Design and evaluation of small–large outer joins in cloud computing environments

Design and Evaluation of Small-Large Outer Joins in Cloud Computing Environments

Long Cheng[a,b,*], Ilias Tachmazidis[c], Spyros Kotoulas[d], Grigoris Antoniou[c]

[a] *Eindhoven University of Technology, The Netherlands*
[b] *TU Dresden, Germany*
[c] *University of Huddersfield, UK*
[d] *IBM Research, Ireland*

Abstract

Large-scale analytics is a key application area for data processing and parallel computing research. One of the most common (and challenging) operations in this domain is the *join*. Though inner join approaches have been extensively evaluated in parallel and distributed systems, there is little published work providing analysis of outer joins, especially in the extremely popular cloud computing environments. A common type of outer join is the small-large outer join, where one relation is relatively small and the other is large. Conventional implementations on this condition, such as one based on hash redistribution, often incur significant network communication, while the duplication-based approaches are complex and inefficient. In this work, we present a new method called DDR (duplication and direct redistribution), which aims to enable efficient small-large outer joins in cloud computing environments while being easy to implement using existing predicates in data processing frameworks. We present the detailed implementation of our approach and evaluate its performance through extensive experiments over the widely used MapReduce and Spark platforms. We show that the proposed method is scalable and can achieve significant performance improvements over the conventional approaches. Compared to the state-of-art method, the DDR algorithm is shown to be easier to implement and can achieve very sim-

*Corresponding author.
Email addresses: l.cheng@tue.nl (Long Cheng), i.tachmazidis@hud.ac.uk (Ilias Tachmazidis), spyros.kotoulas@ie.ibm.com (Spyros Kotoulas), g.antoniou@hud.ac.uk (Grigoris Antoniou)

ilar or better performance under different outer join workloads, and thus, can be considered as a new option for current data analysis applications. Moreover, our detailed experimental results also have provided insights of current small-large outer join implementations, thereby allowing system developers to make a more informed choice for their data analysis applications.

Keywords: parallel joins, outer joins, small-large joins, cloud computing, performance evaluation.

---

1. Introduction

In light of the explosion of available data and the increasing connectivity between data systems, the infrastructure for scalable data analytics is as relevant as ever. An essential operation in this domain is the *join*, which facilitates the combination of records based on a common join key. This data-intensive operation can incur significant costs, in terms of communication and computation. Improving the efficiency of this operation can have a significant impact on the performance of applications mainly for analytical workloads [1].

Although inner join algorithms have been widely studied in parallel and distributed systems [2], [3], [4], [5], there has been relatively little done on the topic of outer joins. In fact, outer joins are common in complex queries and widely used in various applications [6]. An example domain where outer joins are particularly important is the Semantic Web: queries containing outer joins account for as much as 50% of the total number of queries, based on the analysis of DBpedia query logs [7]. In general, in AI-related fields and Cognitive computing applications, there is a trend towards computation with sparse, semi-structured data. Data in this domain lack normalisation, which makes outer joins as relevant as ever.

Inner join implementation can discard records with keys that do not match the keys from the other side of the join as soon as this is discovered. For outer joins, this decision needs to be taken at a later stage, since the final join results contain not only the matched part but also the non-matched part. For example, for a left outer join ($\overline{\bowtie}$) between two input relations $R(a, x)$ and $S(b, y)$ on their attributes a and b, the final output contains not only the matched records in the form of $< a, x, y >$, for a match condition, but also $< a, x, null >$, when values do not match.

Recently, several approaches for efficiently implementing outer joins have been proposed [8], [9]. However, such methods focus on large-large table

outer joins, while a special case, namely small-large outer joins, seems to have been neglected. In fact, small-large outer joins are quite common in real applications. For example, in business intelligence, a small record with customer ids often left outer join with a large transaction record, to analyze purchase patterns [10].

Most of the small-large outer join implementations still rely on the two conventional outer join algorithms [10], namely the redistribution outer join algorithm (ROJA) and the duplication outer join algorithm (DOJA). However, as we will explain later, both methods could meet performance issues. The state-of-the-art algorithm DER [10] (duplication and efficient redistribution), which has been designed specifically for small-large outer joins, has achieved significant performance improvements over the conventional approaches. Regardless, the DER algorithm is specifically designed for relational database management systems (RDBMSs), with its implementation heavily relying on the schema of the input data (as explained later), which could require several modifications for other environments such as the currently popular cloud computing platforms that are studied in this work.

In fact, with data applications growing in scale, cloud environments play a key role in application scale-out, exploiting parallelisation to speed up operations and extending the amount of available memory available to developers [11]. In such scenarios, efficient parallel implementation of the small-large outer joins over scale-out data platforms, best suited to cloud environments (such as MapReduce [12] and Spark [13]), is increasingly desirable. Such platforms are frequently preferable for certain non-transactional workloads, since they allow for easy deployment and straightforward scale-out capability, compared to the conventional parallel RDBMSs [14]. In fact, most vendors provide, either on-premise or on the cloud, solutions that compute massive volumes of structured, semi-structured and unstructured data for their business applications [11].

Motivated by the application domain mentioned above, in this paper, we present a new approach called DDR (duplication and direct redistribution). Our approach is aimed at efficiently performing small-large outer joins over distributed systems. We show that DDR is easier to implement for cloud-based computing frameworks and can achieve similar performance, compared to the state-of-the-art algorithm. To the best of our knowledge, this is the first work specified on the detailed design and evaluation of small-large outer joins in cloud computing environments. Here, we summarize the contributions of this work as follows:

- We adapt the state-of-the-art approach called DER to a cloud computing environment and present the detailed implementation of the variant.

- We propose a new approach called DDR (duplication and direct redistribution) aimed at efficient small-large outer joins in cloud computing environments. We show that DDR is easier to implement than DER and does not require any customization of existing operations.

- We present a highly efficient design for the implementations of DER and DDR over the MapReduce framework, which utilizes various details of MapReduce job execution, making that both approaches can be done within a single MapReduce job.

- We conduct a detailed experimental evaluation of our method and compare its performance with current approaches. Experimental results show that DDR is scalable, and can clearly outperform conventional approaches with achieving performance similar to the state-of-art DER method. Thus, DDR can be considered as a new option for data analytics in large-scale distributed scenarios.

- We also examine the potential factors of performance by using different inputs in our evaluations, the results characterizing the performance of current small-large outer join implementations as well as their differences over different underlying platforms will provide helpful information on query optimization for data analytics in real applications.

Additionally, in contrast to the original DER implementation, which is protected by a patent [15], our approach as well as the algorithms provided in this manuscript do not have such restrictions, which could make our approach more attractive. The rest of this paper is organized as follows: In Section 2, we analyze current outer join implementations, including the state-of-the-art DER method. In Section 3, we introduce our DDR approach and compare its details with the DER algorithm. In Section 4, we present the detailed implementation of our approach over current cloud-based computing platforms. Section 5 describes the experimental framework while Section 6 provides the experimental results. We report on related work in Section 7 and conclude in Section 8.

2. Background

In this section, we introduce the current outer join approaches and discussing their possible limitations. Moreover, we also present a variant implementation of the state-of-art DER method, in order to make it applicable for cloud computing environments.

We focus on left outer joins in the following, since they are the most commonly used outer joins. In addition, to capture the core performance of queries, we focus on a single join operation between two input relations R and S over a n-node system[1]. We assume that both relations are in the form of <key, value> pairs, where *key* is the join attribute. In the meantime, we assume that the relation R is smaller than the relation S.

2.1. Conventional Outer Join Methods

2.1.1. ROJA

The implementation of the redistribution outer join algorithm is very similar to inner joins, and consists of the following two main steps:

- *Step 1.* The initially partitioned relations R_i and S_i at each node i are partitioned into distinct sets R_{ik} and S_{ik} respectively, normally according to the hash values k of their join key attributes. Then, each of these sets is transferred to a corresponding remote node. For example, tuples in R_{ik} and S_{ik} at node i will be transferred to the k-th node.

- *Step 2.* A local outer join between received R_k (i.e., $\bigcup_{i=1}^{n} R_{ik}$) and S_k (i.e., $\bigcup_{i=1}^{n} S_{ik}$) at each node k is implemented in parallel to formulate the final outputs.

2.1.2. DOJA

The implementation of the duplication outer join algorithm has significant differences compared to inner joins. It contains two main stages as follows.

- *Stage 1.* The tuples in R_i at each node are firstly duplicated (broadcast) to all nodes. Then, an inner join between received $\bigcup_{i=1}^{n} R_i$ (i.e., R) and locally kept S_i is implemented in parallel at each node i. This formulates an intermediated result I_i at each node.

[1]Here, we focus on explaining the join pattern in a distributed environment. In terms of terminology, a node here means a computing unit (e.g., a Reducer in MapReduce or an execution core in Spark).

- *Stage 2.* An outer join between the relation R and the intermediate join results I_k to construct the final outputs, based on the ROJA approach described above.

2.1.3. Discussion

Every step for the two approaches above can be implemented in parallel across the computing nodes, and the number of execution units can be increased by deploying additional machines. Therefore, both algorithms show the potential ability on scale-out processing of large input datasets. Moreover, since their join patterns only reply on the operators of redistribution and duplication, which can be easily done in a cloud computing environment, they can be applied in cloud-based data analytics directly. However, with regards to the detailed performance, for a small-large case, the ROJA algorithm could be expensive as it has to redistribute the large relation. Moreover, in the presence of data skew [16], load balancing would be a problem, thus deteriorating the overall performance. Meanwhile, for the DOJA algorithm, though the broadcast cost could be small, such a scheme could still encounter performance bottlenecks. The reason is that the intermediated result I, which has to be redistributed in the *stage 2*, could be very large in some cases (e.g., Cartesian product) [8]. In such scenarios, an efficient small-large outer join implementation suitable for a cloud computing environment is becoming desirable.

2.2. State-of-the-art DER

The DER algorithm (*duplication and efficient redistribution*) is the state-of-art method specified for optimizing small-large table outer joins in RDBMSs. As shown in Figure 1, for a 2-node system, the detailed implementation of this algorithm comprises of four main steps as follows.

- *Step 1.* Following the DOJA algorithm, tuples of R_i at each node i are duplicated to all other computing nodes in this step.

- *Step 2.* A local left outer join between the received tuples of R and locally kept S_i is implemented in parallel at each node after the duplication. In contrast to a conventional approach, the outer join here is *customized*, recording *ids* of all non-matched rows of R in this step[2].

[2]For a RDBMS, each row has a unique id, thus we can retrieve the required ids directly based on the non-matched results.

- *Step 3.* After the local outer joins, the extracted ids at each node will be redistributed among nodes based on their hash values.
- *Step 4.* As demonstrated, this step contains two main operations:
  - The received ids at each node are counted, if the number of times an id appears is equal to the number of computing nodes n (i.e., 2 in this example), then this row-id will be recorded.
  - After that, all tuples in R with the recorded row-ids will be extracted to formulate the non-matched results.

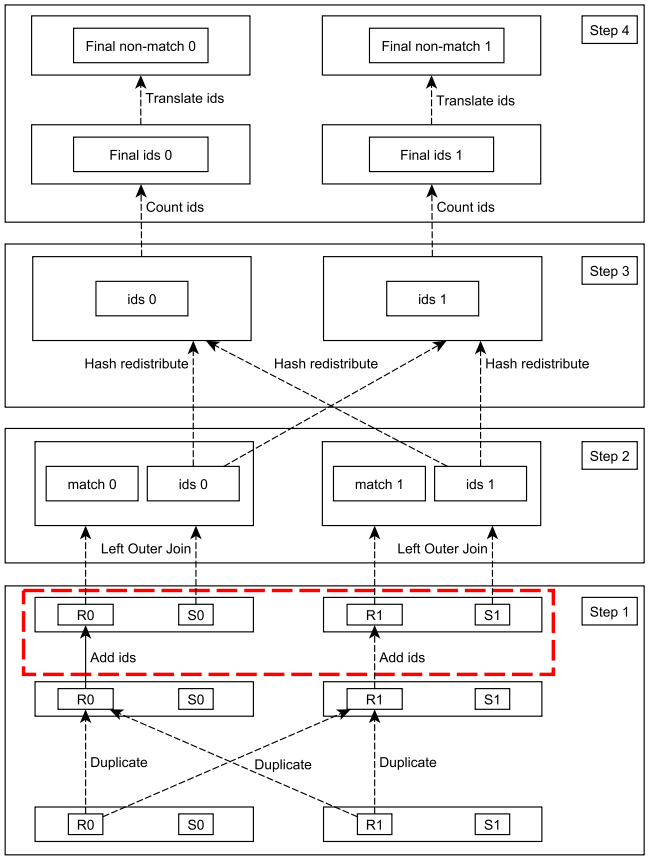


Figure 1: The implementation flow of the DER algorithm over a 2-node system. The additional operation in the dash square is only specified for a cloud computing environment as described in Section 2.3.

The final output is the union of the matched results in the second step and the non-matched results in the fourth step. In fact, DER presents a very efficient way to extract non-matched results. Note that the *join* in the first step of the DOJA algorithm is an **inner** join rather than an outer join, the reason is that a direct outer join would bring either redundant or erroneous non-matched outputs [8]. The DOJA approach alleviates this problem by redistributing all the inner join results. In comparison, DER uses a more efficient way, in that each tuple can be indicated by a row-id from the relation R, which is redistributed. As the transferred ids are always very small (because the number of ids is smaller than $|R|$), the network communication can be greatly reduced. Additionally, since non-matched tuples can be retrieved by ids directly as described in the *step 4*, the local computing cost could be also reduced. The experimental results presented in [10] have shown that the DER algorithm can achieve significant speedups over current methods on small-large outer joins in a parallel DBMS.

2.3. Applying DER in Cloud Environments

It can be seen that row-ids for the tuples in relation R are critical to successfully implementing the DER algorithm. In RDBMSs, it is common that each tuple has an id. However, for a cloud computing environment, normally there will be no id for each tuple that participates in an outer join. The main reason is that relations usually come from different data sources and lack schemas.

For this condition, we add an extra operation in the *step 1* of the DER approach, so as to make it applicable to a cloud environment. As shown in the dashed square in Figure 1, we explicitly assign a unique id for each tuple in the relation R and use the ids for the subsequent join executions. Since the relation R is small for a small-large join case, this additional operation would be extremely lightweight, and will not affect significantly the performance of the original DER implementation. We will show that this new implementation still performs efficiently on small-large outer joins (for details see Section 6). For simplification, since in this work we focus on approaches in cloud computing environments, in the following we will refer to this variant of the original DER algorithm as DER.

3. Our Approach

In this section, we first describe the proposed DDR approach and its detailed work flow. Then, we conduct a comprehensive comparison of this method with the state-of-art DER algorithm.

3.1. The DDR Algorithm

Duplication is usually the ideal operation for processing small-large joins. In such a case, for outer join implementations, the core challenge becomes how to efficiently identify the correct non-matched results. To achieve this, our DDR approach directly outputs the non-matched results and then redistributes them. The details of our implementation are shown as Figure 2. Similar to the DER algorithm, DDR also consists of four main steps:

- *Step 1.* This step is similar to the first step of DOJA where all tuples of R_i at each node are broadcast.
- *Step 2.* A common left outer join between received R_i (i.e., R) and locally kept S_i is implemented in parallel at each node i. In this operation, the matched results will be output directly and the non-matched results will be recorded.
- *Step 3.* The non-matched results at each node are redistributed to all nodes based on their hash values.
- *Step 4.* The received records at each node are counted, if the number of times a record appears is equal to the number of computing nodes, then this record will be considered as one of the final non-matched results.

Compared to the conventional DOJA algorithm, it can be seen that we focus on redistributing the non-matched results in *step 2* in order to eliminate possible errors and redundancies. Evidently, our implementation adopts the most common duplication and redistribution operations, thus it can be directly implemented on a RDBMS. Moreover, compared to DER algorithm, we do not rely on any id information, and thus, DDR can be used directly on a cloud computing platform.

We first *duplicate* all the tuples of R, then we identify the non-matched results by another efficient way—*direct redistribution*, compared to the DER

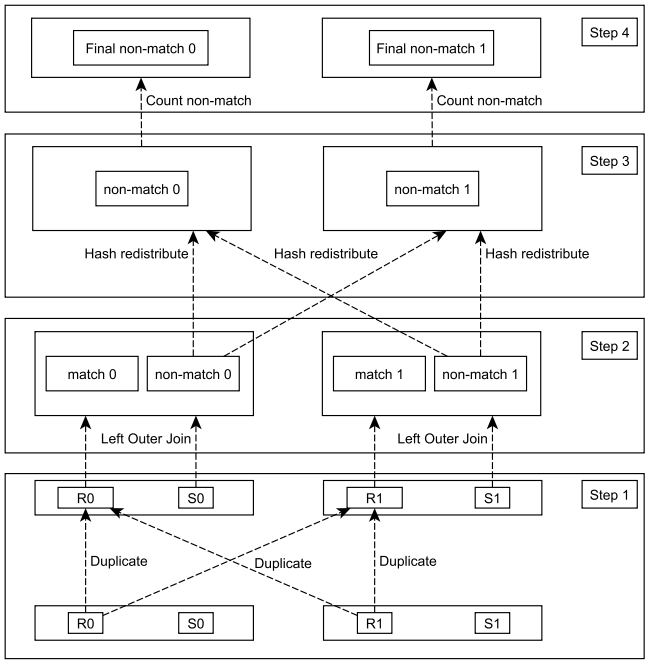


Figure 2: The implementation flow of the DDR algorithm.

approach, hence we named our approach DDR (duplication and direct redistribution). We will show later that DDR is very efficient on processing small-large outer joins.

In fact, the proposed DDR algorithm can be applied to small-large *full outer joins* as well. For example, if we change the left outer join to a full outer join in our example, the only change in our DDR implementation is that we only need to change the local left outer join implementation in the *step 2* to full outer join and then *directly* output the non-matched tuples of S as part of the final results. The reason is that each tuple in S appears only once as well as at one node, while each S_i examines the non-matched tuples over the *full* relation R at each node, which means that there will be no error or redundant non-matched results from S. Here, we do not consider the right outer joins since they can be rewritten to equivalent left outer joins [10]. Recall that because left outer joins are the most popular outer join operations, we only focus on the performance of such operations in our evaluations in Section 6.

3.2. Compared to the State-of-the-art DER

We compare the proposed DDR method with the presented DER algorithm in two aspects. Firstly, we focus on their general implementations. Then, we concentrate on the performance comparison between the two methodologies at a theoretical level.

3.2.1. High-level Implementation

We focus on efficiently processing small-large outer joins. Compared to the DER implementation, it can be seen that our DDR approach has several clear differences, which mainly include the following four parts.

- In *step 1*, the DER approach has to use extra operations to assign the row-id to each tuple, but DDR does not need to do so and the input tuples can be processed directly.
- In *step 2*, to get the non-matched row-ids, for the DER algorithm, we have to customize the local outer join implementation and use the relation R to search over S. In comparison, the proposed DDR approach does not have such limitations.
- In *step 4*, the DER approach has to convert the non-matched ids to the corresponding tuples by accessing the relation R. In contrast to that, our DDR approach does not need such operation because we can get the non-matched tuples directly.
- In *step 2, 3 and 4*, the DER approach has to keep the duplicated R in memory through the whole process, because in its *step 4*, the final non-matched results have to be retrieved from R, as described[3]. In contrast, after the left outer joins in the *step 2* of our DDR approach, the relation R will no longer be required for the remaining steps and thus can be discarded immediately.

[3]Note that, we can also discard R directly or keep R in disks in the DER implementation, however, these operations will bring extra costs on distributed joins or system I/O in the final step. Such implementation is required by MapReduce as each Mapper or Reducer initializes its memory prior to processing any given subset. However, for Spark, in order to keep reduce runtime, in our implementations, we have chosen to keep R in memory, since we believe that R is small and this will not bring memory pressure for the underlying system in a small-large join case.

In this work, we focus on implementing our approach in a cloud computing environment. More specifically, we are interested in applying our approach in the most popular platforms, namely MapReduce [12] and Spark [13]. In such settings, the above differences could make our DDR algorithm easier, more flexible and more suitable for a real application, in terms of detailed implementations.

For example, the Spark platform has provided several join APIs, and thus, can be used directly in DDR. In comparison, an implementation of DER needs customized local outer joins, as we have described above. More importantly, in a large-scale distributed scenario, large computational resources would be tapped in a short time, which requires very fast data loading of the target dataset(s). In turn, to shorten the data processing life-cycle for each query, exploration and analysis should be done in an interactive manner. In such scenarios, additional operations could bring more synchronizations and require more strict fault-tolerant mechanisms, and consequently impact the system performance or make the implementation challenging.

3.2.2. Performance Analysis

In order to examine the performance difference between the proposed DDR algorithm and the state-of-art DER approach as well as investigate the possible impact factors of the difference, we conduct here a concise theoretical analysis. We are interested in the performance of join implementations rather than other operations of a join such as data loading and result materialization. Considering the whole time cost in an outer join, there are generally two kinds of time costs in a distributed implementation: (1) cost on local operators, which do not contains any inter-machine communication; and (2) cost on network communication, which do involve data shipping over networks. For simplification, we assume that data is always uniformly distributed in each redistribution operation. In the meantime, the local joins adopt the commonly used hash operators[4] (i.e., hash table building & hash table probing [18]). For convenience, we use the notations in Table 1.

Following the implementation of DER algorithm, the time cost for an outer join implementation C_{DER} at each node will be composed by six parts

[4]When calculate the time cost of local joins shown in the equations below, the number of whole hash operations of a join between A and B will be $|A| + |B|$, because of the hash operations on tuple inserting and searching. This is an ideal condition, that the inserting and searching of a tuple is done in O(1) time with a hash function [17].

Table 1: Table of notations

Notation	Meaning
T	*a join participated relation (R or S)*
$\|T\|$	*the number of tuples in T*
n	*the number of computing nodes in the system*
t	*average transmission time for a tuple in size*
α	*the join selectivity (represented by the non-matched ration of R)*
j	*average hash time table building/probing time for a tuple*
m	*processing time on mapping a tuple to an id or reverse*
l	*the ration of the length of an id to a tuple* ($l < 1$)
c	*average time of count operator for a tuple*

as following:

$$C_{DER} \begin{cases} t \times |R| & \text{(broadcast)} \\ m \times |R| & \text{(id assignment)} \\ j \times (|R| + \frac{|S|}{n}) & \text{(local join)} \\ t \times \alpha|R| \times l & \text{(id redistribution)} \\ c \times \alpha|R| & \text{(count)} \\ j \times (1+\alpha)|R| & \text{(retrieve tuple)} \end{cases} \tag{1}$$

Similarly, the time cost of DDR implementation contains the following four parts:

$$C_{DDR} \begin{cases} t \times |R| & \text{(broadcast)} \\ j \times (|R| + \frac{|S|}{n}) & \text{(local join)} \\ t \times \alpha|R| & \text{(tuple redistribution)} \\ c \times \alpha|R| & \text{(count)} \end{cases} \tag{2}$$

From above two equations, it can be seen that for a given underlying platform (i.e., the values of t, j, m and c are fixed), higher values for α, $|R|$ and $|S|$ will lead to longer runtimes (higher time costs) for both the DER and DDR algorithms. Thus, reducing the join selectivity and increasing the size of two inputs will result in increased join execution time. This is reasonable, as both the local computing and network communication costs are increased

under these conditions. Moreover, compared the Equation (1) and (2), we have that the time cost difference between DER and DDR is:

$$\begin{aligned}\Delta C &= C_{DER} - C_{DDR} \\ &= m \times |R| + t \times \alpha|R| \times (l-1) + j \times (1+\alpha)|R| \\ &= |R| \times (m - \alpha t \times (1-l) + j \times (1+\alpha)) \\ &= |R| \times \delta\end{aligned}$$

We can see that the value of δ is critical for ΔC, which indicates when an algorithm is faster. In more detail, it is possible that:

- DER will outperform DDR when the value of $\alpha t \times (1-l)$ in δ is comparably larger, namely $\Delta C < 0$. This could happen in conditions where the non-matched ration is high, the length of each tuple is large and the network transmission rate is low. For a given input with fixed α and l, the larger the t is, the smaller the ΔC will be. This means that the performance advantage of DDR is more network bounded. The main reason for this is that DDR redistributes the non-matched results in the form of tuples, which brings in more network communication compared to DER, in which only row-ids are redistributed.

- On the other hand, DDR will outperform DER when the value $m + j \times (1+\alpha)$ is comparably larger. Namely, ΔC would be positive in the conditions that the values of m, j and α are large. The former two parameters are highly related to the capability of system CPUs and memory access, for a fixed input, in contrast to DDR, the performance advantage of DER will be more CPU and memory bounded.

For a given outer join workload, the above analysis has shown that DER and DDR algorithms would beat each other over systems with different hardware configurations. Regardless, for current computing platforms, especially in large-scale distributed cases, the values of t, j and m are always very small, and thus, the value of δ will be small. In the meantime, when $|R|$ is small enough for an ideal small-large case, then, we have that $\Delta C \approx 0$. This means that the proposed DDR approach will generally perform at the same level with the state-of-art DER algorithm for a small-large outer join execution. As we will show in our experimental results in Section 6, under different outer join workloads (with different values on $|T|$, α and l), over a

commodity computer cluster, DDR generally achieves the same performance as the state-of-art DER algorithm.

4. Implementation

We have implemented our approach over two of the most popular platforms—MapReduce and Spark, in terms of cloud computing. Here, we only present the details of our implementations over the former framework. We have two reasons for this choice: (1) join operations can be challenging in MapReduce [18]; and (2) MapReduce paradigm is *highly sensitive* to the number of jobs [19]. Thus, it brings more challenges on the DER and DDR algorithms, as they have more steps than the ROJA and DOJA algorithms. On the other hand, Spark has provided abstract level operations, where various data-parallel applications, initially designed for MapReduce, can be expressed and executed efficiently using Spark.

In the following, we first introduce the related MapReduce and HDFS systems, and then we present the detailed implementation of our DDR algorithm as well as the DER approach.

4.1. Overview of MapReduce and HDFS

MapReduce is a framework for parallel processing over huge datasets [12]. The data is typically stored in the underlying Hadoop Distributed File System (HDFS) [20] and each unit of processing (a job) is carried out in two phases, a map and a reduce phase. For each phase, a set of user-defined map and reduce functions are run in parallel. The former performs a user-defined operation over an arbitrary part of the input and partitions the data, while the latter performs a user-defined operation on each partition.

MapReduce is designed to operate over key/value pairs. Specifically, each *Map* function receives a key/value pair and emits a set of key/value pairs. All key/value pairs produced during the map phase are grouped by their key and passed to reduce phase. During the reduce phase, a *Reduce* function is called for each unique key, processing the corresponding set of values. More details and examples have been given in the tutorial at `http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`.

HDFS is designed to provide high throughput access to large-scale data, which is critical to the implementations of MapReduce programs. Internally, a data file is split into one or more blocks and these blocks are stored across different computing nodes. Compared to other file systems, HDFS caters

to cloud computing environments since it is highly fault-tolerant. There are mainly two reasons for this [21]: (1) data replication across machines in a large cluster ensures that very large files can be reliably stored. When part of the data on a node is lost, replicas stored on other nodes will still be accessible; and (2) HDFS has a master/slave architecture and applies heartbeats between the master node and slaves to check the availability of each node. Currently, HDFS also provides a robust and convenient file system for Spark [11].

4.2. Implementation with MapReduce

4.2.1. Challenges

In order to solve a given problem using MapReduce, it should first be transformed so as to follow the MapReduce paradigm. However, this may result in an implementation requiring multiple number of jobs, which in turn could deteriorate significantly the overall performance as multiple iterations over the input lead to I/O and communication overheads [18, 19, 22]. Since both DER and DDR are more complex than the two conventional approaches, namely ROJA and DOJA, a straightforward implementation may result in multiple jobs. In contrast, ROJA and DOJA can easily be implemented within a single MapReduce job.

Unlike the RDBMSs, careless design of MapReduce jobs will impair the advantage of DER and DDR over the two conventional approaches. Thus, in order to retain the same level of performance, we designed a highly efficient algorithm for DER and DDR, which utilizes various details of MapReduce job execution, resulting in both DER and DDR being computed within a *single* MapReduce job. We have also implemented ROJA and DOJA in a single MapReduce job. In addition, we have made our code, for the four evaluated algorithms in this work, publicly available at `https://github.com/longcheng11/small-large`.

4.2.2. Detailed Implementation

As shown in Algorithms 1 and 2, we present a high-level pseudo-code of our implementation over MapReduce. In this way, we provide the insights that allowed us to implement both DER and DDR within a **single** MapReduce job like the ROJA implementation. *Step 1* is implemented in the "setup(Context *context*)" of Map (see Algorithm 1), where relation R is loaded for each Map directly from HDFS (see Algorithm 1, line 1). Note that relation R is kept in-memory (r_Map), which is modeled as a map of maps in

Algorithm 1 DER and DDR algorithm over MapReduce - Steps 1 and 2

```
    // Step 1 of DDR/DER
    setup(Context context):
    // matched = false
    // r_Map (DDR): R.a → R.x → matched
    // r_Map (DER): R.a → R.x → id → matched
 1: r_Map = load_R_from_HDFS()

    // Step 2 of DDR/DER
    map(Long key, String value):
    // key: position in document (irrelevant)
    // value: document line (tuple)
 2: if value.predicate == "S" then
 3:     if r_Map.contains(value.b) then
 4:         for all R.x ∈ r_Map.get(value.b).keySet() do
                // output matched results directly to HDFS
 5:             emit("<value.b, R.x, value.y>","")
                // mark matching tuples in r_Map
    // DDR:
 6:             r_Map.get(value.b).put(R.x, true)
    // DER:
 7:             for all id ∈ r_Map.get(value.b).
                                        get(R.x).keySet() do
 8:                 r_Map.get(value.b).get(R.x).put(id, true)
 9:             end for
10:         end for
11:     end if
12: end if

    cleanup(Context context):
    // output non-matched tuples of R to Reduce
13: for all R.a ∈ r_Map.keySet() do
    // DDR:
14:     if r_Map.get(R.a).containsValue(false) then
15:         for all R.x ∈ r_Map.get(R.a).keySet() do
16:             emit("<R.a, R.x, null>","")
17:         end for
18:     end if
    // DER:
19:     for all R.x ∈ r_Map.get(R.a).keySet() do
20:         if r_Map.get(R.a).get(R.x).containsValue(false) then
21:             for all id ∈ r_Map.get(R.a).get(R.x).keySet() do
22:                 emit("id","")
23:             end for
24:         end if
25:     end for
26: end for
```

order to provide efficient lookups[5]. Thus, values of $R.a$ are connected with

[5]We focus on small-large joins in this work, the *small* means that the size of the small relation is much smaller (e.g., 1000×) than the large one (e.g., user information outer joins with large collected logs), thus it can be stored in memory in a large system. Actually,

Algorithm 2 DER and DDR algorithm over MapReduce - Steps 3 and 4

```
    // Step 3 of DDR/DER
    MapReduce grouping/sorting

    // Step 4 of DDR/DER
    setup(Context context):
1:  maps = context.getNumberOfMaps()
    // DER:
    // r_Map: id → {R.a, R.x}
2:  r_Map = load_R_from_HDFS()

    reduce(String key, Iterator values):
    // key: non-matched tuple (DDR) or id (DER)
    // values: list of empty values
3:  count = 0
    // count non-matched tuples (DDR) or ids (DER)
4:  for all v ∈ values do
5:      count += 1
6:  end for
7:  if count == maps then
    // DDR:
    // output non-matched tuple as is
8:      emit("key","")
    // DER:
    // translate id using r_Map
9:      emit("<r_Map.get(key).a,r_Map.get(key).x,null","")
10: end if
```

values of $R.x$ ($R.a \rightarrow R.x$), while for DER we also need to connect values of R.x with values of id ($R.x \rightarrow id$). Finally, the last field, namely *matched*, is initiated as *false* since at the beginning of the Map, none of R tuples have matched tuples of S.

During Map (*step 2*), at each node, we check whether the predicate of input tuple (*value*) is S, and whether $S.b$ (*value.b*) matches any existing $R.a$ in *r_Map* (see Algorithm 1, lines 2-3). For each matched $R.a$ (or $S.b$) in *r_Map* we need to go through all corresponding $R.x$ in *r_Map* (see Algorithm 1, lines 4-10), and (1) output matching results, to HDFS, directly as final output (see Algorithm 1, line 5) and (2) mark matched tuples of R by setting the last field of *r_Map*, namely *matched*, as *true*. Note that we provide the implementation of marking the field *matched* for both DDR (see Algorithm 1, line 6) and DER (see Algorithm 1, lines 7-9), with DER requiring an additional iteration over each *id*.

This knowledge of matched tuples of R is used during "cleanup(Context

keeping a small relation in memory is a commonly used strategy in current solutions using Hadoop (e.g., the SCOPE system in Microsoft [9] and the approaches presented in [18]).

context)" of Map in order to pass non-matched tuples or ids of R to *Reduce*. Thus, for each $R.a$ in *r_Map* (see Algorithm 1, lines 13-26) we provide the implementation for both DER and DDR. For DDR (see Algorithm 1, lines 14-18), we need to check whether there is a $R.x \rightarrow matched$ in *r_Map* such that *matched* equals to *false* (because the join is performed on $R.a$, if at least one $R.x \rightarrow matched$ is false then all are). If *matched* is *false* then we output the non-matched tuple itself (see Algorithm 1, line 16). For DER (see Algorithm 1, lines 19-25), an additional iteration through $R.x$ is required. Then we need to check whether there is an $id \rightarrow matched$ in *r_Map* such that *matched* equals to *false*. However, as opposed to DDR, for DER we output only the *id* of the non-matched tuple (see Algorithm 1, line 22). Note that in both cases, namely DER and DDR, we output an empty value ("") in order to minimize the communication overhead, while each empty value corresponds to one occurrence of the given non-matched tuple or id.

The MapReduce framework will perform grouping/sorting (*step 3*), resulting in groups where the non-matched tuple or id is the unique key followed by the corresponding list of empty values (see Algorithm 2). Then (*step 4*), prior to processing each group, in "setup(Context *context*)" of *Reduce* (see Algorithm 2, lines 1-2), we need to get the number of executed Map operations during the current job (see Algorithm 2, line 1), for both DER and DDR, as it will be the threshold that determines whether the given non-matched tuple or id will be considered as final output. In addition, for DER we also need to load relation R from HDFS (see Algorithm 2, line 2). Note that relation R is kept in-memory (*r_Map*) only for DER, which is modeled as a map from *id* to the corresponding pair of $R.a$, $R.x$, in order to be able to translate each *id* to its corresponding tuple.

During *Reduce* (see Algorithm 2, lines 3-10), at each node, we need to count the occurrences of each non-matched tuple or id of R, namely the number of its corresponding empty values (see Algorithm 2, lines 3-6). In case the number of empty values is equal to the number of executed Map operations (see Algorithm 2, line 7), we have encountered a final output. For DDR, we output the non-matched tuple itself (see Algorithm 2, line 8), which is the *key*. For DER, generating the final output is more complex as we need to translate the *id* (see Algorithm 2, line 9), which is the *key*, using *r_Map*, to each field of relation R, namely $R.a$ and $R.x$.

4.2.3. Discussion

It is clear from Algorithms 1 and 2, that DER and DDR follow a similar algorithmic structure. However, DDR is easier to implement compared to DER. Indeed, in "setup(Context *context*)" of Map the in-memory map of relation R (r_Map) has a simpler structure for DDR, thus leading to easier parsing of relation R, while DER requires also the storage of corresponding ids. This simpler structure of r_Map also results in an easier way of recording which tuples of R match with tuples of S, during Map, and an easier way of emitting non-matched tuples to *Reduce* during "cleanup(Context *context*)" of Map.

In addition, in "setup(Context *context*)" of *Reduce*, DDR only needs to get the number of executed Map operations. On the other hand, DER also needs to load in-memory the relation R. Note that the map of relation R in "setup(Context *context*)" of *Reduce* has a different structure compared to the one loaded in "setup(Context *context*)" of Map, and thus, a different parsing method needs to be implemented. In order to emit a non-matched tuple during *Reduce*, for DDR we can emit directly the *key* itself, which is the non-matched tuple. However, for DER we first need to translate the given *id* into the corresponding tuple.

It is evident that DDR provides a simpler implementation compared to DER for a cloud computing environment. However, for both DER and DDR a naive approach would require **two** jobs, as follows. In the first job, during Map, tuples of S are redistributed over k Reducers, with *step* 1 performed in "setup(Context *context*)" of *Reduce* and *step* 2 performed during *Reduce*, emitting both matched and non-matched results. In the second job, during Map, both matched and non-matched results are passed to *Reduce*, followed by MapReduce framework grouping/sorting (*step* 3), while *step* 4 is performed during *Reduce*, emitting both matched and final non-matched tuples. Such a naive implementation introduces the following overheads compared to our algorithm: (1) the redistribution of tuples of S during the first job, (2) emitting matched results during the first job and subsequently processing them during the second job, and (3) processing non-matched tuples during Map of the second job.

5. Experimental Setup

We have conducted a rigorous quantitative evaluation of the proposed approach based on the setup as follows.

5.1. Platform

Our evaluation platform is the HRSK-II system of ZIH at TU Dresden. Each node we used has two 12-core Intel Xeon CPU E2680 processors running at 2.50 GHz, resulting in a total of 24 cores per physical node. Each node has 64GB of RAM and a single 128GB SSD local disk, and nodes are connected by Infiniband. The operating system is Linux kernel version 2.6.32-279 and the software stack consists of Hadoop version 1.2.1, Spark version 1.2.1, Scala version 2.10.4 and Java version 1.7.0_25.

5.2. Datasets

We have used the TPC-H benchmark [23] in our tests. There are two advantages for this choice: (1) the benchmark has been widely used in evaluating database queries with joins, and (2) there is no skew in the generated data sets, which let us to be able to focus on the performance of join implementation, rather than other issues (e.g., skew handling). We use the following query in our experiments:

```
select *
from CUSTOMER left outer join SUPPLIER
on C.NATIONKEY = S.NATIONKEY
```

In this case, the generated tuples can be then considered as in the form <key, payload> pairs. There are only 25 unique Nationkey values in TPC-H, similar to other work [4], [24], we increase the number of the unique Nationkey values to 10000. This guarantees that the generated tuples can be more evenly distributed during the hash redistribution process. In the meantime, the number of output results can be also efficiently reduced[6].

For simplification, we refer to the relation CUSTOMER as R and the relation SUPPLIER as S in the following. We generate different datasets by varying the scale factor of TPC-H to generate datasets. As the inner join cardinality (referred to as selectivity) is critical for outer joins (i.e., making outer join and inner join implementations different), in our experiments, we vary the values of selectivity for R and S from 0% to 100% (20% increment in each step) by controlling the values on their join keys, while keeping the sizes of R and S constant. For example, the selectivity with value 60% means

[6]For example, with only 25 unique keys, the join between 10K and 1B tuples will bring in 400B tuples for a full match condition. In comparison, with 10K keys, the number will be reduced to 1B (approximately 160GB).

that we randomly choose 60% of tuples of R and keep their keys unchanged, while the rest of the tuples of R have their keys changed to their negative values, so that they will have no matches in S.

5.3. Setup

We have chosen 17 nodes (408 cores) from the cluster for our experiments. We chose one node as the master and the rest as the slavers (workers). We implemented our tests over both the MapReduce and Spark framework. We set the following parameters for the MapReduce implementations: *map.tasks.maximum* and *reduce.tasks.maximum* to 24, the *mapred.child.java.opts* to 1GB, while the remaining parameters were set to their default values. For Spark, we set the following system parameters: *spark_worker_memory* and *spark_executor_memory* were set to 60GB, and *spark_worker_cores* was set to 24. As the computational infrastructure is using a shared network and non-virtualised computational resources, we believe that this setup approximates commercial offerings. In particular, it very closely approximates the *Bare Metal* servers in IBM Softlayer [25] and the *Cluster* instances in Amazon EC2 [26].

For all of our experiments, we read input files and write the output on HDFS. We configure HDFS to use the SSD on each node and use a 64MB block size. For fault tolerance, each HDFS block is replicated three times. We measure runtime as the elapsed time from job submission to the job being reported as finished and we record the mean value based on three measurements. Additionally, to focus on the runtime performance of each outer join implementation, we only record the number of the final outputs, rather than materialising the output (due to large output volumes in our cases).

6. Evaluation

In this section, we present the experimental evaluation of the proposed DDR approach and compare its performance with the approaches described in Section 2, namely the ROJA, DOJA and DER algorithm. We are interested in how efficient the four approaches are, and how the join workloads impact their performance, in the presence of small-large conditions. Therefore, we focus on the four possible impact factors in our evaluation: the number of tuples in S and R as well as the size of payloads in S and R. We

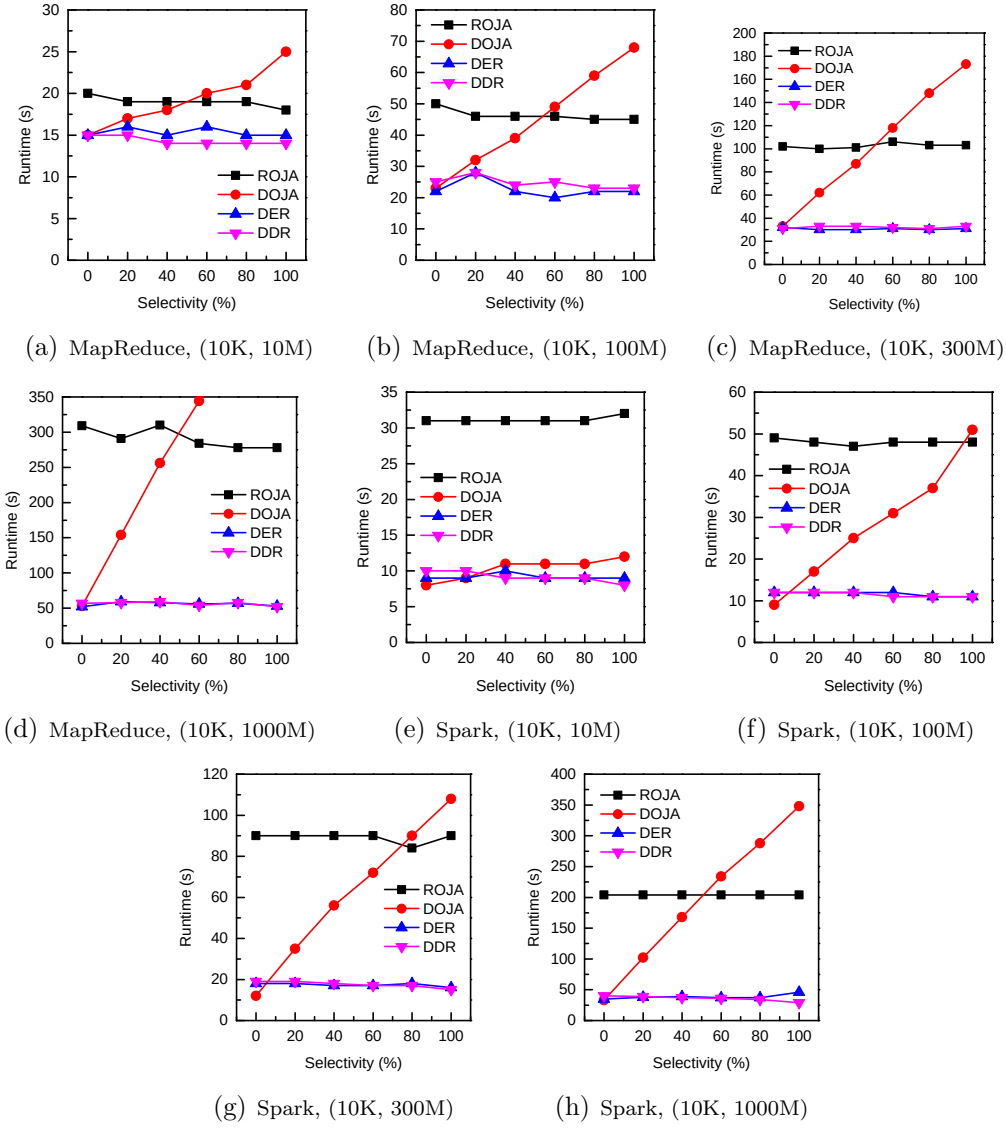
present the detailed results in the following. Moreover, we also evaluate the scalability of each approaches by varying the number of nodes.

6.1. Impact of $|S|$

The size of tuples in S could affect the required time for tuple redistribution and local joins. To see how performance changes for increasing numbers of tuples in S, we fix the number of tuples of R to 10K and vary the size of S by 10M (1.5GB), 100M, 300M and 1000M tuples (160GB). The detailed results over MapReduce and Spark are presented in Figure 3, where the two numbers in the caption of each subfigure correspond to the number of tuples of R and S.

For the MapReduce and Spark implementations, we have a clear picture of the results. The ROJA, DER and DDR algorithms generally have steady performance for varying join selectivity (recall that we do not include the final output, which is the same for all four methods). In comparison, the DOJA algorithm is highly affected by the join hit rate, mainly due to the intermediate materialization of matching R and S joins (see the *Stage 1* of DOJA in Section 2). In the meantime, it can be observed that the DER and DDR algorithms generally outperform the ROJA and DOJA algorithms for different $|S|$. Moreover, though the runtime of all four algorithms is increasing with increasing sizes of S, the cost difference between the two conventional approaches and DER as well our proposed DDR is becoming more clear, demonstrating the efficiency of the latter two approaches on processing small-large outer joins. For a general case with selectivity 60%, Figure 4 shows a more intuitive view of these changes. We can see that the runtime of ROJA and DOJA increases sharply with increasing sizes of $|S|$, while there is only slight increase for DER and DDR, due to the increment of the local join workloads.

Comparing the runtimes of the four algorithms over the MapReduce and Spark platform, it can be seen that the implementations of DOJA, DER and DDR over the latter framework are generally faster than the former one. The possible reason for this could be that these three algorithms result in relatively complex implementations, and Spark processes the workflows in a more efficient way, while having the advantage of in-memory storage. In comparison, for the simplest implementation - ROJA, when $|S|$ is small, MapReduce is initially faster than Spark and then becomes slower for increasing sizes of $|S|$. The possible reasons could be that: (1) the overheads of implementation of Spark is greater than our MapReduce implementation at the beginning,



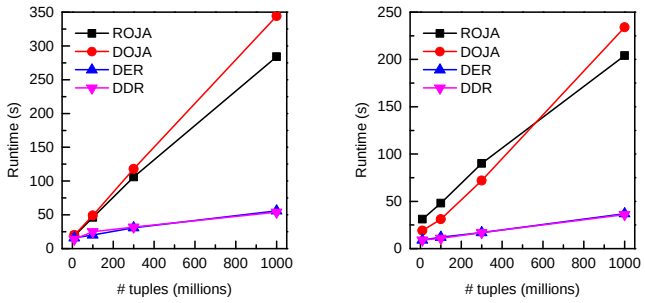
(a) MapReduce, (10K, 10M) (b) MapReduce, (10K, 100M) (c) MapReduce, (10K, 300M)

(d) MapReduce, (10K, 1000M) (e) Spark, (10K, 10M) (f) Spark, (10K, 100M)

(g) Spark, (10K, 300M) (h) Spark, (10K, 1000M)

Figure 3: The runtime of the four algorithms over MapReduce and Spark by varying the inner join selectivity and the number of tuples in S.

and (2) when $|S|$ is huge, the data management (transferring and in-memory storage) of Spark is more efficient than MapReduce.



(a) MapReduce (b) Spark

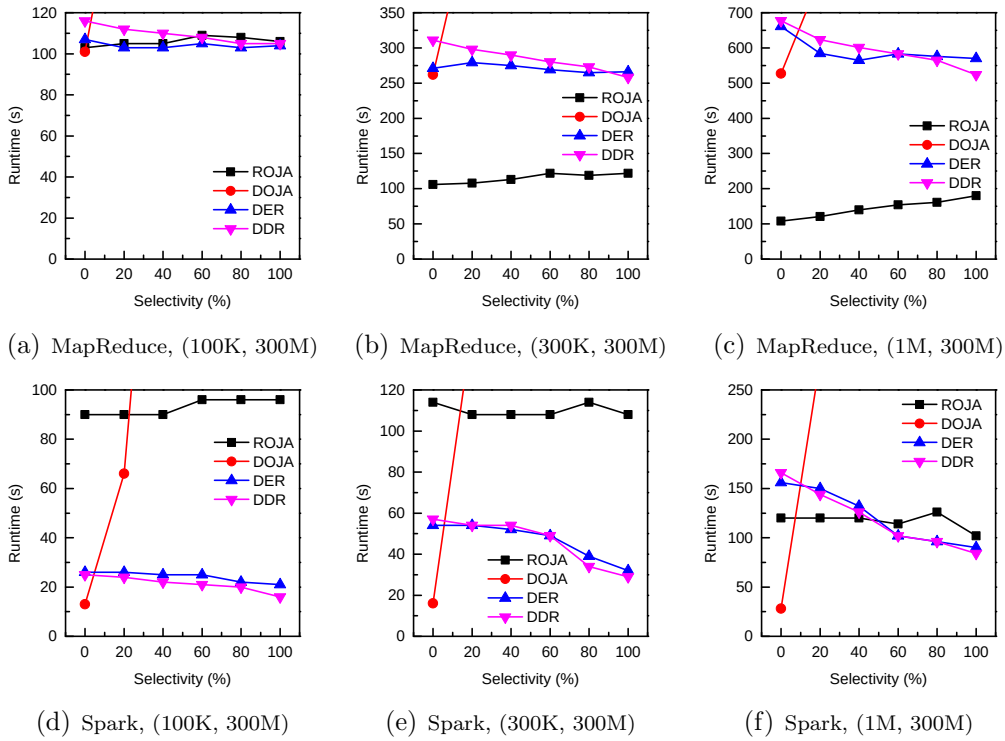
Figure 4: The trends of runtime of different algorithms by varying the size of $|S|$. Here, $|R|$ = 10K and selectivity = 60%.

6.2. Impact of $|R|$

As the number of $|R|$ would affect both the network communication (due to data redistribution and duplication) and local join workloads, we also examine how this factor impacts the runtime. In the scope of a small-large case, we keep the number of tuples in S to 300M (46GB) and vary the number of tuples in R by 10K, 100K, 300K and 1M (136MB). The detailed results of the implementations over MapReduce and Spark are shown in Figure 5. Results for $|R|$ equal to 10K have been presented in Figure 3(c) and 3(g) respectively.

It can be observed that runtimes for the ROJA algorithm remain relatively stable for varying join selectivity, while runtimes for DOJA increase sharply. Moreover, for the conditions with non-zero selectivity, DOJA cannot cope with the experiments when $|R|$ is greater than 10K. The reason for this could be that the number of intermediate matched results becomes very large and redistributing them is extremely expensive. This also means that DOJA is generally not well-suited for small-large outer joins. In the meantime, we can see that the DER and DDR algorithms show similar performance in all four cases and their runtime generally decreases with the increase of the selectivity. The reason for this could be that the number of non-matched tuples decreases, which results in less data transferring. In the meantime, we notice that the decrease of the runtime over Spark is more clear than that over MapReduce. The reason could be that the performance over Spark is more sensitive to the network communication.

When $|R|$ is very small (i.e., 10K), DER and DDR perform much faster



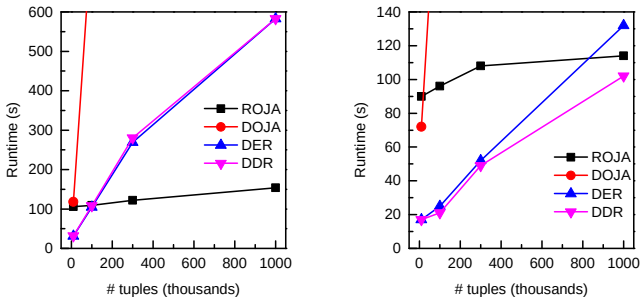
(a) MapReduce, (100K, 300M) (b) MapReduce, (300K, 300M) (c) MapReduce, (1M, 300M)

(d) Spark, (100K, 300M) (e) Spark, (300K, 300M) (f) Spark, (1M, 300M)

Figure 5: The runtime of different algorithms by varying the size of $|R|$.

than ROJA over both MapReduce and Spark platforms. However, their performance changes as $|R|$ is growing. For the MapReduce-based implementations, the runtime of DER and DDR start to perform slower than the ROJA algorithm when $|R|$ is greater than 100K. Meanwhile, when $|R|$ reaches 1M, the DER and DDR algorithms begin to perform similarly to ROJA, over the Spark platform. For the general case with selectivity 60%, Figure 6 shows the runtime trends of each algorithm with increasing $|R|$. It can be observed that the duplication-based approaches are very sensitive to the size of R and increase sharply with the increase of $|R|$. There are three possible reasons for this: (1) the broadcast is becoming more expensive with increasing $|R|$; (2) the increased number of non-matched results (ids) also brings in communication overheads; and (3) local joins between the broadcast $|R|$ and locally kept S become more expensive. In this process, DDR always performs similar to DER, which means that redistributing non-matched results or ids does not have a clear impact on small-large outer joins, in terms of performance.

Moreover, w.r.t. performance advantage compared to the ROJA, it can be noticed that the DER and DDR algorithm shows more tolerance on the size of relation R over the Spark platform than MapReduce. The reason could be the existing I/O overheads for the MapReduce-based implementations. It can be seen that DER and DDR could lose their performance advantages over ROJA, when $|R|$ reaches 1M, which is still a much smaller size compared to the relation S (300M tuples). This means that in real applications we should examine or detect the threshold of the small relation very carefully in small-large outer joins, though in many cases the relation may appear very small.



(a) MapReduce (b) Spark

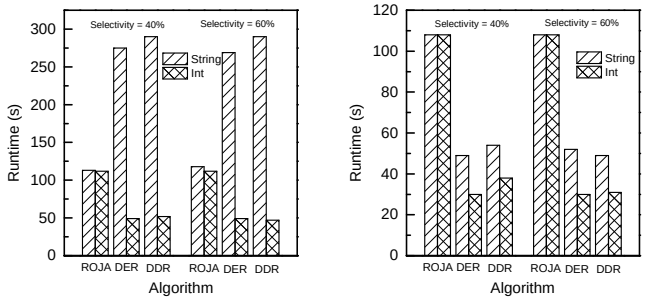
Figure 6: The trends of runtime of different algorithm by varying the size of $|R|$. Here, $|S| = 300$M and selectivity = 60%.

6.3. Impact of payload size in R

In the TPC-H datasets, the payloads for all tuples are always long strings, while transferring such strings could bring heavy network communication as well as local computation overheads (because of the extra serialized/deserialized processing). Though large datasets in real applications could contains large number of long strings, to check the detailed effects of the size of payload in R, in our tests, we simply change the payload in a tuple from a string to an integer.

Figure 7 presents the results for the join between 300K and 300M tuples by changing the payloads in R. In this case, the size of R in MB is changed from 41MB to 4MB. For a general case, we just show the runtimes where selectivity is 40% and 60% (as the DOJA algorithm is costly we only present the remaining three algorithms). It can be seen that the runtime of ROJA

remains generally the same with varying the payload size over both MapReduce and Spark platforms. This is reasonable, since reducing the payload size of the small relation has no clear impact on the time spent on redistribution and the local join implementations. In comparison to that, the DER and DDR algorithms show a clear decrease on runtime with reducing the payload size. The reason for this could be that the cost of broadcast is highly reduced. In the meantime, the I/O costs are also highly reduced for the MapReduce-based implementations, which brings in more clear performance improvements, compared to that over the Spark platform. This is partly attributed to the fact that non-matched tuples between map and reduce phase are temporarily stored in HDFS (for both DER and DDR), while for DER the broadcast data is read by reducers (see Algorithm 2, line 2). All these results together with the ones presented in previous subsection, indicate that the size of the small relation in MB is also critical for the success of the DER and DDR algorithms, for the case of small-large outer joins.



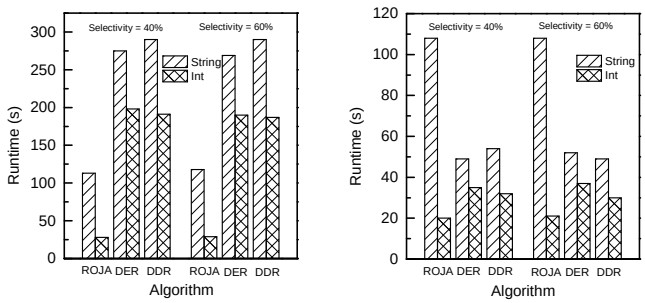
(a) MapReduce (b) Spark

Figure 7: Runtime of each algorithm by changing the size of payloads in R. The outer join is between 300K and 300M tuples.

6.4. Impact of payload size in S

Similar as above, we also examine how the payload size of tuples in S impacts the join performance by changing the payloads from strings to integers. In this condition, the size of S is changed from 46GB to 4GB. Figure 8 shows the results of the three algorithms (ROJA, DER and DDR) for the join between 300K and 300M. The reduction of the size of S brings clear runtime decrease for the three algorithms over both MapReduce and Spark framework. Regardless, it can be observed that runtime of ROJA decreases

more sharply than runtime of DER and DDR, which makes it clearly faster than the two algorithms when the payloads are integers, though the outer joins can still be considered as a small-large join, in terms of both number of tuples and size in MB (41MB join with 4GB). This indicates that it is better to use the DER and DDR algorithm under the condition that the large relation is large enough, in terms of size in MB.



(a) MapReduce (b) Spark

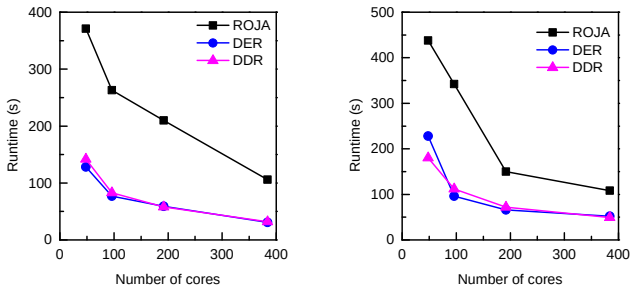
Figure 8: Runtime of each algorithm by changing the size of payloads in S. The outer join is between 300K and 300M tuples.

6.5. Scalability

We test the scalability (scale-out) of our approach by varying the number of slaves (workers), from 48 cores (2 nodes) to 384 cores (16 nodes). For a general case, we keep the selectivity to 60% and the number of tuples in S to 300M (46GB). As we have shown in our previous results, MapReduce and Spark have different tolerance on the concept of *small* in the scope of small-large outer joins. According to the results demonstrated in Figure 6, to make our evaluation here meaningful, i.e., in a case that DER and DDR perform faster than conventional approaches, we keep the number of tuples in R to 10K for MapReduce and 300K for Spark. The detailed results of our tests are presented in Figure 9. We can see that the runtimes of all three algorithms decrease with increasing numbers of cores, which means that they generally scale well over both the MapReduce and Spark frameworks.

Moreover, we can see that the benefit of adding more cores (i.e., the scaled speedup) decreases as the runtime becomes lower for all the cases. We attribute this to platform overhead since the workload is comparably small for the underlying platform as the number of cores increases. Comparing

the detailed performance of DER and DDR, it can be observed that both algorithms perform nearly the same for both MapReduce and Spark. More specifically, when the number of cores is small, DDR performs slightly slower than DER over MapReduce, while slightly faster than DER over Spark. The possible reasons are: (1) DDR transfers more than DER (transferring tuples instead of ids) and thus its I/O overhead in MapReduce could make it slightly slower; and (2) in contrast, there is no such overhead in Spark. In addition, DDR is simpler than DER in terms of system implementations, allowing for an easier and quicker development process. Regardless, with the underlying system going to large-scale, such performance differences become negligible, which is consistent with our previous theoretical analysis in Section 3.2.2.



(a) MapReduce, (10K, 300M) (b) Spark (300K, 300M)

Figure 9: Runtime of each algorithm over MapReduce and Spark by varying the number of cores, with selectivity = 60%.

6.6. Discussion

In terms of general implementations, the proposed DDR approach has the overhead of storing and sending the whole non-matched tuples, however it does not need any further operations on data transformation. In comparison, the DER method has the advantage of less data transmission due to the ids, but it also has an overhead of transforming ids. From the results presented above, we can see that DER and DDR scale well, and always achieve very similar performance under various workloads, over both MapReduce and Spark platforms, which is consistent with our theoretical analysis presented in Section 3.2.2. All these mean that we can implement the proposed DDR algorithm for small-large outer joins in a cloud based environment expecting similar performance as the state-of-the-art DER algorithm. Note that DDR is easier to implement, as we do not need to handle the encoding/decoding of

ids for non-matched results, though these operations would be lightweight as we explained. In such scenarios, the DDR algorithm could be more preferable for cloud-based architectures form an engineering point view.

Moreover, for an ideal small-large outer joins (e.g., the small relation is small enough and the large one is large enough), our experimental results have demonstrated that the DER and DDR algorithms can clearly outperform the two conventional approaches, which confirms that the presented DER and DDR are the state-of-the-art approaches for small-large outer joins in a cloud computing environment. Though in some cases, we see that ROJA performs faster than or similar to the DER and DDR approaches, the main reason is that the small relation is not sufficiently small. On the other hand, it should be noted that ROJA could meet performance issues in the presence of data skew due to the load imbalance problem. In comparison, DER and DDR are duplication-based approaches, thus they are more suitable for processing skewed datasets. In fact, the optimizer of a given system could pick the correct implementation based on the input so as to minimize runtime. Additionally, it can be seen that DER and DDR have slightly different behaviors on the Hadoop and Spark platforms. This is mainly attributed to the fact that Spark runs everything in memory, while Hadoop relies heavily on HDFS, resulting in extra I/O overheads.

7. Related Work

The studies in parallel joins on shared memory systems [1] and GPUs [27] have already achieved significant performance speedups, through improvements in architecture at the hardware-level of modern processors. Nevertheless, with the growing challenges from big data, the performance of their join executions is limited by either the number of available threads, or the system memory and I/O. Therefore, efficient implementations of parallel joins on distributed memory machines such as in a cloud computing environment are becoming more and more attractive.

To efficiently implement joins in a distributed architecture, various techniques such as dynamic scheduling [28] and statistics [29], have been proposed. Regardless, their implementations are still based on the two most conventional join patterns: redistribution and duplication. Specifically, as the latter operation is very costly in a distributed environment, normally it is only applied to process small relations. Though some existing approaches use duplication on large datasets [30], their implementations rely highly on the

emerging computing infrastructures (e.g., underlying high-speed networks), which are normally unavailable in cloud computing environments.

Current research on outer joins focuses on optimization of existing methods, which mainly include outer join elimination [31], outer join reordering [6] and view matching for outer join views [32]. There is little work done on outer join implementations. The reason for this may be the assumption that inner join techniques can be easily applied to outer joins [10]. However, as we have described, outer joins have their unique characteristics, especially in the condition that the implementation is using duplication.

Recently, several approaches have been designed on outer join implementations [8], [9], [33], [34] and the experiments have shown that they are very efficient in this aspect. However, all proposed methods focus on data skew handling (either single or pipeline joins) in large-large joins, but not improving the performance for the small-large outer joins. As we have described, the DER algorithm [10] is the state-of-the-art approach in such cases. Regardless, our DDR method is shown to be easier to implement and has no usage constrains. More importantly, as demonstrated in our experiments, DDR always achieves similar performance as DER under different workloads in a cloud computing environment.

Regarding joins in a cloud computing environment, most of current works focus on proposing novel data skew handling techniques to improve the load-balancing and scalability of join implementations in the presence of big data (e.g., SALA [35], SkewTune [36] and the approaches presented in [37], [38]), as opposed to the detailed implementation and evaluation of joins that is studied in this work. Moreover, the large scale data-analytics community has developed its own set of parallel processing paradigms and related join operations. For example, popular platforms such as Spark [13], provide their APIs for different join implementations. In the meantime, several efforts in designing high level query languages on MapReduce, such as Pig [39] and Hive [40], also provide high level join operations. In addition to that, to support ad-hoc data access, various key-value stores such as HBase [41] and Cassandra [42] provide related functions as well. However, their outer join implementations are still based on the two conventional approaches, namely ROJA and DOJA, without any specified optimizations for the small-large case, which has been studied in this work. Furthermore, though the work in [18] presents an extensive implementation of joins in MapReduce, it focuses on execution profiling and performance evaluation of inner joins, but not of outer joins. The recent work [11, 19] has implemented efficient outer joins

in cloud computing environments, still, it just focuses on skew handling in large-large joins.

To the best of our knowledge, this is the first work specified for detailed design and evaluation of *small-large* outer joins in *cloud* computing environments, and we have shown that the proposed DDR approach is efficient and can be easily applied in such environments. Moreover, we believe that the evaluation conducted in this work and the described results are of value to the community as a basis for understanding the merits of the approach. In the meantime, based on the advantages of our new implementation, we expect that our approach will also be efficient and easy to implement in more complex enviroments such as heterogeneous environments [43], which require more efficient strategies on resource management [28], [44].

8. Conclusions

In this paper, we have introduced a new outer join algorithm called DDR (duplication and direct redistribution), which is specified for efficient processing of small-large outer joins in cloud computing environments. We have presented the detailed implementation of our algorithm and conducted an extensive evaluation of this method over the MapReduce and Spark platforms. Compared to the state-of-art DER approach [10], we have shown that DDR: (1) is easier to implement; (2) scales well and achieves similar performance to DER; and (3) has no usage constrains in real commercial cases. In such scenarios, our new approach can be considered as a new option for cloud-based large-scale data analysis applications. Moreover, the detailed experimental results characterizing current implementations will also contribute to the engineering in this domain.

Acknowledgments

Part of the work was done when Long Cheng worked at TU Dresden, and supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed), the Collaborative Research Center SFB 912 (HAEC) and Emmy Noether grant KR 4381/1-1 (DIAMOND).

References

[1] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, P. Dubey, Sort vs. hash revisited: fast join implementation on modern multi-core CPUs, Proc. VLDB Endowment 2 (2) (2009) 1378–1389.

[2] D. DeWitt, J. Gray, Parallel database systems: the future of high performance database systems, Commun. ACM 35 (6) (1992) 85–98.

[3] D. J. DeWitt, J. F. Naughton, D. A. Schneider, S. Seshadri, Practical skew handling in parallel joins, in: Proc. 18th Int. Conf. Very Large Data Bases, 1992, pp. 27–40.

[4] Y. Xu, P. Kostamaa, X. Zhou, L. Chen, Handling data skew in parallel joins in shared-nothing systems, in: Proc. ACM SIGMOD Int. Conf. Manage. Data, 2008, pp. 1043–1052.

[5] L. Cheng, S. Kotoulas, T. E. Ward, G. Theodoropoulos, Robust and skew-resistant parallel joins in shared-nothing systems, in: Proc. 23rd ACM Int. Conf. Inf. Knowl. Manage., 2014, pp. 1399–1408.

[6] C. Galindo-Legaria, A. Rosenthal, Outerjoin simplification and reordering for query optimization, ACM Trans. Database Syst. 22 (1) (1997) 43–74.

[7] M. Atre, Left bit right: For SPARQL join queries with OPTIONAL patterns (left-outer-joins), in: Proc. ACM SIGMOD Int. Conf. Manage. Data, 2015, pp. 1793–1808.

[8] L. Cheng, S. Kotoulas, T. E. Ward, G. Theodoropoulos, Robust and efficient large-large table outer joins on distributed infrastructures, in: Proc. 20th Eur. Conf. Parallel Process., 2014, pp. 258–369.

[9] N. Bruno, Y. Kwon, M.-C. Wu, Advanced join strategies for large-scale distributed computation, Proc. VLDB Endowment 7 (13) (2014) 1484–1495.

[10] Y. Xu, P. Kostamaa, A new algorithm for small-large table outer joins in parallel DBMS, in: Proc. IEEE 26th Int. Conf. Data Eng., 2010, pp. 1018–1024.

[11] L. Cheng, S. Kotoulas, Efficient skew handling for outer joins in a cloud computing environment, IEEE Trans. Cloud Comput. (in press)doi:10.1109/TCC.2015.2487965.

[12] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation, 2012, pp. 15–28.

[14] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, M. Stonebraker, A comparison of approaches to large-scale data analysis, in: Proc. ACM SIGMOD Int. Conf. Manage. Data, 2009, pp. 165–178.

[15] Y. Xu, O. P. Kostamaa, Performing an outer join between a small table and a large table, U.S. Patent No. 8,600,994 (Dec. 3 2013).

[16] S. Kotoulas, E. Oren, F. Van Harmelen, Mind the data skew: distributed inferencing by speeddating in elastic regions, in: Proc. 19th Int. Conf. World Wide Web, 2010, pp. 531–540.

[17] L. Cheng, S. Kotoulas, T. E. Ward, G. Theodoropoulos, Design and evaluation of parallel hashing over large-scale data, in: Proc. 21st Int. Conf. High Perform. Comput., 2014, pp. 1–10.

[18] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, Y. Tian, A comparison of join algorithms for log processing in MapReduce, in: Proc. ACM SIGMOD Int. Conf. Manage. Data, 2010, pp. 975–986.

[19] L. Cheng, S. Kotoulas, Efficient large outer joins over MapReduce, in: Proc. 22nd Eur. Conf. Parallel Process., 2016, pp. 334–346.

[20] D. Borthakur, HDFS architecture guide, Hadoop Apache Project (2008) 53.

[21] C.-Y. Lin, C.-H. Tsai, C.-P. Lee, C.-J. Lin, Large-scale logistic regression and linear support vector machines using Spark, in: Proc. IEEE 3rd Int. Conf. Big Data, 2014, pp. 519–528.

[22] S. Wu, F. Li, S. Mehrotra, B. C. Ooi, Query optimization for massively parallel data processing, in: Proc. 2nd ACM Symposium on Cloud Computing, no. 12, 2011.

[23] T. P. P. Council, TPC-H benchmark specification, Published at http://www.tpc.org/tpch/.

[24] W. Liao, T. Wang, H. Li, D. Yang, Z. Qiu, K. Lei, An adaptive skew insensitive join algorithm for large scale data analytics, in: Proc. 16th Asia-Pacific Web Conf., 2014, pp. 494–502.

[25] `http://www.softlayer.com/`.

[26] `https://aws.amazon.com/ec2/`.

[27] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. Sander, Relational joins on graphics processors, in: Proc. ACM SIGMOD Int. Conf. Manage. Data, 2008, pp. 511–524.

[28] K. Imasaki, S. P. Dandamudi, An adaptive hash join algorithm on a network of workstations, in: Proc. Int. Parallel and Dist. Process. Symp., 2002.

[29] M. Al Hajj Hassan, M. Bamha, An efficient parallel algorithm for evaluating join queries on heterogeneous distributed systems, in: Proc. Int. Conf. High Perform. Comput., 2009, pp. 350–358.

[30] P. W. Frey, R. Goncalves, M. Kersten, J. Teubner, Spinning relations: high-speed networks for distributed join processing, in: Proc. 5th Int. Workshop Data Manage. on New Hardware, 2009, pp. 27–33.

[31] J. Rao, H. Pirahesh, C. Zuzarte, Canonical abstraction for outerjoin optimization, in: Proc. ACM SIGMOD Int. Conf. Manage. Data, 2004, pp. 671–682.

[32] P.-Å. Larson, J. Zhou, View matching for outer-join views, The VLDB Journal 16 (1) (2007) 29–53.

[33] Y. Xu, P. Kostamaa, Efficient outer join data skew handling in parallel DBMS, Proc. VLDB Endowment 2 (2) (2009) 1390–1396.

[34] L. Cheng, S. Kotoulas, T. Ward, G. Theodoropoulos, Efficient handling skew in outer joins on distributed systems, in: Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput., 2014, pp. 295–304.

[35] Z. Lin, M. Cai, Z. Huang, Y. Lai, SALA: A skew-avoiding and locality-aware algorithm for MapReduce-based join, in: Proc. 16th Int. Conf. Web-Age Information Manage., 2015, pp. 311–323.

[36] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, Skewtune: Mitigating skew in Map-Reduce applications, in: Proc. ACM SIGMOD Int. Conf. Manage. Data, 2012, pp. 25–36.

[37] J. Myung, J. Shim, J. Yeon, S.-g. Lee, Handling data skew in join algorithms using MapReduce, Expert Systems with Applications.

[38] M. A. H. Hassan, M. Bamha, F. Loulergue, Handling data-skew effects in join operations using Map-Reduce, Procedia Computer Science 29 (2014) 145–158.

[39] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, U. Srivastava, Building a high-level dataflow system on top of Map-Reduce: the Pig experience, Proc. VLDB Endowment 2 (2) (2009) 1414–1425.

[40] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive: a warehousing solution over a Map-Reduce framework, Proc. VLDB Endowment 2 (2) (2009) 1626–1629.

[41] L. George, HBase: the definitive guide, " O'Reilly Media, Inc.", 2011.

[42] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44 (2) (2010) 35–40.

[43] Q. Chen, J. Yao, Z. Xiao, Libra: Lightweight data skew mitigation in Map-Reduce, IEEE Trans. Parallel and Dist. Syst. 26 (9) (2015) 2520–2533.

[44] X. Zhang, T. Kurc, T. Pan, U. Catalyurek, S. Narayanan, P. Wyckoff, J. Saltz, Strategies for using additional resources in parallel hash-based join algorithms, in: Proc. 13th IEEE Int. Symp. High Perform. Dist. Compt., 2004, pp. 4–13.