Efficient Parallel Set-Similarity Joins Using MapReduce

Rares Vernica
Department of Computer Science
University of California, Irvine
rares@ics.uci.edu

Michael J. Carey
Department of Computer Science
University of California, Irvine
mjcarey@ics.uci.edu

Chen Li
Department of Computer Science
University of California, Irvine
chenli@ics.uci.edu

ABSTRACT

In this paper we study how to efficiently perform set-similarity joins in parallel using the popular MapReduce framework. We propose a 3-stage approach for end-to-end set-similarity joins. We take as input a set of records and output a set of joined records based on a set-similarity condition. We efficiently partition the data across nodes in order to balance the workload and minimize the need for replication. We study both self-join and R-S join cases, and show how to carefully control the amount of data kept in main memory on each node. We also propose solutions for the case where, even if we use the most fine-grained partitioning, the data still does not fit in the main memory of a node. We report results from extensive experiments on real datasets, synthetically increased in size, to evaluate the speedup and scaleup properties of the proposed algorithms using Hadoop.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, parallel databases*

General Terms

Algorithms, Performance

1. INTRODUCTION

There are many applications that require detecting similar pairs of records where the records contain string or set-based data. A list of possible applications includes: detecting near duplicate web-pages in web crawling [14], document clustering[5], plagiarism detection [15], master data management, making recommendations to users based on their similarity to other users in query refinement [22], mining in social networking sites [25], and identifying coalitions of click fraudsters in online advertising [20]. For example, in master-data-management applications, a system has to identify that names "`John W. Smith`", "`Smith, John`", and "`John William Smith`" are potentially referring to the same person. As another example, when mining social networking sites where users' preferences are stored as bit vectors (where a "`1`" bit means interest in a certain domain), applications want to use the fact that a user with preference bit vector "`[1,0,0,1,1,0,1,0,0,1]`" possibly has similar interests to a user with preferences "`[1,0,0,0,1,0,1,0,1,1]`".

Detecting such similar pairs is challenging today, as there is an increasing trend of applications being expected to deal with vast amounts of data that usually do not fit in the main memory of one machine. For example, the Google N-gram dataset [27] has 1 trillion records; the GeneBank dataset [11] contains 100 million records and has a size of 416 GB. Applications with such datasets usually make use of clusters of machines and employ parallel algorithms in order to efficiently deal with this vast amount of data. For data-intensive applications, the MapReduce [7] paradigm has recently received a lot of attention for being a scalable parallel shared-nothing data-processing platform. The framework is able to scale to thousands of nodes [7]. In this paper, we use MapReduce as the parallel data-processing paradigm for finding similar pairs of records.

When dealing with a very large amount of data, detecting similar pairs of records becomes a challenging problem, even if a large computational cluster is available. Parallel data-processing paradigms rely on data partitioning and redistribution for efficient query execution. Partitioning records for finding similar pairs of records is challenging for string or set-based data as hash-based partitioning using the entire string or set does not suffice. The contributions of this paper are as follows:

- We describe efficient ways to partition a large dataset across nodes in order to balance the workload and minimize the need for replication. Compared to the equi-join case, the set-similarity joins case requires "partitioning" the data based on set contents.
- We describe efficient solutions that exploit the MapReduce framework. We show how to efficiently deal with problems such as partitioning, replication, and multiple inputs by manipulating the keys used to route the data in the framework.
- We present methods for controlling the amount of data kept in memory during a join by exploiting the properties of the data that needs to be joined.
- We provide algorithms for answering set-similarity self-join queries *end-to-end*, where we start from records containing more than just the join attribute and end with actual pairs of joined records.

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

- We show how our set-similarity self-join algorithms can be extended to answer set-similarity R-S join queries.
- We present strategies for exceptional situations where, even if we use the finest-granularity partitioning method, the data that needs to be held in the main memory of one node is too large to fit.

The rest of the paper is structured as follows. In Section 2 we introduce the problem and present the main idea of our algorithms. In Section 3 we present set-similarity join algorithms for the self-join case, while in Section 4 we show how the algorithms can be extended to the R-S join case. Next, in Section 5, we present strategies for handling the insufficient-memory case. A performance evaluation is presented in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8. A longer technical report on this work is available in [26].

2. PRELIMINARIES

In this work we focus on the following set-similarity join application: identifying similar records based on string similarity. Our results can be generalized to other set-similarity join applications.

Problem statement: Given two files of records, R and S, a set-similarity function, sim, and a similarity threshold τ, we define the set-similarity join of R and S on $R.a$ and $S.a$ as finding and combining all pairs of records from R and S where $sim(R.a, S.a) \geq \tau$.

We map strings into sets by tokenizing them. Examples of tokens are words or q-grams (overlapping sub-strings of fixed length). For example, the string "`I will call back`" can be tokenized into the word set `[I, will, call, back]`. In order to measure the similarity between strings, we use a set-similarity function such as Jaccard or Tanimoto coefficient, cosine coefficient, etc.[1]. For example, the Jaccard similarity function for two sets x and y is defined as: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$. Thus, the Jaccard similarity between strings "`I will call back`" and "`I will call you soon`" is $\frac{3}{6} = 0.5$.

In the remainder of the section, we provide an introduction to the MapReduce paradigm, present the main idea of our parallel set-similarity join algorithms, and provide an overview of filtering methods for detecting set-similar pairs.

2.1 MapReduce

MapReduce [7] is a popular paradigm for data-intensive parallel computation in shared-nothing clusters. Example applications for the MapReduce paradigm include processing crawled documents, Web request logs, etc. In the open-source community, Hadoop [1] is a poplar implementation of this paradigm. In MapReduce, data is initially partitioned across the nodes of a cluster and stored in a distributed file system (DFS). Data is represented as `(key, value)` pairs. The computation is expressed using two functions:

```
map    (k1,v1)       → list(k2,v2);
reduce (k2,list(v2)) → list(k3,v3).
```

Figure 1 shows the data flow in a MapReduce computation. The computation starts with a map phase in which the `map` functions are applied in parallel on different partitions of the input data. The `(key, value)` pairs output by each `map` function are hash-partitioned on the `key`. For each partition the pairs are sorted by their `key` and then sent across the cluster in a shuffle phase. At each receiving node, all the received partitions are merged in a sorted order by their `key`. All the pair values that share a certain key are passed to a single reduce call. The output of each `reduce` function is written to a distributed file in the DFS.

[1]The techniques described in this paper can also be used for approximate string search using the edit or Levenshtein distance [13].

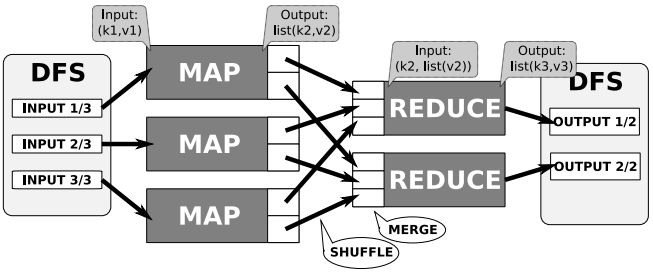


Figure 1: Data flow in a MapReduce computation.

Besides the `map` and `reduce` functions, the framework also allows the user to provide a `combine` function that is executed on the same nodes as mappers right after the `map` functions have finished. This function acts as a local reducer, operating on the local `(key, value)` pairs. This function allows the user to decrease the amount of data sent through the network. The signature of the `combine` function is:

```
combine (k2,list(v2)) → list(k2,list(v2)).
```

Finally, the framework also allows the user to provide initialization and tear-down function for each MapReduce function and customize hashing and comparison functions to be used when partitioning and sorting the keys. From Figure 1 one can notice the similarity between the MapReduce approach and query-processing techniques for parallel DBMS [8, 21].

2.2 Parallel Set-Similarity Joins

One of the main issues when answering set-similarity joins using the MapReduce paradigm, is to decide how data should be partitioned and replicated. The main idea of our algorithms is the following. The framework hash-partitions the data across the network based on keys; data items with the same key are grouped together. In our case, the join-attribute value cannot be directly used as a partitioning key. Instead, we use (possibly multiple) *signatures* generated from the value as partitioning keys. Signatures are defined such that similar attribute values have at least one signature in common. Possible example signatures include: the list of word tokens of a string and ranges of similar string lengths. For instance, the string "`I will call back`" would have 4 word-based signatures: "`I`", "`will`", "`call`", and "`back`".

We divide the processing into three stages:

- **Stage 1:** Computes data statistics in order to generate good signatures. The techniques in later stages utilize these statistics.
- **Stage 2:** Extracts the record IDs ("RID") and the join-attribute value from each record and distributes the RID and the join-attribute value pairs so that the pairs sharing a signature go to at least one common reducer. The reducers compute the similarity of the join-attribute values and output RID pairs of similar records.

- **Stage 3:** Generates actual pairs of joined records. It uses the list of RID pairs from the second stage and the original data to build the pairs of similar records.

An alternative to using the second and third stages is to use one stage in which we let key-value pairs carry complete records, instead of projecting records on their RIDs and join-attribute values. We implemented this alternative and noticed a much worse performance, so we do not consider this option in this paper.

2.3 Set-Similarity Filtering

Efficient set-similarity join algorithms rely on effective filters, which can decrease the number of candidate pairs whose similarity needs to be verified. In the past few years, there have been several studies involving a technique called *prefix filtering* [6, 4, 29], which is based on the pigeonhole principle and works as follows. The tokens of strings are ordered based on a global token ordering. For each string, we define its *prefix* of length n as the first n tokens of the ordered set of tokens. The required length of the prefix depends on the size of the token set, the similarity function, and the similarity threshold. For example, given the string, s, "`I will call back`" and the global token ordering `{back, call, will, I}`, the prefix of length 2 of s is `[back, call]`. The prefix filtering principle states that similar strings need to share at least one common token *in their prefixes.* Using this principle, records of one relation are organized based on the tokens in their prefixes. Then, using the prefix tokens of the records in the second relation, we can probe the first relation and generate candidate pairs. The prefix filtering principle gives a necessary condition for similar records, so the generated candidate pairs need to be verified. A good performance can be achieved when the global token ordering corresponds to their increasing token-frequency order, since fewer candidate pairs will be generated.

A state-of-the-art algorithm in the set-similarity join literature is the PPJoin+ technique presented in [29]. It uses the prefix filter along with a length filter (similar strings need to have similar lengths [3]). It also proposed two other filters: a positional filter and a suffix filter. The PPJoin+ technique provides a good solution for answering such queries on one node. One of our approaches is to use PPJoin+ in parallel on multiple nodes.

3. SELF-JOIN CASE

In this section we present techniques for the set-similarity self-join case. As outlined in the previous section, the solution is divided into three stages. The first stage builds the global *token ordering* necessary to apply the prefix-filter.[2] It scans the data, computes the frequency of each token, and sorts the tokens based on frequency. The second stage uses the prefix-filtering principle to produce a list of similar-RID pairs. The algorithm extracts the RID and join-attribute value of each record, and replicates and re-partitions the records based on their prefix tokens. The MapReduce framework groups the RID and join-attribute value pairs based on the prefix tokens. It is worth noting that using the infrequent prefix tokens to redistribute the data helps us avoid unbalanced workload due to token-frequency skew. Each group represents a set of candidates that are cross paired and verified. The third stage uses the list of similar-RID pairs and the original data to generate pairs of similar records.

[2]An alternative would be to apply the length filter. We explored this alternative but the performance was not good because it suffered from the skewed distribution of string lengths.

3.1 Stage 1: Token Ordering

We consider two methods for ordering the tokens in the first stage. Both approaches take as input the original records and produce a list of tokens that appear in the join-attribute value ordered increasingly by frequency.

3.1.1 Basic Token Ordering (BTO)

Our first approach, called Basic Token Ordering ("BTO"), relies on two MapReduce phases. The first phase computes the frequency of each token and the second phase sorts the tokens based on their frequencies. In the first phase, the `map` function gets as input the original records. For each record, the function extracts the value of the join attribute and tokenizes it. Each token produces a `(token, 1)` pair. To minimize the network traffic between the `map` and `reduce` functions, we use a `combine` function to aggregates the 1's output by the `map` function into partial counts. Figure 2(a) shows the data flow for an example dataset, self-joined on an attribute called "`a`". In the figure, for the record with RID 1, the join-attribute value is "`A B C`", which is tokenized as "`A`", "`B`", and "`C`". Subsequently, the `reduce` function computes the total count for each token and outputs `(token, count)` pairs, where "`count`" is the total frequency for the token.

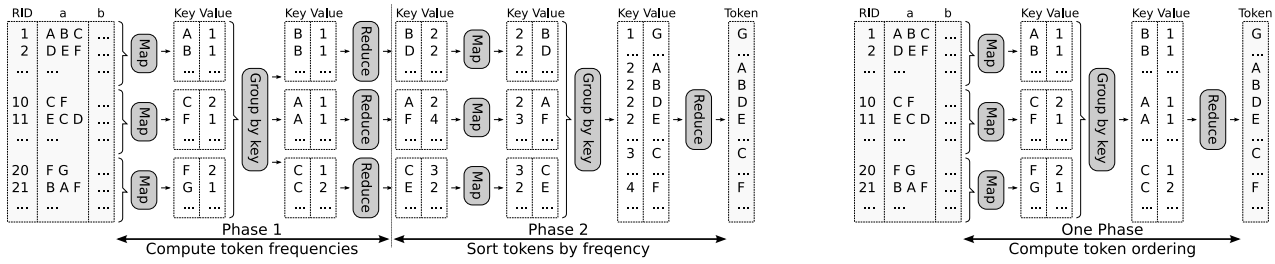
The second phase uses MapReduce to sort the pairs of tokens and frequencies from the first phase. The `map` function swaps the input keys and values so that the input pairs of the `reduce` function are sorted based on their frequencies. This phase uses exactly one reducer so that the result is a totally ordered list of tokens. The pseudo-code of this algorithm and other algorithms presented is available in [26].

3.1.2 Using One Phase to Order Tokens (OPTO)

An alternative approach to token ordering is to use one MapReduce phase. This approach, called One-Phase Token Ordering ("OPTO"), exploits the fact that the list of tokens could be much smaller than the original data size. Instead of using MapReduce to sort the tokens, we can explicitly sort the tokens in memory. We use the same `map` and `combine` functions as in the first phase of the BTO algorithm. Similar to BTO we use only one reducer. Figure 2(b) shows the data flow of this approach for our example dataset. The `reduce` function in OPTO gets as input a list of tokens and their partial counts. For each token, the function computes its total count and stores the information locally. When there is no more input for the `reduce` function, the reducer calls a tear-down function to sort the tokens based on their counts, and to output the tokens in an increasing order of their counts.

3.2 Stage 2: RID-Pair Generation

The second stage of the join, called "Kernel", scans the original input data and extracts the prefix of each record using the token order computed by the first stage. In general the list of unique tokens is much smaller and grows much slower than the list of records. We thus assume that the list of tokens fits in memory. Based on the prefix tokens, we extract the RID and the join-attribute value of each record, and distribute these record projections to reducers. The



(a) Basic Token Ordering (BTO)

(b) One-Phase Token Ordering (OPTO)

Figure 2: Example data flow of Stage 1. (Token ordering for a self-join on attribute "a".)

join-attribute values that share at least one prefix token are verified at a reducer.

Routing Records to Reducers. We first take a look at two possible ways to generate `(key, value)` pairs in the `map` function. (1) *Using Individual Tokens*: This method treats each token as a key. Thus, for each record, we would generate a `(key, value)` pair for each of its prefix tokens. Thus, a record projection is replicated as many times as the number of its prefix tokens. For example, if the record value is "`A B C D`" and the prefix tokens are "`A`", "`B`", and "`C`", we would output three `(key, value)` pairs, corresponding to the three tokens. In the reducer, as the values get grouped by prefix tokens, all the values passed in a `reduce` call share the same prefix token.

(2) *Using Grouped Tokens*: This method maps multiple tokens to one synthetic key, thus can map different tokens to the same key. For each record, the `map` function generates one `(key, value)` pair for each of the groups of the prefix tokens. In our running example of a record "`A B C D`", if tokens "`A`" and "`B`" belong to one group (denoted by "`X`"), and token "`C`" belongs to another group (denoted by "`Y`"), we output two `(key, value)` pairs, one for key "`X`" and one for key "`Y`". Two records that share the same token group do not necessarily share any prefix token. Continuing our running example, for record "`E F G`", if its prefix token "`E`" belongs to group "`Y`", then the records "`A B C D`" and "`E F G`" share token group "`Y`" but do not share any prefix token. So, in the reducer, as the values get grouped by their token group, no two values share a prefix token. This method can help us have fewer replications of record projections. One way to define the token groups in order to balance data across reducers is the following. We use the token ordering produced in the first stage, and assign the tokens to groups in a Round-Robin order. In this way we balance the sum of token frequencies across groups. We study the effect of the number of groups in Section 6. For both routing strategies, since two records might share more that one prefix token, the same pair may be verified multiple times at different reducers, thus it could be output multiple times. This is dealt with in the third stage.

3.2.1 Basic Kernel (BK)

In our first approach to finding the RID pairs of similar records, called Basic Kernel ("BK"), each reducer uses a nested loop approach to compute the similarity of the join-attribute values. Before the `map` functions begin their executions, an initialization function is called to load the ordered tokens produced by the first stage. The `map` function then retrieves the original records one by one, and extracts the RID and the join-attribute value for each record. It tokenizes the join attribute and reorders the tokens based on their frequencies. Next, the function computes the prefix length and extracts the prefix tokens. Finally, the function uses either the individual tokens or the grouped tokens routing strategy to generate the output pairs. Figure 3(a) shows the data flow for our example dataset using individual tokens to do the routing. The prefix tokens of each value are in bold face. The record with RID 1 has prefix tokens "`A`" and "`B`", so its projection is output twice.

In the `reduce` function, for each pair of record projections, the reducer applies the additional filters (e.g., length filter, positional filter, and suffix filter) and verifies the pair if it survives. If a pair passes the similarity threshold, the reducer outputs RID pairs and their similarity values.

3.2.2 Indexed Kernel (PK)

Another approach on finding RID pairs of similar records is to use existing set-similarity join algorithms from the literature [23, 3, 4, 29]. Here we use the PPJoin+ algorithm from [29]. We call this approach the PPJoin+ Kernel ("PK").

Using this method, the `map` function is the same as in the BK algorithm. Figure 3(b) shows the data flow for our example dataset using grouped tokens to do the routing. In the figure, the record with RID 1 has prefix tokens "`A`" and "`B`", which belong to groups "`X`" and "`Y`", respectively. In the `reduce` function, we use the PPJoin+ algorithm to index the data, apply all the filters, and output the resulting pairs. For each input record projection, the function first probes the index using the join-attribute value. The probe generates a list of RIDs of records that are similar to the current record. The current record is then added to the index as well.

The PPJoin+ algorithm achieves an optimized memory footprint because the input strings are sorted increasingly by their lengths [29]. This works in the following way. The index knows the lower bound on the length of the unseen data elements. Using this bound and the length filter, PPJoin+ discards from the index the data elements below the minimum length given by the filter. In order to obtain this ordering of data elements, we use a composite MapReduce key that also includes the length of the join-attribute value. We provide the framework with a custom partitioning function so that the partitioning is done only on the group value. In this way, when data is transferred from `map` to `reduce`, it gets partitioned just by group value, and is then locally sorted on both group and length.

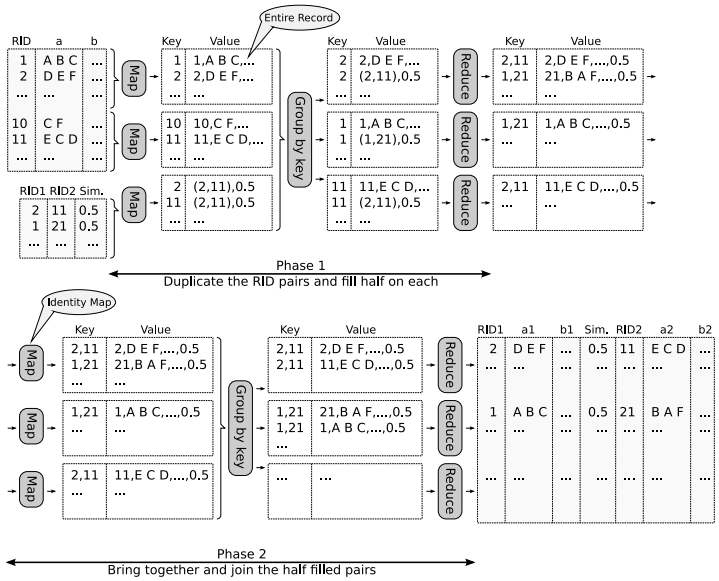


Figure 4: Example data flow of Stage 3 using Basic Record Join (BRJ) for a self-join case. "a1" and "a2" correspond to the original attribute "a", while "b1" and "b2" correspond to attribute "b".

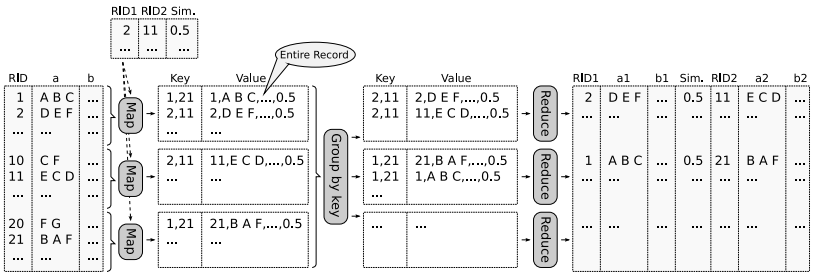


Figure 5: Example data flow of Stage 3 using One-Phase Record Join (OPRJ) for a self-join case. "a1" and "a2" correspond to the original attribute "a" while, "b1" and "b2" correspond to attribute "b".

modify the partitioning function so that partitioning is done on the part of the key that does not include the relation name. (However, the sorting is still done on the full key.) We now explain the three stages of an R-S join.

Stage 1: Token Ordering. In the first stage, we use the same algorithms as in the self-join case, only on the relation with fewer records, say R. In the second stage, when tokenizing the other relation, S, we discard the tokens that do not appear in the token list, since they cannot generate candidate pairs with R records.

Stage 2: Basic Kernel. First, the mappers tag the record projections with their relation name. Thus, the reducers receive a list of record projections grouped by relation. In the `reduce` function, we then store the records from the first relation (as they arrive first), and stream the records from the second relation (as they arrive later). For each record in the second relation, we verify it against all the records in the first relation.

Stage 2: Indexed Kernel. We use the same mappers as for the Basic Kernel. The reducers index the record projections of the first relation and probe the index for the record projections of the second relation.

As in the self-join case, we can improve the memory footprint of the `reduce` function by having the data sorted increasing by their lengths. PPJoin+ only considered this improvement for self-joins. For R-S joins, the challenge is that we need to make sure that we first stream all the record projections from R that might join with a particular S record before we stream this record. Specifically, given the length of a set, we can define a lower-bound and an upper-bound on the lengths of the sets that might join with it [3]. Be-

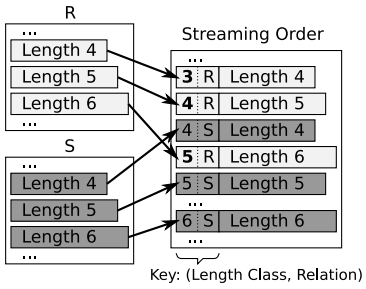


Figure 6: Example of the order in which records need to arrive at the reducer in the PK kernel of the R-S join case, assuming that for each length, l, the lower-bound is $l-1$ and the upper-bound is $l+1$.