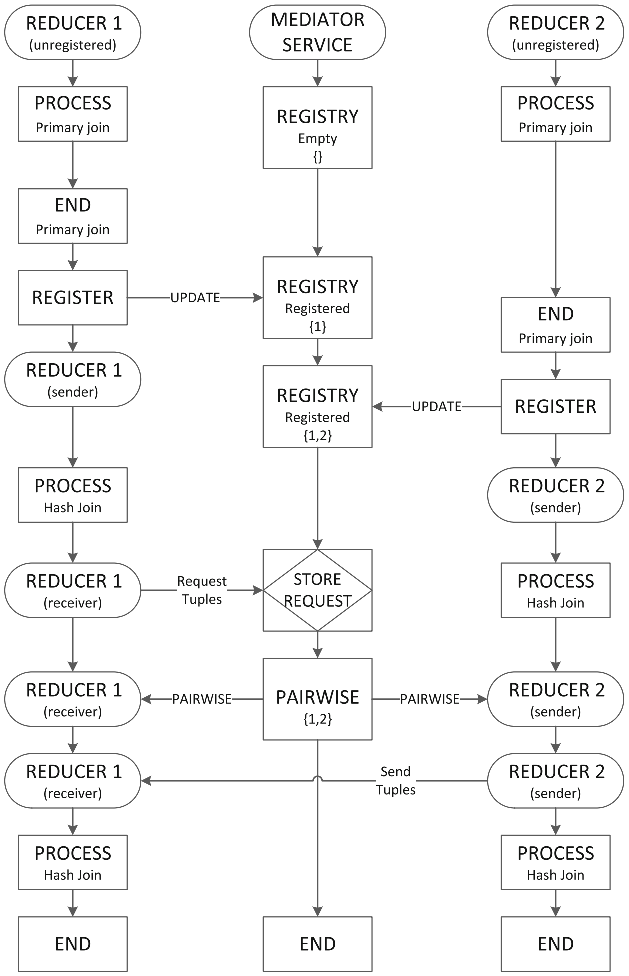


Fig. 3 Multiway join tuple redistribution with SmartJoin.

workload directly between reducers with help of a mediator service, as shown in Fig. 3.

Reduce-side joins and hash joins are both two-way joins. Two-way joins are joins that involve only two tables. Multiway joins are joins involving more than two tables. The multiway join presented in this paper uses a reduce-side join to join the two largest datasets and a hash join on the reducer side to join that result with several smaller datasets. The two largest datasets are partitioned and sent to the various reducers using the typical reduce-side join mechanism.

Unlike the typical multiway join mechanism [9], the smaller datasets are sent to all the reducers. This can be done by duplicating tuples in the mapper phase so that a copy is sent to each reducer, or by using Hadoop's distributed cache mechanism. Therefore, this method is appropriate in situations where the aggregate size of the smaller datasets can fit in the memory of each node used to execute a reduce task.

For the sake of clarity in this paper, we define three types of reducers, unregistered, senders and receivers. The term unregistered refers to reducers that are working on the primary join and have yet to register with the mediator. The term sender refers to reducers that have the potential to send data and refers to those reducers that are working on the hash join and still have tuples to join. The term receivers refer to reducers that have already processed their workload.

Once the mappers have sent the reducers all the tuples, the reducers are able to process the tuples. The two largest datasets are then joined with a traditional reduce-side join. After completing the initial primary join, the reducer registers with the mediator. The reducer then provides the mediator details about how many tuples it has, and details on which node it resides. The mediator then stores this information in its registry. From then on reducers update the mediator periodically their status and the amount of tuples they have left to process. Reducers then continue to process their data using a hash join. At this stage, the reducers are potential senders that may send tuples to receivers if requested. Before a sender sends its tuples to a receiver, both reducers (sender and receiver) are marked as busy in the registry. After they complete the transaction of the tuples, they update the mediator registry to signal that they are no longer busy. Once the hash join has completed all its allocated tuples, reducers become receivers. These receivers then request the mediator to have tuples sent to them. If there are no senders available, the receiver is stored in a list (Table 2).

Once one or more senders are available the mediator determines which senders should send which receivers tuples. Since the time a MapReduce job takes to complete depends on the last reducer to complete its workload, priority is given to senders based on which ones have the most tuples. The pairwise algorithm is describes as follows:

Algorithm 1 Pairwise Algorithm

```
Input:
        SENDER: the reducer to send tuples from.
        LIST_OF_RECEIVERS: list of reducers that have finished their tuples.
Output:
        RECEIVER: the reducer one sends tuples to.
RECEIVER = findReceiverWithLeastDistance(SENDER, LIST_OF_RECEIVERS)
if(RECEIVER.distance == 0 )
{
   select first available RECEIVER in this category.
}
else if (RECEIVER.distance == 2 )
{
   select first available RECEIVER in this category.
}
else if (RECEIVER.distance == 4 )
{
   //select a receiver based on the switch speed of the rack it is on.
   //get racks sorted fastest to slowest, that are slower than the rack the receiver is on
   LIST_OF_SLOWER_RACKS = getSlowerRacks( RECEIVER )

   //get racks sorted slowest to fastest, that are faster than the rack the receiver is on
   LIST_OF_EQUAL_OR_FASTER_RACKS = getFasterRacks( RECEIVER )
   if( LIST_OF_EQUAL_OR_FASTER_RACKS.size()>0)
   {
         RECEIVER = select slowest rack in this list
   }
   else if( LIST_OF_SLOWER_RACKS > 0 )
   {
         RECEIVER = select fastest rack in this list
   }
}
return RECEIVER
```

Table 2 Mediator registry : reducer metadata

Reducer ID	Node ID	Unprocessed tuples	Request tuples (boolean)	Is busy (boolean)	Is sender (boolean)	Is receiver (boolean)	Is registered (boolean)
1	n/a	n/a	n/a	n/a	n/a	n/a	false
2	1	1000	false	false	true	false	true
3	2	800	false	true	true	false	true
4	2	0	true	false	false	true	true

Once a sender has been identified, the mediator finds the receiver nearest to that sender in terms of network distance. When redistributing data in MapReduce one must contend with the network topology. For reasons of both performance and reducing network traffic, the target node should be as close as possible to the source node. Whether two nodes are close together is determined by their network distance. The second factor one needs to contend with when sending data over the network is the speed of the switches on the network. When sending data between racks on the network, the time it takes to send data between racks is dependent on the speed of the slowest switch. Therefore, when sending nodes between racks, the best rack to send data to, is a rack using the same switch speed or faster.

The speed of the switch on each rack is stored in a configuration file. The mediator uses the configuration file to sort racks by their switch speed. In the pairwise algorithm, if there are no receivers available on the same rack, then a receiver is chosen from a different rack. When data is sent between racks, the speed of transmission is determined by the speed of the slowest switch. Therefore, the pairwise algorithm will try to find a receiver that is on a rack with the switch speed equal to or greater to the switch speed of the sender. If the sender can only send tuples to racks that have a slower switch, the pairwise algorithm will give preference to the rack with the fastest (least slowest) switch.

Once the mediator makes a pairing it informs the sender that a receiver is requesting tuples. The sender then sends a batch of tuples directly to the receiver. During the transfer of tuples from the sender to the receiver, the mediator flags both reducers as busy. The mediator then ignores the busy reducers until the transfer of tuples is complete. The redistribution of tuples continues until all tuples have been processed.

The number of tuples transferred in a batch is based on the number of tuples stored on reducers registered with the mediator. The following algorithm calculates the number of tuples a sender sends to a receiver.

Algorithm 2 BatchSize

```
Input:
        SENDER: the reducer from which to send tuples from.
Output:
        RECEIVER: the reducer one sends tuples to.
if( localTuples > averageTuples)
  {
   n = min( localTuples – averageTuples, localTuples*0.5, averageTuples)
   SEND( n )
  }
```

The batchsize algorithm is based on the following rules:

1. localTuples = number of tuples stored on the local reducer
2. averageTuples = average number of tuples stored on each reducer
3. Sender should not send tuples if (localTuples < = averageTuples)
4. Sender should not send more than half the tuples stored locally.
5. Sender should not send more tuples than averageTuples

These rules ensure that when sending tuples to a receiver, the receiver's tuples do not exceed the number of tuples each reducer would need, if it were to have a balanced workload. Furthermore, it ensures that the number of tuples sent is less than the amount that remains on the local sender. This is because it is better to spend the time processing tuples locally, then it is to send a proportionally larger workload to another reducer. Finally, the sender should not send tuples at all if the sender itself has less tuples than or equal the target workload needed for that sender to be balanced. It is better at this point for that sender simply to process all its own tuples, after which it can start to acquire tuples as a receiver instead.

4 Evaluation

4.1 Experiment configuration

To evaluate the performance of the proposed technique, we implemented the SmartJoin method and tested its performance on a simulated MapReduce environment. The simulated environment was built in-house and was based on the Hadoop MapReduce platform [3]. We then evaluated the performance of SmartJoin against tuple redistribution methods that do not take into account the network configuration (Fig. 4).

In order to test our proposed algorithm the MapReduce environment was setup to emulate a cluster of computers. For this purpose, we model a network with three racks. One rack has a 750 Mbps switch for intra rack communication, one rack has a 500 Mbps switch and another rack has a 250 Mbps switch. Connecting these three racks is a 1Gbps switch for inter rack communication, with 10 nodes per rack. Each node on the network contains four reducers, which are executing

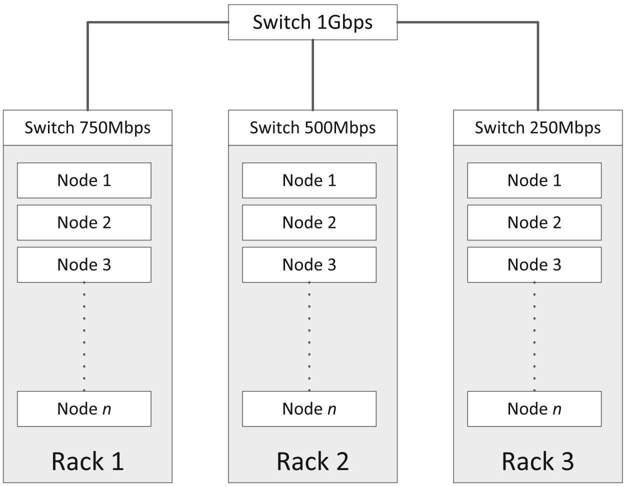


Fig. 4 Experiment network configuration.

on separate cores with identical characteristics. To emulate this environment, switches in the simulator processed tuples at 750, 500 and 250 tuples per second (tps).

4.2 Experiment results

To evaluate the performance of the proposed technique, we implemented the SmartJoin and tested these methodologies using a set of randomly generated input data on a simulated MapReduce environment. We then evaluated the SmartJoin using different CPU speeds, by changing the number of nodes and reducers executing on the network, and by changing the type of loads executing on the network.

The performance of the SmartJoin was compared against systems with no data redistribution, with random reducer selection, and with worst-case reducer selection. Random reducer selection distributes data from the sender with the largest workload to any available receiver picked at random. Random methods were run multiple times and an average result was taken. In worst-case reducer selection (WorstJoin), SmartJoins were modified so that they would pairwise senders and receivers based on the greatest network distance (instead of the least network distance).

Each reducer on the network was loaded with a number of tuples to process. The number of tuples on each reducer initially ranged from a thousand tuples to a million tuples. The initial processing capability of the CPU used in the simulation was then set to 100 tps. The performance of the SmartJoin was then tested and the results recorded. To investigate the efficacy of the SmartJoin at redistributing tuples with different workloads, the minimum load of the reducers was increased and the test was rerun. The loading of the reducers on the network for these tests is shown in Fig. 5.

As shown in Fig. 6, the efficacy of SmartJoin declines as the difference between the minimum load and maximum load decreases regardless of CPU speed. In Fig. 6a there is a case when a SmartJoin takes longer to process the data than a WorstJoin. This is due to their being no free receivers being available when the last unregistered reducer finished its primary workload and became a sender. Later, when a receiver became available, it was on a different rack from the sender. The WorstJoin algorithm managed to reverse this situation by offloading the workload onto different racks first. This meant the only receiver available to the last sender was one that happened to be on the same rack. In most cases, the SmartJoin outperforms the other methods.

The performance of the SmartJoin was then compared against networks with different number of nodes. The simulation used workloads that ranged from a thousand tuples to a million tuples. The average number of tuples on each reducer is initially 380,000 tuples for a network environment containing 10 nodes per rack. The number of nodes on each rack where then changed from 10 nodes to 20 nodes and then finally to 30 nodes. During these tests, the workload is redistributed amongst the nodes so that the total number of tuples used in the test remains unchanged (approximately 46 million tuples).

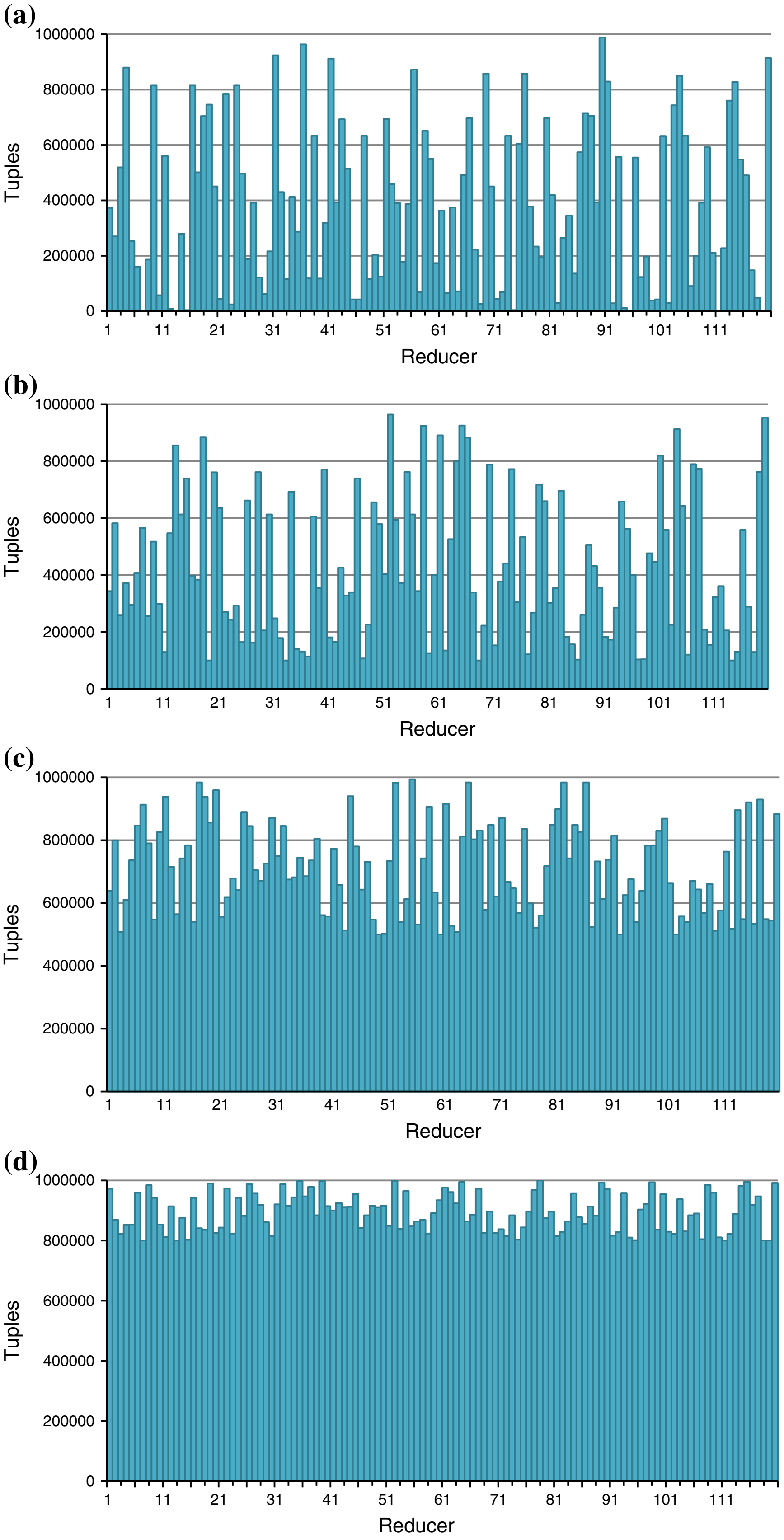
As shown in Fig. 7, the efficacy of SmartJoin seems to increase as the number of nodes increases. As shown in the following table SmartJoin's best performance occurs when there are 30 nodes per rack and the CPU is executing at 200 tps (Table 3).

The performance of the SmartJoin was then compared against networks with different number of reducers per node. The simulation used workloads that ranged from a thousand tuples to a million tuples. The average number of tuples on each reducer is initially 380,000 tuples for a network environment containing 10 nodes per rack. The number of reducers on each node where then changed from 4 reducers to 8 reducers and then finally to 12 reducer. Each reducer is assumed to run on a separate processing core. During these tests, the workload is redistributed amongst the reducers so that the total number of tuples used in the test remains unchanged (approximately 46 million tuples).

As shown in Fig. 8, the efficacy of SmartJoin seems to increase as the number of reducers increases. As shown in the following table SmartJoin's best performance occurs when there are 12 reducers per node and the CPU is executing at 200 tps (Table 4).

Overall, the performance of SmartJoin is better than other tuple redistribution methods. By having more nodes, SmartJoin has more opportunities to match a receiver and sender on the same rack. By having more reducers on each node, SmartJoin has more opportunities to match a receiver and sender on the same node. Consequently, the performance of SmartJoin markedly improves as one adds more nodes and more reducers.

Fig. 5 Workload distribution on reducers. **a** 1K-1M tuples **b** 100K-1M tuples **c** 500K-1M tuples **d** 800K-1M tuples



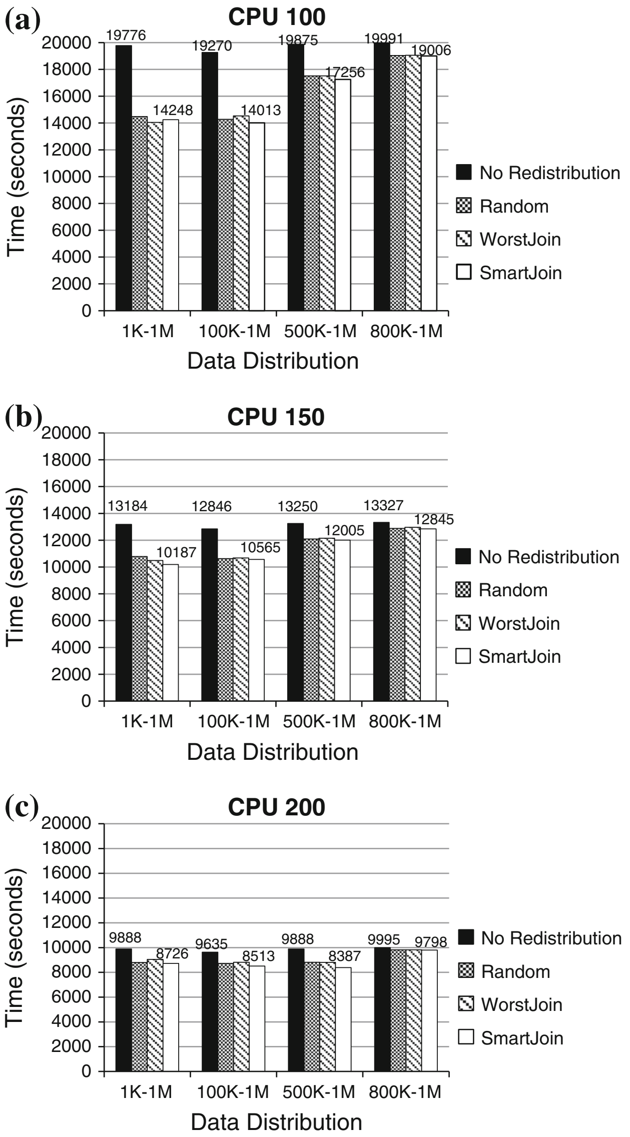


Fig. 6 The affect of workload on SmartJoin performance. **a** CPU 100 tps **b** CPU 150 tps **c** CPU 200 tps

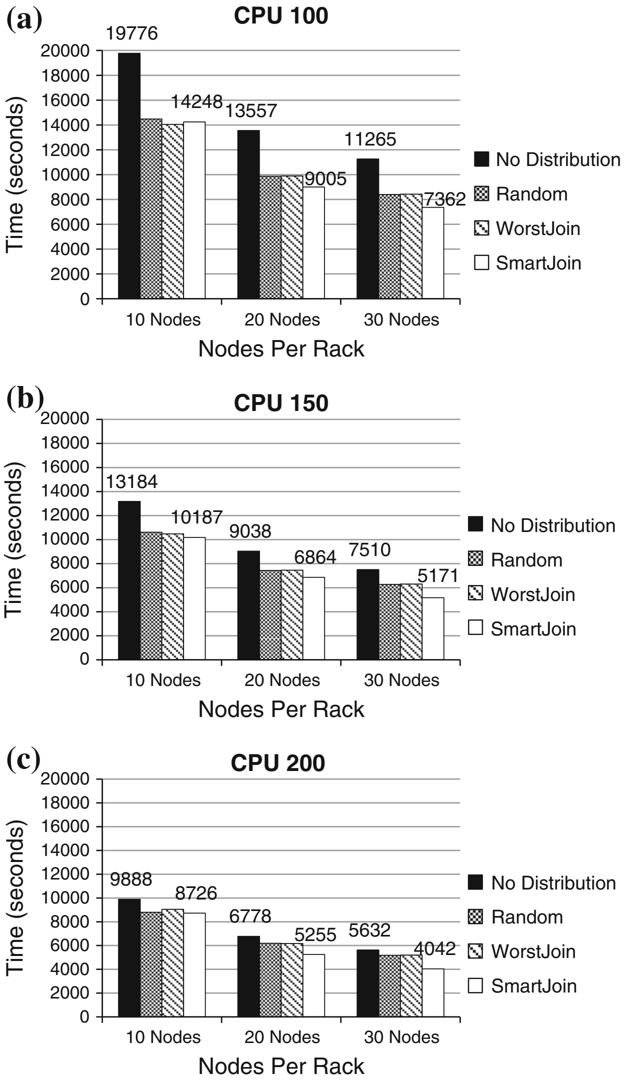


Fig. 7 SmartJoin performance after increasing the number of nodes per rack. **a** CPU 100 tps **b** CPU 150 tps **c** CPU 200 tps

Table 3 SmartJoin performance: different number of nodes per rack

SmartJoin time reduction (%)				
CPU (tps)	Nodes	No distribution	Random	WorstJoin
100	10	28.0	1.6	−1.5
	20	33.6	8.7	8.9
	30	34.6	12.3	12.6
150	10	22.7	4.0	2.8
	20	24.1	7.6	7.9
	30	31.1	17.6	17.9
200	10	11.8	0.8	3.5
	20	22.5	15.1	14.8
	30	28.2	22.0	22.2

5 Related work

Joins have been studied in detail by many sources and investigations and descriptions of various joins have been collated in other works [9,10]. A recent investigation into hash joins by [11] has shown that the hash join algorithm is an efficient algorithm for performing joins in modern multicore processors in main memory environments. The results of the hash join algorithm study showed that a simple hash join technique without partitioning any of its input relations often outperforms other more complex partitioning-based join alternatives. In addition, the relative performance of this simple hash join technique rapidly improves with increasing skew, and it outperforms every other algorithm in the presence of even small amounts of skew. Overall both [9] and [11] indicate hash join as being an efficient method for handling joins on a processor, however it is limited by the memory available to store the hash table.

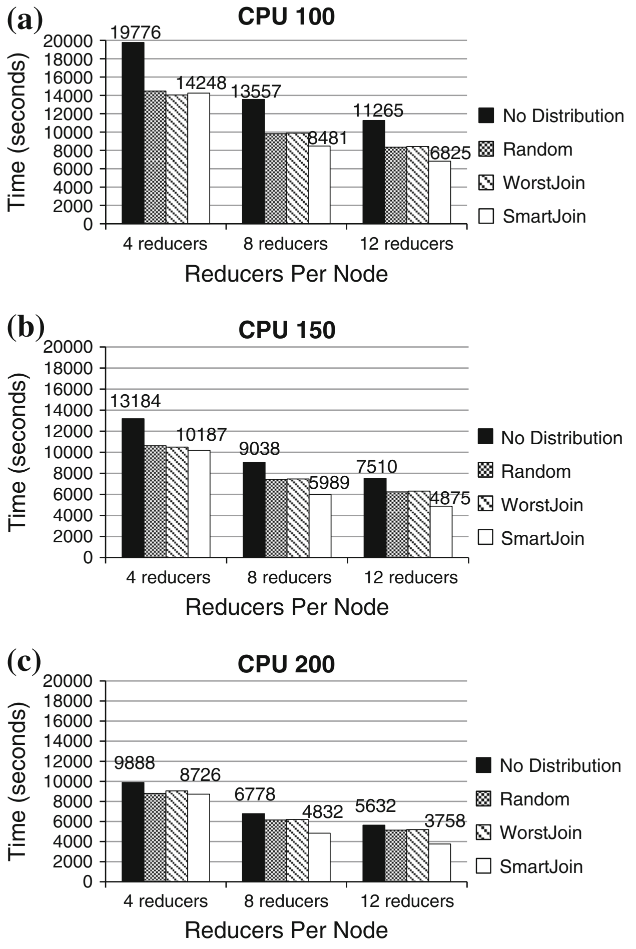


Fig. 8 SmartJoin performance after increasing the reducers per node. **a** CPU 100 tps **b** CPU 150 tps **c** CPU 200 tps

Table 4 SmartJoin performance: different number of reducers per node

SmartJoin time reduction (%)				
CPU (tps)	Nodes	No distribution	Random	WorstJoin
100	10	28.0	1.6	−1.5
	20	37.4	13.6	14.2
	30	39.4	18.2	19.0
150	10	22.7	4.0	2.8
	20	33.7	18.9	19.6
	30	35.1	21.7	22.6
200	10	11.8	0.8	3.5
	20	28.7	21.4	22.0
	30	33.3	26.8	27.6

Data skew has a big impact on overall performance in MapReduce. It is for this reason SmartJoin redistributes its tuples amongst reducers. An alternative technique to tuple redistribution is to preempt the overloading of any particular reducer in the first place. For this purpose, a Skew hANDling join called a SAND join is proposed in [12]. The SAND join is a two-way join that replaces the hash partitioning approach used by reduce-side joins in favor of its own range partitioning approach and samples data before partitioning. This method helps reduce skew amongst the reducers. SmartJoin's approach to workload balancing via tuple redistribution could work in conjunction with this approach.

The use of a bloom filter [13] can reduce the amount of work processed by a join. A bloom filter is used to filter out redundant intermediate records. By filtering out tuples that are not matched in the join, the bloom filter reduces the workload. Consequently, this improves the efficiency of the join. This approach is an orthogonal approach toward MapReduce joins and can be used alongside the approach used by SmartJoin.

Another work that discusses handling joins using a mediator over a network is presented by [14]. This system employs a balanced network utilization metric to optimize the use of all network paths in a global-scale database federation. It uses a metric that allows algorithms to exploit excess capacity in the network, while avoiding narrow, long-haul paths. Another work similar to our paper that handles tuple redistribution in multiway joins is presented by [8] but it uses a distributed queue [15] rather than using peerwise network connections and does not take into account network distance when redistributing tuples.

Awareness of where data is located, is an issue that needs to be considered in MapReduce. This is because the physical location of nodes, processes and data on the network affects MapReduce performance. SmartJoin takes advantage of physical location of nodes when it distributes tuples between reducers. To take advantage of data locality between mappers and reducers Purlieus resource allocation system [16] uses locality awareness in both the map and reduce stages thereby reducing job execution time and reducing network contention within the data center. Data locality is also considered by the Mesos [17] platform, which is a resource allocation system which shares resources in a fine-grained manner amongst different frameworks such as Hadoop and MPI. Mesos allows each framework to achieve data locality by taking turns reading data stored on each machine. As an extension to the Mesos system, an alternative resource allocation system is presented in [18], which attempts to distribute resources fairly in a system containing different resource types and where different applications have different resource requirements. An alternative topology aware resource allocation (TARA) [19] system also takes into account physical location of resources on the physical network in order to optimize allocation of resources for Infrastructure-as-a-Service (IaaS)-based cloud systems. The purpose of TARA is to overcome deficiencies in current IaaS systems, which do not consider the resource requirements of its hosted application and allocated resources independent of its needs. TARA's approach is to incorporate a prediction engine into the resource allocation system. This prevents IaaS providers relying on clients to provide possibly flawed resource requirements. TARA's prediction engine is based on a lightweight simulator that estimates the performance of a specific resource allocation and a genetic algorithm that it uses to search for an optimal solution.

Research in recent years on joins and the MapReduce programming model has resulted in the creation of new programming models to improve join performance in various scenarios. Research by Wang et al [20] investigates how to perform an equi-join on large datasets on a ring architecture distributed system rather than the master-slave architecture distributed system used by MapReduce programming models like Hadoop. Meanwhile, research by Jiang et al [21] extends the MapReduce model to a MapJoinReduce model, which performs filtering-join-aggregation tasks in two successive MapReduce jobs. This approach allows multiple data sets to be joined in one go and avoids frequent checkpointing and shuffling of intermediate results. These methods are unlike the proposed SmartJoin, which builds on top of the pre-existing framework rather than changing the programming model or working environment.

In order to improve joins for multiple datasets a new frame called Llama [22] was created that has a distributed file system that stores data in both row-wise and column-wise format. The developers of Llama noted that the MapReduce model needs to process multiple joins using multiple jobs. Since this requires storing intermediate results of consecutive jobs to a file system like HDFS (Hadoop Distributed File System) it incurs a very high I/O cost. Llama's new DFS allows it to handle joins more efficiently using join algorithms designed to take advantage of its unique DFS.

Researchers have studied MapReduce joins for use in specific applications. Research into matrix multiplication [23] identified how multiway joins can be beneficial when multiplying large matrices on MapReduce as it reduced the number of binary multiplications. Matrix multiplications were performed in this paper by translating a multiplication into a join operation on a database system. Research such as this could be positively impacted by SmartJoin. Other research into MapReduce joins include similarity joins, such as a distance-based similarity self-join [24,25] which can process large vector data sets and the V-SMART-Join [26] that improves performance for all-pair similarity joins for multisets and vectors.

As shown in the literature [21–26], there has been much research into MapReduce joins. Since the join operation is a common data processing task, it is an attractive target for optimization. The need for such optimization will likely continue as researchers try to marry traditional database technologies like SQL with MapReduce [27–32].

6 Conclusion and future work

In this paper, a network aware multiway MapReduce join called SmartJoin is presented that redistributes the workload in a MapReduce job. The simulation results show that SmartJoin can significantly improve tuple redistribution for multiway joins in MapReduce applications. SmartJoin has shown up to 39 % improvement compared to the non-redistribution method, with up to 26.8 % improvement over the random redistribution method and up to 27.6 % improvement over the WorstJoin redistribution method.

SmartJoin is designed for users who intend to perform multiway joins between two large datasets and several smaller datasets with MapReduce. In future work it would be desirable to explore how this system could be extended to handle more than two large datasets and how to improve its performance on different network topologies or hardware configurations. We leave these tasks for future work.

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
2. Hoefler, T., Lumsdaine, A., Dongarra, J.: Towards efficient MapReduce using MPI. Lecture Notes Comput. Sci. **5759**, 240–249 (2009)
3. White, T.: Hadoop the Definitive Guide, 2nd edn. O'Reilly, Sebastopol (2010)
4. Xhafa, F.: Processing and analysing large log data files of a virtual campus. J. Converg. **3**(3), 1–8 (2012)
5. Augusto, J., Callaghan, V., Cook, D., Kameas, A., Satoh, I., Saba, T., Chorianopoulos, K., Howard, N., Cambria, E., Gupta, V.: Intelligent environments: a manifesto. Human-centric Comput. Inf. Sci. **3**(12), 1–18 (2013)
6. Ihm, H.: Mining consumer attitude and behavior, an exploratory study on movie audience attitude extracted from twitter. J. Converg. **4**(2), 29–35 (2013)
7. Afrati, F.N., Ullman, J.D.: Optimizing multiway joins in a map-reduce environment. IEEE Knowl. Data Eng. **23**(9), 1282–1298 (2011)
8. Lynden, S., Tanimura, Y., Kojima, I., Matono, A.: Dynamic data redistribution for MapReduce joins. In: IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom), pp. 717–723 (2011).
9. Chandar, J.: Join Algorithms using Map/Reduce. Master of science. Thesis. School of informatics, University of Edinburgh (2010).
10. Palla, K.: A comparative analysis of join algorithms using the hadoop map/reduce framework. Master of science. Thesis. School of informatics, University of Edinburgh (2009).
11. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: Proceedings of the ACM SIGMOD (2011).
12. Atta, F., Viglas, S.D., Niazi, S.: SAND Join–A skew handling join algorithm for Google's MapReduce framework. In: Multitopic Conference (INMIC), 2011 IEEE 14th, International, pp. 170–175. (2011).
13. Lee, T., Kim, K., Kim, H.J.: Join processing using Bloom filter in MapReduce. In: Proceedings of the 2012 ACM Research in Applied Computation Symposium 2012, pp. 100–105. (2012).

14. Wang, X., Burns, R., Terzis, A., Deshpande, A.: Network-aware join processing in global-scale database federations. In: IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008, pp. 586–595. (2008).
15. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: Wait-free coordination for Internet-scale systems. In: USENIX ATC (2010).
16. Palanisamy, B., Singh, A., Liu, L., Jain, B.: Purlieus: locality-aware resource allocation for MapReduce in a cloud. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis 2011, pp. 58. (2011).
17. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I.: Mesos: A platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation 2011, pp. 22–22. USENIX Association (2011).
18. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: fair allocation of multiple resource types. In: USENIX NSDI (2011).
19. Lee, G., Tolia, N., Ranganathan, P., Katz, R.H.: Topology-aware resource allocation for data-intensive workloads. In: Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems 2010, pp. 1–6. ACM (2010).
20. Wang, X., Shen, D., Nie, T., Kou, Y., Yu, G.: The equi-join processing and optimization on ring architecture key/value database. In: Web Technologies and Applications, pp. 243–254 (2012).
21. Jiang, D., Tung, A.K.H., Chen, G.: Map-join-reduce: Toward scalable and efficient data analysis on large clusters. Knowl. Data Eng. IEEE Trans. **23**(9), 1299–1311 (2011)
22. Lin, Y., Agrawal, D., Chen, C., Ooi, B.C., Wu, S.: Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In: Proceedings of the 2011 International Conference on Management of Data, pp. 961–972. ACM (2011).
23. Myung, J., Lee, S.: Matrix chain multiplication via multi-way join algorithms in MapReduce. In: Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication 2012, pp. 53. ACM (2012).
24. Seidl, T., Fries, S., Boden, B.: MR-DSJ: distance-based self-join for large-scale vector data analysis with MapReduce. In: 15th BTW Conference on Database Systems for Business, Technology, and Web, Magdeburg, pp. 37–56 (2013).
25. Afrati, F.N., Sarma, A.D., Menestrina, D., Parameswaran, A., Ullman, J.D.: Fuzzy joins using MapReduce. In: 2012 IEEE 28th International Conference on Data Engineering (ICDE), pp. 498–509. IEEE (2012).
26. Metwally, A., Faloutsos, C.: V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors. Proc. VLDB Endow. **5**(8), 704–715 (2012)
27. Gowraj, N., Ravi, P.V., Mouniga, V., Sumalatha, M.: S2MART: smart sql to Map-Reduce translators. In: Web Technologies and Applications, pp. 571–582. Springer (2013).
28. Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., Zhang, X.: Ysmart: Yet another sql-to-mapreduce translator. In: 31st International Conference on Distributed Computing Systems (ICDCS), pp. 25–36. IEEE (2011).
29. Xu, Y., Hu, S.: QMapper: a tool for SQL optimization on hive using query rewriting. In: Proceedings of the 22nd International Conference on World Wide Web Companion, pp. 211–212. (2013).
30. Lu, J., Guting, R.H.: Parallel secondo: boosting database engines with hadoop. In: IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), pp. 738–743. IEEE (2012).
31. Chung, W.-C., Lin, H.-P., Chen, S.-C., Jiang, M.-F., Chung, Y.-C.: JackHare: a framework for SQL to NoSQL translation using MapReduce. Autom. Softw. Eng. 1–20 (2013).
32. Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**(1), 64–71 (2010)

Kenn Slagter received an NZCE in electronics and computer technology from the Eastern Institute of Technology in 1996, a B.S. degree in Computer Science from the University of Waikato in 2000 and a Master of Computer Studies from the University of New England in 2007. In 2008 he joined the Department of Computer Science at National Tsing Hua University as a PhD candidate. He has also over 8 years work experience in the private sector as a software engineer. His research interests include high performance computing, cloud computing and parallel & distributed systems. He is a student member of the IEEE computer society.

Ching-Hsien Hsu received B.S. and Ph.D. degrees in Computer Science from Tung Hai University and Feng Chia University, Taiwan, in 1995 and 1999, respectively. From 2001 to 2002, Dr. Hsu had been an assistant professor in the department of Electrical Engineering at Nan Kai College. He joined the department of Computer Science and Information Engineering, Chung Hua University in 2002, and has become an associate professor since August 2005. Doctor Hsu has published more than 100 academic papers in journals, books and conference proceedings. His research interests include parallel and distributed processing, concurrent programming, parallelizing compilers, grid and pervasive computing. He is a senior member of the IEEE computer society.

Yeh-Ching Chung received a B.S degree in Information Engineering from Chung Yuan Christian University in 1983, and the M.S. and Ph.D. degrees in Computer and Information Science from Syracuse University in 1988 and 1992, respectively. He joined the Department of Information Engineering at Feng Chia University as an associate professor in 1992 and became a full professor in 1999. From 1998 to 2001, he was the chairman of the department. In 2002, he joined the Department of Computer Science at National Tsing Hua University as a full professor. His research interests include parallel and distributed processing, cloud computing, and embedded systems. He is a senior member of the IEEE computer society.

Gangman Yi received his master's degree in Computer Science at Texas A&M University, USA in 2007, and received his doctorate in Computer Science from Texas A&M University, USA in 2011. In May 2011, he joined System S/W group in Samsung Electronics, Suwon, Korea. Finally, Dr. Yi joined the Department of Computer Science & Engineering, Gangneung-Wonju National University, Korea in March 2012. Dr. Yi's research interests cover an interdisciplinary field. However, his research focuses especially on the development of computational methods to improve understanding of biological systems and associated big data. Dr. Yi actively serves as a managing editor and reviewer for international journals, and chair for international conferences and workshops.