Hindawi
Scientific Programming
Volume 2017, Article ID 3072813, 9 pages
https://doi.org/10.1155/2017/3072813

Research Article

Cross-Checking Multiple Data Sources Using Multiway Join in MapReduce

Foto Afrati, Zaid Momani, and Nikos Stasinopoulos

National Technical University of Athens, Athens, Greece

Correspondence should be addressed to Zaid Momani; zed_momany@yahoo.com

Received 27 May 2017; Revised 30 August 2017; Accepted 27 September 2017; Published 20 November 2017

Academic Editor: Marco Aldinucci

As data sources accumulate information and data size escalates it becomes more and more difficult to maintain the correctness and validity of these datasets. Therefore, tools must emerge to facilitate this daunting task. Fact checking usually involves a large number of data sources that talk about the same thing but we are not sure which holds the correct information or which has any information at all about the query we care for. A join among all or some data sources can guide us through a fact-checking process. However, when we want to perform this join on a distributed computational environment such as MapReduce, it is not obvious how to distribute efficiently the records in the data sources to the reduce tasks in order to join any subset of them in a single MapReduce job. To this end, we propose an efficient approach using the multiway join to cross-check these data sources in a single round.

1. Introduction

In many applications, we need to extract meaningful conclusions from multiple data sources that provide information about seemingly the same attributes (although the attributes may not have the same name, we have good reason to believe that they refer to the same entity/info). Such different data sources provide pieces of information that may be slightly different, contradicting each other, or even incomplete. For example, in the latter case, the missing data problem is not at all uncommon as there exist abundant examples in the social sciences field [1] and clinical research [2, 3].

Incomplete and missing data problems in medical field are reported [4–6]. These problems have been studied in areas such as knowledge discovery, web personalization, and fact checking [7–10]. In order to make sense of the data, we must address problems such as the missing or inconsistent data problems while at the same time coping with the sheer amount of data presented to us. The typical approaches to dealing with such problems involve employing statistical methods to attack the problem beginning with seminal work such as [11] moving forward to data mining [9] and data integration techniques [12]. We address the missing data problem indirectly; rather than using typical missing data imputation methods that rely on statistical and probabilistic methods [6] to infer the missing value, we opt to use a basic form of majority vote [11, 12] that results in replacing the missing value with the common value among the other data sources. In our approach we allow for missing values to exist in some of the tuples attributes and identify similarity between records with respect to their agreement in nonmissing attributes. Finally, after we join on similar values we can use the result to populate missing attributes in the original sources. The result would be the most common value across all data sources. Inconsistency is resolved in the same manner, consisting of the most common value across data sources.

The view that is presented in this paper gives priority to incorporating as many data sources as possible to our data pool. Since often not a single data source offers good information about all the potential queries, it would be valuable to combine and compare the information the sources are offering and find answers to our queries that are based on such a combination of many data sources that may be partially reliable. It is not uncommon that some queries are answered reliably from certain data sources and some others from a different set of data sources. Also, in contrast to our previous work [13], we expose the advantage of the nature of such data sources (i.e., the presence of overlap between entities across different sources) by providing new experiments with different degrees of overlap. We also provide a more thorough description of problem and contribution. Moreover, we exploit our preestablished approach by introducing a preprocessing step which detects similarities between entities to treat inconsistencies caused by typing errors.

An example follows for a problem that can be viewed as part of an Entity Resolution problem (it is somewhat contrived for the sake of simplicity).

Example 1. Suppose we have four data sources that offer information about movies and actors participating in them and demographic information about the actor, such as telephone numbers and addresses. In many modern applications (e.g., Dremel [14, 15]) all this information is collected in a single relation with many attributes rather than in many relations (as would be the case, e.g., in a star schema with a big fact table and several smaller dimension tables). It might be the case that we have many data sources that offer similar information which, therefore, are over almost the same attributes but not quite. In our actors example, let us assume that we have four relations, R_1, R_2, R_3, and R_4 where R_1 is over attributes *movieTitle*, *actorName*, *telNumber*, and *producer*, R_2 is over attributes *movie-title*, *actor*, *address*, and *tel*, R_3 is over attributes *actorName*, *address*, *producerCollaborating*, and *telNumber*, and R_4 is over attributes *movieTitle*, *producer*, *producerTelNumber*, and *prodAddress* (remember some actors may also be producers). Since the sources are incomplete and not reliable, if we want the telephone number of Tom Cruise, we would inquire all the available sources and either find a single number or find the number that appears more often.

In this paper, we offer a solution to the problem demonstrated in the example above that can be implemented in the MapReduce [16] computational environment. In particular, we show that we can distribute many relations with large overlaps over their attributes in an efficient way across the reducers of a single MapReduce job by using the multiway join algorithm of [17]. We distribute our input relation records as if we intended to join all relations. However, having efficiently distributed our relation tuples, we may choose to join in the reducers any subset of the relations we find suitable for our particular query. We refer again to the example above to show how we may benefit from that.

Example 2. Suppose we want to find the telephone number of Tom Cruise. Then in each reducer we join all relations that contain telephone numbers (observe that relation R_4 may also contain a reference to Tom Cruise if he had been a producer) and see what we find.

```
SELECT tel. FROM R1 JOIN R2 JOIN R3 JOIN R4

WHERE actor_name=Tom Cruise OR prod_name

=Tom Cruise
```

If the output is not satisfying (e.g., empty, which means that some tel. numbers are wrong or there are multiple *tel.* numbers and therefore they did not join), we can join fewer relations in the reducers and come up with answers.

We may use the same distribution, of course, to resolve a different query; for example, find the address of Angelina Jolie. Thus we have used the same distribution to the reducers for multiple queries as well as to resolve queries by choosing which relations to join (without the extra overhead of redistributing the relations).

All of the above is feasible because of a very important property that the multiway algorithm of [17] has and which we will explain in the following section.

The rest of the paper is organized as follows. Section 2 refers to the related work. In Section 3, we review the multiway algorithm described in [17]. In Section 4, we examine closely the communication cost of the multiway join in the case of relations joining over a multitude of shared attributes and how this compares to binary multiway joins, where joins between two relations happen over one attribute, thus arguing that in some cases the multiway algorithm is expected to have exceptionally good performance (which we exploit in this paper) whereas there may be cases where this is not so. In Section 5, we provide representative experimental results. In Section 6 we offer a direct application of the technique presented in this paper. Finally Section 7 provides conclusions and briefly discusses future work and meaningful applications of this work.

2. Related Work

In [18], different types of joins (equi-joins, thetajoins, similarity, k-NN, and Top-k) are listed and compared by the number of MapReduce rounds they require, whether they provide exact or approximate solutions and the number of relations they involve, that is, binary or multiway. According to [18], existing approaches to computing similarity joins on MapReduce are (a) restricted to binary joins [19–24] and (b) most require multiple MapReduce rounds [19–23].

In this paper, we pick [24] as a single round algorithm for computing similarity joins and using [17] we extend to apply for the multiway join. Furthermore, we show that in specific use cases the multiway join of [17] efficiently distributes the data to the reducers to avoid large replication rate (see Section 4.2).

Apart from the work closely related to the technical part of this paper, there is a large amount of work that reflects on the same problem of fact checking, such as [12, 25]. Data integration of conflicting data sources has been studied in [25] and attempts to find true values among conflicting values across data sources. In [12] the notion of dependence was introduced being the case where some data sources copy from others. Dependence and the accuracy of data sources are assessed by different probability models and are taken into consideration when cross-checking conflicting data sources.

Furthermore, work in the context of Entity Resolution (ER) and Record Linkage (RL) [26, 27], also known by other names such as object matching, deduplication, or reference reconciliation, is somewhat related to our goal in this work. ER and RL are important steps in data cleansing resolving inaccuracies. The only difference is that ER and RL look for similar entities to be considered as potential duplicates of real world entities while our typo detection method considers similar entities as potential data entry mistakes or errors.

3. Multiway Join Algorithm for MapReduce

MapReduce is a parallel computation framework for processing over large amounts of distributed data. The user provides an implementation of the *map* function which transforms its input to `<key, value>` pairs. Those pairs are shuffled along the distributed system and assigned to *reduce tasks*, each one distinctly identified by the `key` value. Each *reduce* task applies a reduce function to the `values` associated with each `key` to produce a final result.

In the multiway join algorithm [17], the map phase provides an efficient way to allocate tuples of relations to reduce tasks which in turn perform the actual join. We demonstrate the algorithm using two examples—a simple one and a more complicated one.

Example 3. Assume we have the following 3-way chain join:

$$A_1(x_1,x_2) \bowtie A_2(x_2,x_3) \bowtie A_3(x_3,x_4). \quad (1)$$

To perform a multiway join, the *map task* assigns tuples coming from different relations to different keys with the goal of merging tuples with matching attribute values at the reduce phase. In this example, at the reduce phase we need to focus on the two join attributes: the x_2 attribute, shared between relations A_1 and A_2, and attribute x_3, shared between relations A_2 and A_3.

In order to perform the multiway join correctly in one round of MapReduce, we must ensure that any three tuples that match on x_2 value and on x_3 value meet at the same reduce task. In our Example 3, the tuples of the middle relation A_2 should match on x_2 with the tuples of A_1, and thus all tuples within A_1 with $A_1 \cdot x_2 = A_2 \cdot x_2$ must be in the same reducer, likewise for $A_2 \cdot x_3 = A_3 \cdot x_3$. To accomplish this, the middle relation acts as a key consisting of (x_2, x_3) to indicate which reducer is to receive the emitted `<key,value>` by the mapper. To restrict the arbitrary size of the key produced by (x_2, x_3) to a fixed size, the actual key is composed of hash values reflecting a set of buckets corresponding to a number of reducers (see Figure 1). Note that reducers/buckets and index of bucket/key of reducer can be used interchangeably to denote the same thing.

Let h be the hash function with range $0 \cdots m-1$. Each bucket is identified by a pair (i, j) where each of i, j ranges between 0 and $m-1$. Hence the number of buckets is $m \times m$. All tuples received from the middle relation A_2 will be hashed to bucket $(h(x_2), h(x_3))$ whereas dangling tuples coming from A_1 and A_3 will be replicated over buckets $(h(x_2), j)$ for all values of j and buckets $(i, h(x3))$ for all values of i, respectively. As mentioned earlier the *mapper* emits the pairs `<key,value>`; the key part is the index of the bucket and the value part is the actual tuple including the name

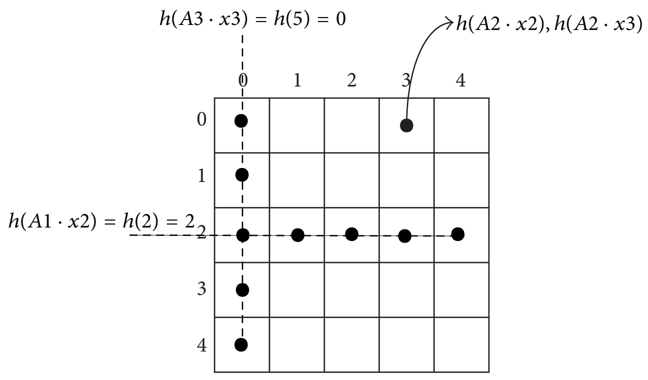

$h(A3 \cdot x3) = h(5) = 0$
$h(A2 \cdot x2), h(A2 \cdot x3)$
0
1
2
3
4
0
1
2
3
4
$h(A1 \cdot x2) = h(2) = 2$

FIGURE 1: Buckets corresponding to the number of reducers.

of the relation, for example, (A_1, x_1, x_2). The reducer then loops through the tuples received and merges the tuples with successful matches on x_2 and x_3.

Figure 1 illustrates an example supposing that a tuple from relation A_1 was received with $x_2 = 5$ and the hash function was (x mod 5). The results would be distributed over buckets $(0, j)$ for all j. Likewise, supposing that a tuple was received from relation A_3 with $x_3 = 2$ and the hash function was (x mod 5), the result would be distributed to buckets $(i, 2)$ for all i. Finally, if, for example, a tuple like $A_2(x_2, x_3) = (10, 3)$ was received it would be hashed to the bucket with index $(0, 3)$.

Example 4. Assume now we want to perform the following 4-way chain join:

$$A_1(x_1,x_2) \bowtie A_2(x_2,x_3) \bowtie A_3(x_3,x_4) \bowtie A_4(x_4,x_5). \quad (2)$$

We extend the same multiway join method to join tuples in a single round. As in Example 3, all middle relations will act as keys. The key will consist of three attributes $(x_2, x_3,$ and $x_4)$ hashed to $m \times m \times m$ reduce tasks. Notice that now Figure 1 becomes a 3-dimensional array of buckets. Since all three attributes do not lie in the same tuple, the *map tasks* must hash according to what attributes are available and replicate the rest. For example, the key for a tuple $A_2(x_2, x_3)$ is $(h(x_2), h(x_3), l)$, where l is a third dimension or index for the third hashed value. Likewise, the key for tuple $A_1(x_1, x_2)$ is $(i, h(x_2), l)$, tuple $A_3(x_3, x_4)$ is $(i, h(x_3), h(x_4))$, and tuple $A_4(x_4, x_5)$ is $(i, j, h(x_4))$. The value part of the `<key, value>` pair emitted by the *map task* will be as before, which is the relation name along with the actual tuple.

In general, the hash function applied on the value of attributes is different for each attribute (selected appropriately to balance the skewness introduced by the different sizes of relations). In other words a dimension of the grid of buckets in Figure 1 may expand to accommodate (balance) notably frequent values. Skewness is not handled in our paper since it requires preprocessing more details provided in [17].

The number of buckets into which the values of a certain attribute are hashed is called the *share* of this attribute. The product of all shares should be equal to the number of reducers k.

So, the communication cost is a sum of terms, one term for each relation. Each term is the product of the size of the relation multiplied by the shares of the attributes that are missing from the relation.

In Example 3, for example,

$$A_1(x_1, x_2) \bowtie A_2(x_2, x_3) \bowtie A_3(x_3, x_4), \quad (3)$$

the communication cost expression is

$$a_1 \cdot s_3 + a_2 \cdot 1 + a_3 \cdot s_2, \quad (4)$$

where a_i is the size of relation A_i and s_i is the share of attribute x_i.

In Example 4, for example,

$$A_1(x_1, x_2) \bowtie A_2(x_2, x_3) \bowtie A_3(x_3, x_4) \bowtie A_4(x_4, x_5), \quad (5)$$

the communication cost expression is

$$a_1 \cdot s_3 s_4 + a_2 \cdot s_4 + a_3 \cdot s_2 + a_4 \cdot s_2 s_3. \quad (6)$$

In [17], it is explained how we find the appropriate s_i that minimize the communication cost.

Dominance Rule. The dominance rule is discovered in [17] which makes the calculations for the communication cost expression simpler and generalizes the intuition in Examples 3 and 4 above where we did not include the end attributes in the key.

Domination Relationship. We say that an attribute Y is *dominated* by another attribute X if every relation R_i that contains Y also contains X. In other words, Y always appears with X in our join schema, while the contrary may not be true.

According to the dominance rule, if an attribute is dominated then the number of shares it gets is equal to 1.

4. Cross-Checking "Fat" Relations in MapReduce

We call *hypergraph* of a join a representation of the join's schema where a node represents an attribute and a hyperedge is a set of nodes that stands for a relation participating in the join [28]. For instance, in the case of Example 3,

$$A_1(x_1, x_2) \bowtie A_2(x_2, x_3) \bowtie A_3(x_3, x_4), \quad (7)$$

the hypergraph would consist of the hyperedges $H_1 = x_1x_2$, $H_2 = x_2x_3$, and $H_3 = x_3x_4$ (see Figure 2).

Then we define a fat hypergraph as follows: we fix a ratio ρ. We say that a hypergraph is ρ-fat if each hyperedge contains at least ρm attributes/nodes where m is the total number of attributes/nodes in the graph. Then we decide a threshold e on ρ and we define a *join with a fat hypergraph* to be one with a hypergraph which is ρ-fat with $\rho > e$.

Hereon after we refer (abusing terminology) to *fat relations* meaning the relations as they appear in a join with a fat hypergraph.

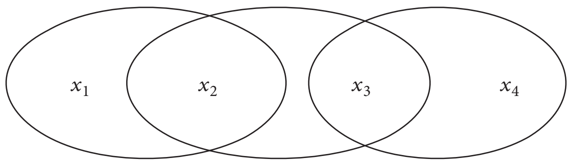


Figure 2: A hypergraph for the 3-way join.

4.1. Optimizing Shares (Examples). We will use the multiway join for fat relations. In the next example, we construct the map-key and calculate the communication cost expression for the join.

Example 5. We assume the following fat relations that form a 4-way fat join:

$$A_1(x_3, x_4, x_5) \bowtie A_2(x_1, x_4, x_5) \bowtie A_3(x_1, x_2, x_5) \bowtie A_4(x_1, x_2, x_3). \quad (8)$$

Using dominance rule for this query, we take the attributes in the middle relations to act as keys. We do not need to take all the attributes in the middle relations considering that some attributes are dominated by other attributes. Notice that x_2 is dominated by x_1, because x_2 is always showing up along with x_1 (in relations A_3 and A_4). Similarly, x_4 is dominated by x_5. However, this is not the case, for example, with attribute x_3. Attribute x_3 is accompanied by x_4 and x_5 in A_1 and with x_1 and x_2 in A_4. Thus, x_3 must be part of the map-key since it is not dominated. Finally the key is composed of $(x_1; x_3; x_5)$.

The communication cost expression for this join is (remember we used the convention that a_i is the size of relation A_i and s_i is the share of attribute x_i) as follows:

$$a_1 \cdot s_1 + a_2 \cdot s_3 + a_3 \cdot s_3 + a_4 \cdot s_5. \quad (9)$$

Observe that if all relations are of the same size, that is, $a_1 = a_2 = a_3 = a_4 = a$, then because of symmetry all the shares are equal, $s_1 = s_3 = s_5$, since we also have the constraint $s_1 \cdot s_2 \cdot s_3 \cdot s_4 = k$.

Also, notice that going back to Example 4, under the same assumption of equal relation size, the communication cost is minimized when the shares are the following: $s_2 = s_4 = \sqrt{k}$ and $s_3 = 1$.

This is not a coincidence, because, in the join of Example 5, all relations share two attributes with the map-key, whereas in Example 4 relations A_1 and A_4 share only one attribute with the map-key. This is an indication that records from those two relations will have high replication. We make this point more formal in the next subsection.

In fact checking, as we established in the Introduction, we often have to extract information from collections of relations with large overlaps over their attributes. This can be done by taking the join of all or some of those relations. We have just demonstrated that we can achieve that in MapReduce with a low communication cost. Thus, in this paper, we present an efficient way to perform fact-checking tasks.

4.2. Optimizing Shares for the Fat Relation Join. In this section, we examine the cost of using the multiway approach in the case of "fat" relations. Such relations overlap over many attributes in their schemas and each one is missing the same small number of attributes.

We define *symmetric* joins to be the joins whose associated hypergraph has adjacency matrix which has the following properties: (a) in the ith row it contains 1 in ith entry through $i+b$th entry (for a given b for this join) and 0 in all other entries, where $i+b$ is mod-d where d is the length of a row and (b) it contains $b+d$ rows. Hence, b is the number of columns.

Thus, properties of symmetric joins include the following:

(i) All relations have the same arity.

(ii) Each attribute appears in exactly $n = d + b$ relations.

Let d be the number of all attributes present in the join query Q and k the number of reducers.

Because of symmetry each share is the same: $k^{1/d}$ (so it holds that $(k^{1/d})^d = k$).

Suppose we have n relations and r tuples on each relation. If each relation participating in the join is missing 2 attributes from the join key, then each relation's tuple must be shared across $(k^{1/d})^2 = k^{2/d}$ reducers. Since there are n relations the total communication cost is $nk^{2/d}$.

The communication load per reducer is

$$q = \frac{\left(rnk^{2/d}\right)}{k} = \frac{rn}{k^{(d-2)/d}}. \quad (10)$$

Similarly, if there are i missing attributes the load is

$$q_i = \frac{rn}{k^{(d-i)/d}}. \quad (11)$$

Notice that rn is the input size of our query which we denote with IN, so that the per reducer communication cost is

$$q_i = \frac{IN}{k^{(d-i)/d}}. \quad (12)$$

Remember that d is the number of all attributes and i is the number of missing attributes at each relation. Naturally, as i decreases, the degree of the overlap is increasing, since each relation is missing fewer attributes from the join key.

We make the following observations:

(i) If i is 1, $q_1 = IN/k^{(d-1)/d}$; that is, the load for each reducer is almost IN/k, which is the cost in the case of embarrassing parallelization.

(ii) If i is close to d, for example, $d-1$, that is, the relation contains very few of the attributes present in the map-key, then the load on each reducer is almost all the input IN. In this case, the tuples need to be replicated to many reducers, which is undesirable.

(iii) In general, the load q_i for each reducer is a decreasing function of i.

5. Experimental Evaluation

We have just proven (see (12)) that joins of relations with high overlaps over their attributes perform favorably in terms of the per reducer communication cost.

We performed two separate experiments on two distinct Hadoop clusters. First cluster was scaled up, with a more powerful processor on 4 compute nodes. The nodes run Core i5 processors (2.7 GHz) with 4 GB of RAM each over a 1 Gbps Ethernet. The second cluster was scaled out with a dual core (2.4 GHz) on 8 compute nodes with 8 GB of RAM each over a 1 Gbps Ethernet.

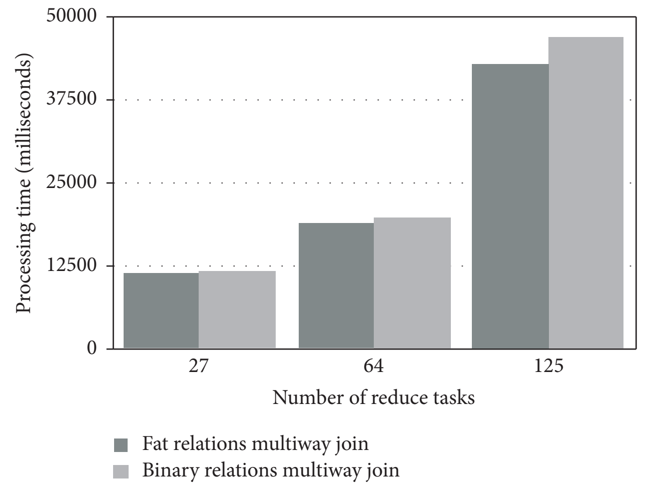
The attributes of the relations in our datasets were generated by creating a random one-digit integer using the Random class [29] from java.util library. This class generates a pseudorandom number that uses a uniform distribution; that is, no skewness is present in the datasets. A one-digit number pertains a smaller alphabet of possible values for the attributes. This causes more matches between attributes and more joins to materialize for a rigorous evaluation. We use an integer number for input as opposed to strings for easier management of the data and faster processing. To retain these benefits when dealing with string data we could use dictionary encoding or compression techniques to encode string input data into numbers and decode the latter after processing.

The number of reducers was chosen based on the number of buckets as established earlier (see Figure 1). Thus if the hash function was mod 3 and with a key size of 3 (three dimensions) the number of reducers/buckets will amount to $3 * 3 * 3 = 27$ reducers. Likewise if the hash function was mod 3 and with a key size of 4 the number of reducers/buckets will amount to $3 * 3 * 3 * 3 = 81$ reducers.

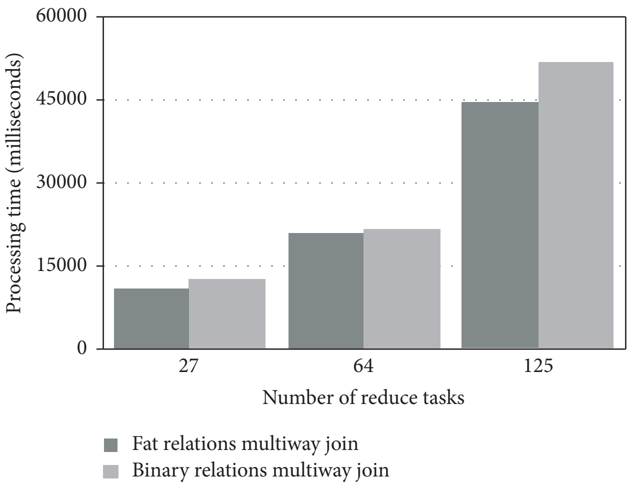
In the first set of experiments on the first cluster we assume the evaluation of two joins—the 4-way fat relation join where the degree of overlap is two out of three attributes in adjacent relations (join 1 shown below) and the 4-way binary relation (i.e., nonfat) join (join 2 shown below) to demonstrate that the first performs better both in communication load and in wall-clock time.

Regarding the first experiment, we designed an experiment where both queries have their relation tuples distributed from the mappers to the reducers. However, in this first case, we choose not to perform the actual join on the reduce side, in order to have a clearer comparison of the communication cost for both joins. We call this scenario piped through due to the fact that tuples from the mapper are only relayed to the reducer and no work is being done by the reducers. As seen in Figures 3(a) and 3(b), processing times are consistently in favor of the fat relation join, regardless of the number of reduce tasks.

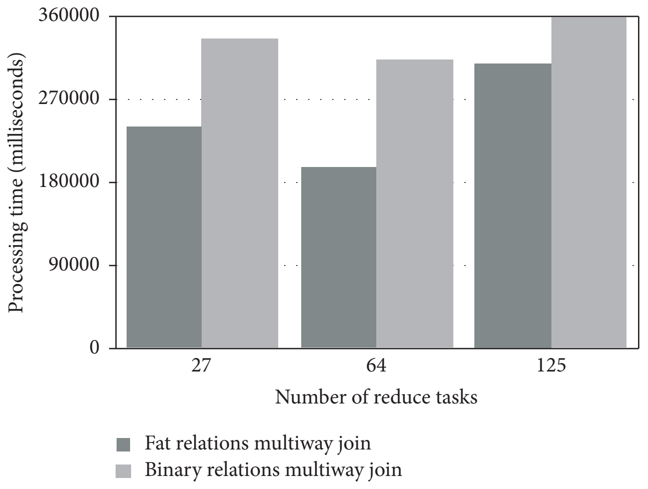
Furthermore, we discover that this advantage is amplified when we let the reducers actually join the tuples that come their way. Although the computation time dominates the total evaluation time for the join queries, the fat relation join performs significantly better (see Figures 3(c) and 3(d)). This is a direct product of the lower per reducer communication load in the case of the fat relation join. A reducer with less tuples to join has always less work to do; thus a lower communication



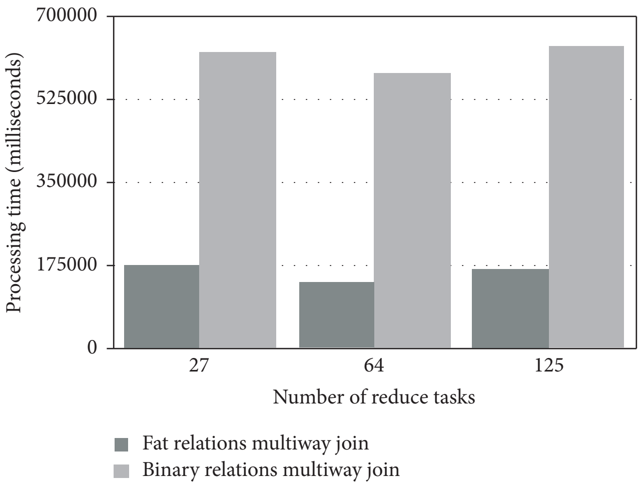
(a) 20,000 records piped through



(b) 24,000 records piped through



(c) 20,000 records



(d) 24,000 records

FIGURE 3: Processing time comparison of binary multiway versus fat multiway join.

cost has a carry-over effect on the computation cost, making the choice to minimize the former a right one.

Apart from the comparison of binary join and fat join you will notice, if we compare number of reducers used for each type of join alone, that while increasing the number of reducers to 125 reducers the processing time also increases. This has to do with finding the optimal number of reducers. More reducers do not necessarily mean faster execution due to other determinants that influence the processing time such as the incurred overhead setting up the reducer and balance between reduce jobs. Also notice in Figures 3(c) and 3(d) that 64 reducers have a lower processing time than 27 and 125 reducers; this is a direct product of the reducer job per-node balance. Considering that our cluster consists of 4 nodes, an even number of reducers will be evenly balanced among these nodes, that is, exactly 16 reduce jobs per node.

In the second set of experiments on the second cluster aimed at comparing execution speed with regard to degree of overlap, we experiment with another two 5-way fat joins (see joins 3 and 4) with a higher degree of overlap including the previous 4-way fat join 1. Joins 3 and 4 are similar in the degree of overlap (3 out of attributes overlap) but join 4 is in a sense more sparse and some attributes do not get cross-checked. For example, x_4 and x_5 only appear once so they remain un-cross-checked. On the other hand join 3 is more dense where each attribute appears at least three times and thereby must be matched three times. The downside is that in join 3, due to many attributes across several relations, the dominance rule will not be taken advantage of effectively to reduce the key size thereby leading to key size of four parts (x_1, x_3, x_4, and x_6) causing more replication. Compared to join 3, join 4 will have a two-part key (x_1, x_8).

Table 1 shows that the two new joins with higher overlap perform better than join 1. Also, surprisingly, join 4 that is dense performed better than the sparse join 3 despite the fact it would cause more replication (see Figure 4):

$$A_1(x_3, x_4, x_5) \bowtie A_2(x_1, x_4, x_5) \bowtie A_3(x_1, x_2, x_5) \bowtie A_4(x_1, x_2, x_3),$$

Table 1: Fat relations processing times (in milliseconds) on 8 compute nodes.

		Join 1: 2 out of 3 overlap			Join 3: 3 out of 4 overlap			Join 4: 3 out of 4 overlap
		# of reducers						
		27	64	125	9	16	25	81
# of records	24,000	490840	351080	446020	38230	44970	61800	51780
	28,000	633200	466330	470890	61010	61200	67680	56420
	32,000	1068060	767200	694910	99940	95430	95280	60810
	36,000	1687300	1174030	1026560	159570	145420	145860	66490
	40,000	2620760	1852560	1484280	268030	245040	291930	74070

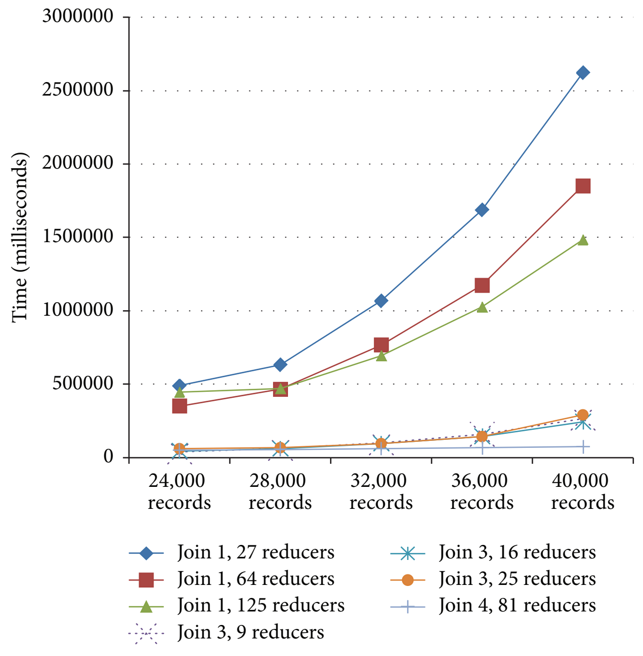


Figure 4: Processing time comparison of fat joins with different degrees of overlap.

$$\begin{aligned}
&A_1(x_1,x_2) \bowtie A_2(x_2,x_3) \bowtie A_3(x_3,x_4) \\
&\quad \bowtie A_4(x_4,x_5), \\
&A_1(x_3,x_4,x_5,x_6) \bowtie A_2(x_1,x_4,x_5,x_6) \\
&\quad \bowtie A_3(x_1,x_2,x_5,x_6) \\
&\quad \bowtie A_4(x_1,x_2,x_3,x_6) \\
&\quad \bowtie A_5(x_1,x_2,x_3,x_4), \\
&A_1(x_5,x_6,x_7,x_8) \bowtie A_2(x_1,x_6,x_7,x_8) \\
&\quad \bowtie A_3(x_1,x_2,x_7,x_8) \\
&\quad \bowtie A_4(x_1,x_2,x_3,x_8) \\
&\quad \bowtie A_5(x_1,x_2,x_3,x_4).
\end{aligned} \tag{13}$$

6. Exploiting Our Approach

Computing similarity multiway joins may be more relevant to fact checking since when dealing with multiple sources we do not expect all of them to be reliable or nonconflicting. There are very few works on that [18] and many of which address this problem in multiple rounds of MapReduce. In this section, we explain how our approach can be exploited by just adding a preprocessing layer to be able to compute similarity multiway joins in the presence of typing errors.

Typing errors or errors with a small data footprint are a common annoyance in nowadays web. This kind of errors is a constant threat to our attempts to make sense of the abundant data available to us. Such small deviations, commonly meaningless, are severely hampering our algorithms performance. For example, misspelling the name of Tom Cruise across different sources and then trying to combine their contribution to our knowledge will be pointless as desired aspects of the entity in the name will never meet.

For the scenario described in this section, we opt to add preprocessing layer to the preestablished cross-checking method resulting in a two-step approach. Each step is a separate round of MapReduce. The first step will detect similarities within a single attribute class to accommodate for minor typing errors in values, whereas during the second step we take into account multiple attributes of each input tuple, using the original technique described in Section 4.

We now describe our approach (see also Figure 5). Here are the two steps.

Step 1. We use the similarity join ideas in [24, 30] to identify similar values in a single attribute, that is, values that only differ according to a certain metric distance, for example, the edit distance. For example, a common misspelling of the name of Tom Cruise is Tom Cruz and the edit distance between those two values is $d = 3$. We use an algorithm that discovers pairs of similar values, such as [30]. Now, since in our scenario, we care for common typo errors we make the following assumption:

(i) The pairs found in each reducer form almost disjoint classes where there are no similar values across classes.

This is a reasonable assumption since we do not expect, for example, that classes for Tom Cruise and for Angelina Jolie

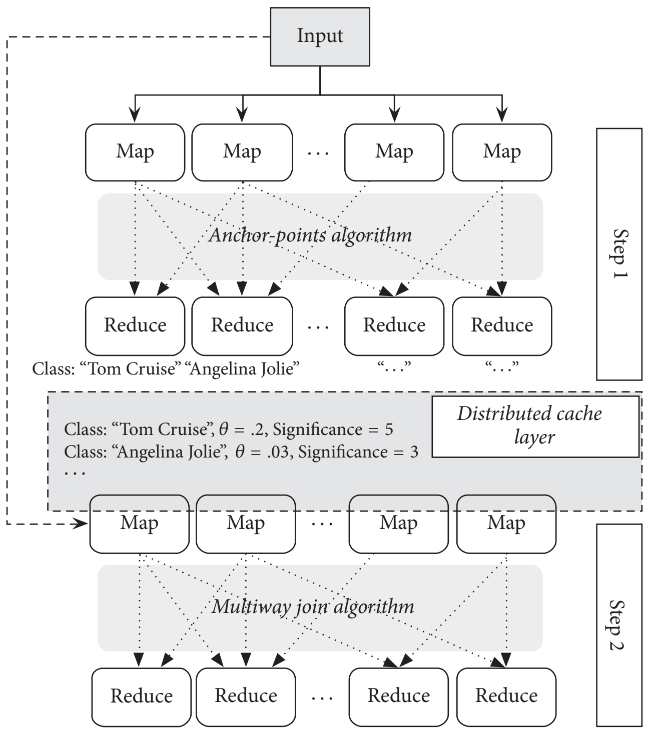


Figure 5: Step 1: use anchor-points algorithm. Step 2: exploit fat relations to distribute tuples optimally.

will share the same value or, otherwise, that a variation of Tom Cruise's name will coincide with a variation of Angelina Jolie's name.

We want to constrain the number of classes formed because we want the typos dictionary size B to fit the Distributed Cache. We achieve this by

(i) manipulating the threshold of the edit distance d;

(ii) taking into account the relative number θ of variations on the total number of members in the class;

(iii) taking into account the significance of a class, that is, the number of members in this class.

We may choose to take some or all of the above condition as it applies.

Given a set of input strings of fixed length n over some alphabet, we find pairs of strings that are within distance d, that is, differ in at most d positions. The anchor-points algorithm described there uses a set A of anchor-point strings such that all strings of length n are within distance d of some anchor point in A. The algorithm operates by creating one reducer for each anchor point. The mappers send each string w to the reducer for each anchor point at distance at most $2d$ from w. Each reducer then searches for strings at a distance up to d from each other, among the strings it has received. While not always the best algorithm, [30] showed that, for some inputs and parameters, anchor-points algorithm is the best among known algorithms.

Step 2. We send to each mapper of the second round the original data and the classes found in Step 1 alongside the respective θ and significance metrics. Since, as we have argued, this number of classes is not large in most of the scenarios we consider, this information can be distributed to each mapper using the *Distributed Cache* mechanism of the Hadoop framework.

7. Conclusions and Future Work

We have demonstrated an efficient way to match information coming from different data sources. Attributed to the multiway join algorithm we are able to perform matches in a single MapReduce job. Aside from that the algorithm performs exceptionally well due to the overlap that exists between data sources. Experimental results show that there is an exceptional gain in speed of execution as the overlap degree increases. Furthermore if the attributes being cross-checked are characterized as dense across the relations participating in a join this would also result in an additional increase in speed of execution (see Figure 4). Building on top of this method we intend to refine cross-checking of data. Here are possible ways to achieve that:

(i) Dangling records that did not join completely on the reducer can be either partially matching or referring to completely different entities. For the former, we may decide to lower our join threshold by using smaller map-keys. However, this comes with the inefficiencies we discussed earlier in Section 4.2.

(ii) By counting the number of records matching from each origin data source, we can measure the authority of the data source. The authority of a data source is an indicator of how much we trust it. Knowing about the trustworthiness of source, we will then characterize facts coming from it.

(iii) Records that join at the reducers match on their critical (map-key) attributes. We can assume that they should match also in other attributes. For example, actors matching on name and age may come with slightly different telephone numbers which is not a prerequisite for entity matching. We can correct errors this way at the reducer by employing similarity techniques.

(iv) https://Data.gov [31] claims having up to 400,000 datasets. Our method can be very useful since many of these datasets have missing values and characters.

(v) We can cross-check our results with RDF data, but since HDFS only stores and manipulates flat files, RDF triples will have to be transformed into tabular form.

Disclosure

This paper is a significantly extended version of previous work published in the proceedings of the East European Conference on Advances in Databases and Information Systems (ADBIS 2015) [13].

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] F. T. Juster and J. P. Smith, "Improving the quality of economic data: lessons from the HRS and AHEAD," *Journal of the American Statistical Association*, vol. 92, no. 440, pp. 1268–1278, 1997.

[2] J. W. Graham, "Missing data analysis: making it work in the real world," *Annual Review of Psychology*, vol. 60, pp. 549–576, 2009.

[3] A. C. Acock, "Working with missing values," *Journal of Marriage and Family*, vol. 67, no. 4, pp. 1012–1028, 2005.

[4] A. Holzinger, M. Dehmer, and I. Jurisica, "Knowledge discovery and interactive data mining in bioinformatics—state-of-the-art, future challenges and research directions," *BMC Bioinformatics*, vol. 15, no. 6, pp. 1–9, 2014.

[5] S. F. Messner, "Exploring the consequences of erratic data reporting for cross-national research on homicide," *Journal of Quantitative Criminology*, vol. 8, no. 2, pp. 155–173, 1992.

[6] A. M. Wood, I. R. White, and S. G. Thompson, "Are missing outcome data adequately handled? A review of published randomized controlled trials in major medical journals," *Clinical Trials*, vol. 1, no. 4, pp. 368–376, 2004.

[7] J. W. Grzymala-Busse and M. Hu, "A comparison of several approaches to missing attribute values in data mining," in *Rough Sets and Current Trends in Computing*, pp. 378–385, Springer, 2001.

[8] B. Padmanabhan, Z. Zheng, and S. O. Kimbrough, "Personalization from incomplete data: what you don't know can hurt," in *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '01)*, pp. 154–163, ACM, August 2001.

[9] M. Magnani, "Techniques for dealing with missing data in knowledge discovery tasks," *Obtido*, vol. 15, no. 1, article 2007, 2004.

[10] X. Li, X. L. Dong, K. Lyons, W. Meng, and D. Srivastava, "Truth finding on the deep web: is the problem solved?" *Proceedings of the VLDB Endowment*, vol. 6, no. 2, pp. 97–108, 2012.

[11] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 39, no. 1, pp. 1–38, 1977.

[12] X. L. Dong, L. Berti-Equille, and D. Srivastava, "Integrating conflicting data: the role of source dependence," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 550–561, 2009.

[13] F. Afrati, Z. Momani, and N. Stasinopoulos, "Cross-checking data sources in MapReduce," in *New Trends in Databases and Information Systems*, vol. 539 of *Communications in Computer and Information Science*, pp. 165–174, Springer International Publishing, Cham, Switzerland, 2015.

[14] F. N. Afrati, D. Delorey, M. Pasumansky, and J. D. Ullman, "Storing and querying tree-structured records in Dremel," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1131–1142, 2014.

[15] S. Melnik, A. Gubarev, J. J. Long et al., "Dremel: Interactive analysis of web-scale datasets," *Communications of the ACM*, vol. 54, no. 6, pp. 114–123, 2011.

[16] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[17] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *Proceedings of the 13th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '10)*, pp. 99–110, ACM, March 2010.

[18] C. Doulkeridis and K. Nørvåg, "A survey of large-scale analytical query processing in MapReduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, 2014.

[19] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, pp. 495–506, ACM, June 2010.

[20] Y. Kim and K. Shim, "Parallel top-k similarity join algorithms using MapReduce," in *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE '12)*, pp. 510–521, IEEE, April 2012.

[21] A. Metwally and C. Faloutsos, "V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 704–715, 2012.

[22] R. Baraglia, G. De Francisci Morales, and C. Lucchese, "Document similarity self-join with MapReduce," in *Proceedings of the 10th IEEE International Conference on Data Mining (ICDM '10)*, pp. 731–736, IEEE, December 2010.

[23] Y. N. Silva, J. M. Reed, and L. M. Tsosie, "MapReduce-based similarity join for metric spaces," in *Proceedings of the 1st International Workshop on Cloud Intelligence*, p. 3, ACM, August 2012.

[24] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman, "Fuzzy joins using MapReduce," in *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE '12)*, pp. 498–509, IEEE, Washington, DC, USA, April 2012.

[25] X. L. Dong, E. Gabrilovich, G. Heitz et al., "From data fusion to knowledge fusion," in *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 881–892, June 2014.

[26] L. Kolb and E. Rahm, "Parallel entity resolution with dedoop," *Datenbank-Spektrum*, vol. 13, no. 1, pp. 23–32, 2013.

[27] L. Kolb, A. Thor, and E. Rahm, "Don't match twice: redundancy-free similarity computation with MapReduce," in *Proceedings of the 2nd Workshop on Data Analytics in the Cloud*, pp. 1–5, ACM, June 2013.

[28] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems—The Complete Book*, Pearson Education, 2nd edition, 2009.

[29] Oracle, Class Random Java Documentation, https://docs.oracle.com/javase/7/docs/api/java/util/Random.html.

[30] F. N. Afrati, A. D. Sarma, A. Rajaraman, P. Rule, S. Salihoglu, and J. Ullman, "Anchor-points algorithms for hamming and edit distances using mapreduce," in *Proceedings of the 17th International Conference on Database Theory (ICDT '14)*, pp. 4–14, Athens, Greece, March 2014.

[31] U.S. General Services Administration, U.S. government's open data, 2013, http://www.data.gov/.