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.

- 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

---

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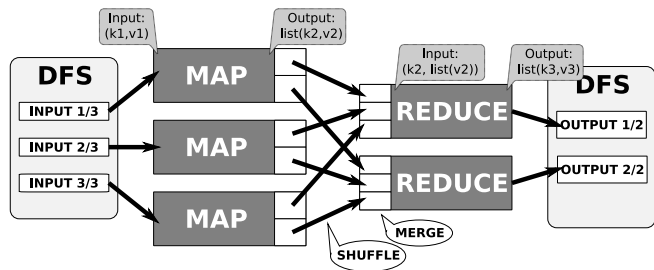

Figure 1: Data flow in a MapReduce computation.

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.

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

---

[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.

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.

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
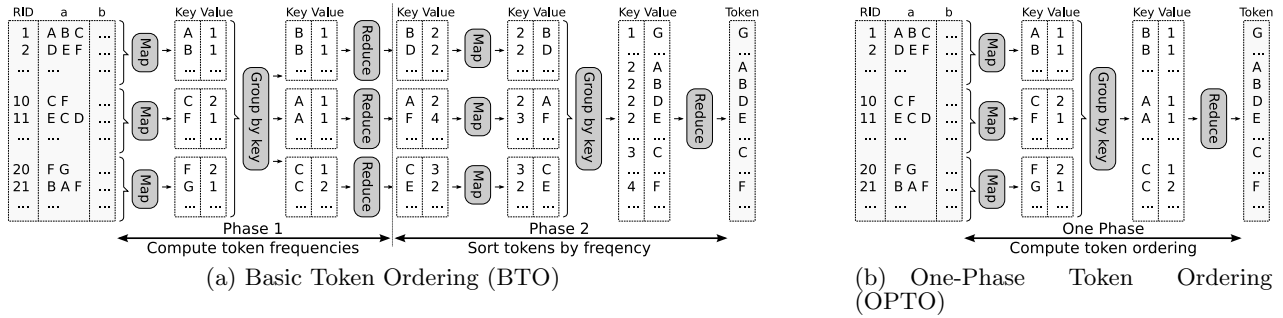
(a) Basic Token Ordering (BTO)

(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.

(a) Basic Kernel (BK) using individual tokens for routing

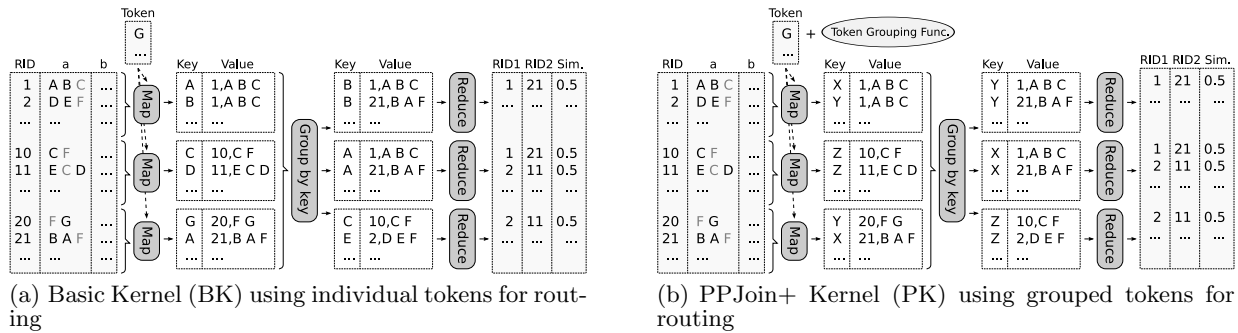(b) PPJoin+ Kernel (PK) using grouped tokens for routing

Figure 3: Example data flow of Stage 2. (Kernel for a self-join on attribute "a".)

3.3 Stage 3: Record Join

In the final stage of our algorithm, we use the RID pairs generated in the second stage to join their records. We propose two approaches for this stage. The main idea is to first fill in the record information for each half of the pair and then use the two halves to build the full record pair. The two approaches differ in the way the list of RID pairs is provided as input. In the first approach, called Basic Record Join ("BRJ"), the list of RID pairs is treated as a normal MapReduce input, and is provided as input to the `map` functions. In the second approach, called One-Phase Record Join ("OPRJ"), the list is broadcast to all the maps and loaded before reading the input data. Duplicate RID pairs from the previous stage are eliminated in this stage.

3.3.1 Basic Record Join (BRJ)

The Basic Record Join algorithm uses two MapReduce phases. In the first phase, the algorithm fills in the record information for each half of each pair. In the second phase, it brings together the half-filled pairs. The `map` function in the first phase gets as input both the set of original records and the RID pairs from the second stage. (The function can differentiate between the two types of inputs by looking at the input file name.) For each original record, the function outputs a (`RID, record`) pair. For each RID pair, it outputs two (`key, value`) pairs. The first pair uses the first RID as its `key`, while the second pair uses the second RID as its `key`. Both pairs have the entire RID pair and their similarity as their value. Figure 4 shows the data flow for our example dataset. In the figure, the first two mappers take records as their input, while the third mapper takes RID pairs as its input. (Mappers do not span across files.) For the RID pair $(2, 11)$, the mapper outputs two pairs, one with key 2 and one with key 11.

The `reduce` function of the first phase then receives a list of values containing exactly one record and other RID pairs. For each RID pair, the function outputs a (`key, value`) pair, where the `key` is the RID pair, and the value is the record itself and the similarity of the RID pair. Continuing our example in Figure 4, for key 2, the first reducer gets the record with RID 2 and one RID pair $(2, 11)$, and outputs one (`key, value`) pair with the RID pair $(2, 11)$ as the key.

The second phase uses an identity map that directly outputs its input. The `reduce` function therefore gets as input, for each `key` (which is a RID pair), a list of values containing exactly two elements. Each element consists of a record and a common similarity value. The reducer forms a pair of the two records, appends their similarity, and outputs the constructed pair. In Figure 4, the output of the second set of mappers contains two (`key, value`) pairs with the RID pair $(1, 21)$ as the key, one containing record 1 and the other containing record 21. They are grouped in a reducer that outputs the pair of records $(1, 21)$.

3.3.2 One-Phase Record Join (OPRJ)

The second approach to record join uses only one MapReduce phase. Instead of sending the RID pairs through the MapReduce pipeline to group them with the records in the reduce phase (as we do in the BRJ approach), we broadcast and load the RID pairs at each `map` function before the input data is consumed by the function. The `map` function then gets the original records as input. For each record, the function outputs as many (`key, value`) pairs as the number of RID pairs containing the RID of the current record. The output `key` is the RID pair. Essentially, the output of the `map` function is the same as the output of the `reduce` function in the first phase of the BRJ algorithm. The idea of joining the data in the mappers was also used in [10] for the case of equi-joins. The `reduce` function is the same as the `reduce` function in the second phase of the BRJ algorithm. Figure 5 shows the data flow for our example dataset. In the figure, the first mapper gets as input the record with RID 1 and outputs one (`key, value`) pair, where the key is the RID pair $(1, 21)$ and the value is record 1. On the other hand, the third mapper outputs a pair with the same key, and the value is the record 21. The two pairs get grouped in the second reducer, where the pair of records $(1, 21)$ is output.

4. R-S JOIN CASE

In Section 3 we described how to compute set-similarity self-joins using the MapReduce framework. In this section we present our solutions for the set-similarity R-S joins case. We highlight the differences between the two cases and discuss an optimization for carefully controlling memory usage in the second stage.

The main differences between the two join cases are in the second and the third stages where we have records from two datasets as the input. Dealing with the binary join operator is challenging in MapReduce, as the framework was designed to only accept a single input stream. As discussed in [10, 21], in order to differentiate between two different input streams in MapReduce, we extend the key of the (`key, value`) pairs so that it includes a relation *tag* for each record. We also
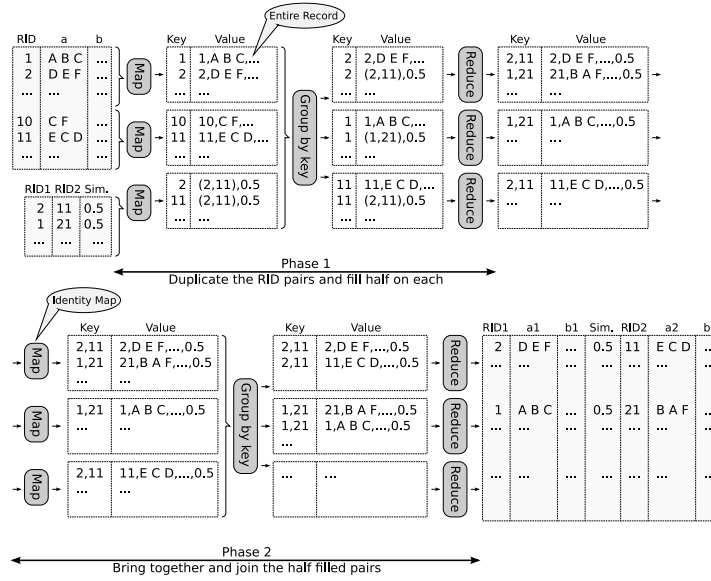
Figure 4: Example data flow of Stage 3 using Basic Record Join (BRJ) for a self-join case. "a1" and "a2" correspond to the original attribute "a", while "b1" and "b2" correspond to attribute "b".
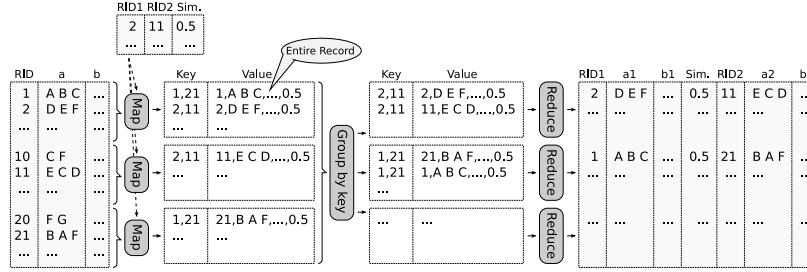

Figure 5: Example data flow of Stage 3 using One-Phase Record Join (OPRJ) for a self-join case. "a1" and "a2" correspond to the original attribute "a" while, "b1" and "b2" correspond to attribute "b".

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 in-
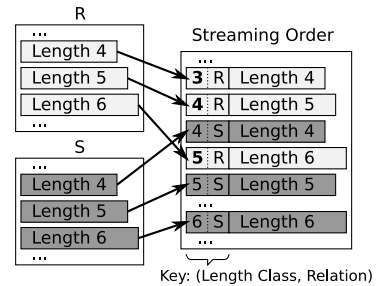

Figure 6: Example of the order in which records need to arrive at the reducer in the PK kernel of the R-S join case, assuming that for each length, l, the lower-bound is $l-1$ and the upper-bound is $l+1$.

creasing 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-

fore we stream a particular record projection from S, we need to have seen all the record projections from R with a length smaller than or equal to the *upper-bound* length of the record from S. We force this arrival order by extending the keys with a *length class* assigned in the following way. For records from S, the length class is their actual length. For records from R, the length class is the *lower-bound* length corresponding to their length. Figure 6 shows an example of the order in which records will arrive at the reducer, assuming that for each length l, the lower-bound is $l - 1$ and the upper-bound is $l + 1$. In the figure, the records from R with length 5 get length class 4 and are streamed to the reducer before those records from S with lengths between $[4, 6]$.

Stage 3: Record Join. For the BRJ algorithm the mappers first tag their outputs with the relation name. Then, the reducers get a record and their corresponding RID pairs grouped by relation and output half-filled pairs tagged with the relation name. Finally, the second-phase reducers use the relation name to build record pairs having the record form R first and the record form S second. In the OPRJ algorithm, for each input record from R, the mappers output as many (`key,value`) pairs as the number of RID pairs containing the record's RID in the R column (and similar for S records). For each pair, the key is the RID pair plus the relation name. The reducers proceed as do the second-phase reducers for the BRJ algorithm.

5. HANDLING INSUFFICIENT MEMORY

As we saw in Section 3.2, reducers in the second stage receive as input a list of record projections to be verified. In the BK approach, the entire list of projections needs to fit in memory. (For the R-S join case, only the projections of one relation must fit.) In the PK approach, because we are exploiting the length filter, only the fragment corresponding to a certain length range needs to fit in memory. (For the R-S join case, only the fragment belonging to only one relation must fit.) It is worth noting that we already decreased the amount of memory needed by grouping the records on the infrequent prefix tokens. Moreover, we can exploit the length filter even in the BK algorithm, by using the length filter as a secondary record-routing criterion. In this way, records are routed on token-length-based keys. The additional routing criterion partitions the data even further, decreasing the amount of data that needs to fit in memory. This technique can be generalized and additional filters can be appended to the routing criteria. In this section we present two extensions of our algorithms for the case where there are just no more filters to be used but the data still does not fit in memory. The challenge is how to compute the cross product of a list of elements in MapReduce. We sub-partition the data so that each block fits in memory and propose two approaches for processing the blocks. First we look how the two methods work in the self-join case and then discuss the differences for the R-S join case.

Map-Based Block Processing. In this approach, the `map` function replicates the blocks and interleaves them in the order they will be processed by the reducer. For each block sent by the `map` function, the reducer either loads the block in memory or streams the block against a block already loaded in memory. Figure 7(a) shows an example of how blocks are processed in the reducer, in which the data is sub-partitioned into three blocks A, B, and C. In the first step, the first block, A, is loaded into memory and self-joined.

(a) Map-based block processing
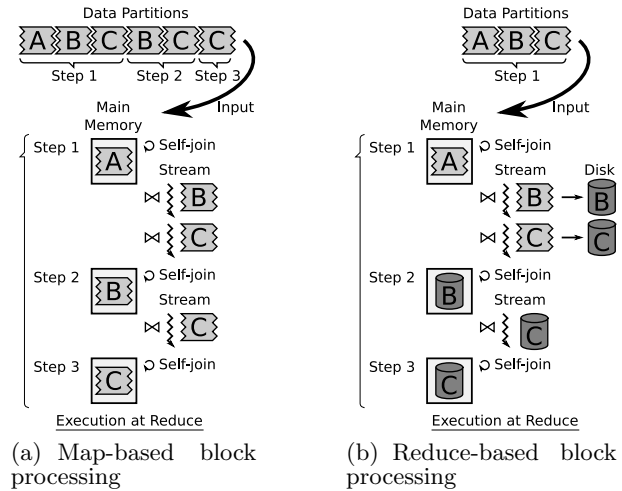
(b) Reduce-based block processing

Figure 7: Data flow in the reducer for two block processing approaches.

After that, the next two blocks, B and C, are read from the input stream and joined with A. Finally, A is discarded from memory and the process continues for blocks B and C. In order to achieve the interleaving and replications of the blocks, the `map` function does the following. For each (`key, value`) output pair, the function determines the pair's block, and outputs the pair as many times as the block needs to be replicated. Every copy is output with a different composite key, which includes its position in the stream, so that after sorting the pairs, they are in the right blocks and the blocks are in the right order.

Reduce-Based Block Processing. In this approach, the `map` function sends each block exactly once. On the other hand, the `reduce` function needs to store all the blocks except the first one on its local disk, and reload the blocks later from the disk for joining. Figure 7(b) shows an example of how blocks are processed in the reducer for the same three blocks A, B, and C. In the first step, block A is loaded into memory and self-joined. After that, the next two blocks, B and C, are read from the input stream and joined with A and also stored on the local disk. In the second step, A is discarded from memory and block B is read from disk and self-joined. Then, block C is read from the disk and joined with B. The process ends with reading C from disk and self-joining it.

Handling R-S Joins. In the R-S join case the `reduce` function needs to deal with a partition from R that does not fit in memory, while it streams a partition coming from S. We only need to sub-partition the R partition. The `reduce` function loads one block from R into memory and streams the entire S partition against it. In the map-based block processing approach, the blocks from R are interleaved with multiple copies of the S partition. In the reduce-based block processing approach all the R blocks (except the first one) and the entire S partition are stored and read from the local disk later.

6. EXPERIMENTAL EVALUATION

In this section we describe the performance evaluation of the proposed algorithms. To understand the performance

of parallel algorithms we need to measure absolute running time as well as relative speedup and scaleup [8].

We ran experiments on a 10-node IBM x3650 cluster. Each node had one Intel Xeon processor E5520 2.26GHz with four cores, 12GB of RAM, and four 300GB hard disks. Thus the cluster consists of 40 cores and 40 disks. We used an extra node for running the master daemons to manage the Hadoop jobs and the Hadoop distributed file system. On each node, we installed the Ubuntu 9.04, 64-bit, server edition operating system, Java 1.6 with a 64-bit server JVM, and Hadoop 0.20.1. In order to maximize the parallelism and minimize the running time, we made the following changes to the default Hadoop configuration: we set the block size of the distributed file system to 128MB, allocated 1GB of virtual memory to each daemon and 2.5GB of virtual memory to each map/reduce task, ran four map and four reduce tasks in parallel on each node, set the replication factor to 1, and disabled the speculative task execution feature.

We used the following two datasets and increased their sizes as needed:

- **DBLP**[3] It had approximately 1.2M publications. We preprocessed the original XML file by removing the tags, and output one line per publication that contained a unique integer (RID), a title, a list of authors, and the rest of the content (publication date, publication journal or conference, and publication medium). The average length of a record was 259 bytes. A copy of the entire dataset has around 300MB before we increased its size. It is worth noting that we did not *clean* the records before running our algorithms, i.e., we did not remove punctuations or change the letter cases. We did the cleaning inside our algorithms.

- **CITESEERX**[4] It had about 1.3M publications. We preprocessed the original XML file in the same way as for the DBLP dataset. Each publication included an abstract and URLs to its references. The average length of a record was 1374 bytes, and the size of one copy of the entire dataset is around 1.8GB.

Increasing Dataset Sizes. To evaluate our parallel set-similarity join algorithms on large datasets, we increased each dataset while maintaining its set-similarity join properties. We maintained a roughly constant token dictionary, and wanted the cardinality of join results to increase linearly with the increase of the dataset. Increasing the data size by duplicating its original records would only preserve the token dictionary-size, but would blow up the size of the join result. To achieve the goal, we increased the size of each dataset by generating new records as follows. We first computed the frequencies of the tokens appearing in the title and the list of authors in the original dataset, and sorted the tokens in their increasing order of frequencies. For each record in the original dataset, we created a new record by replacing each token in the title or the list of authors with the token after it in the token order. For example, if the token order is (`A, B, C, D, E, F`) and the original record is "`B A C E`", then the new record is "`C B D F`." We evaluated the cardinality of the join result after increasing the dataset in this manner, and noticed it indeed increased linearly with the increase of the dataset size.

We increased the size of each dataset 5 to 25 times. We re-

---

[3]http://dblp.uni-trier.de/xml/dblp.xml.gz
[4]http://citeseerx.ist.psu.edu/about/metadata

fer to the increased datasets as "DBLP$\times n$" or "CITESEERX $\times n$", where $n \in [5, 25]$ and represents the increase factor. For example, "DBLP$\times 5$" represents the DBLP dataset increased five times.

Before starting each experiment we balanced its input datasets across the ten nodes in HDFS and the four hard drives of each node in the following way. We formatted the distributed file system before each experiment. We created an identity MapReduce job with as many reducers running in parallel as the number of hard disks in the cluster. We exploited the fact that reducers write their output data to the local node and also the fact that Hadoop chooses the disk to write the data using a Round-Robin order.

For all the experiments, we tokenized the data by word. We used the concatenation of the paper title and the list of authors as the join attribute, and used the Jaccard similarity function with a similarity threshold of 0.80. The 0.80 threshold is usually the lower bound on the similarity threshold used in the literature [3, 6, 29], and higher similarity thresholds decreased the running time. A more complete set of figures for the experiments is contained in [26]. The source code is available at `http://asterix.ics.uci.edu/`.

6.1 Self-Join Performance

We did a self-join on the DBLP$\times n$ datasets, where $n \in [5, 25]$. Figure 8 shows the total running time of the three stages on the 10-node cluster for different dataset sizes (represented by the factor n). The running time consisted of the times of the three stages: token ordering, kernel, and record join. For each dataset size, we used three combinations of the approaches of the stages. For example, "1-BTO" means we use BTO in the first stage. The second stage is the most expensive step, and its time increased the fastest with the increase of the dataset size. The best algorithm is BTO-PK-OPRJ, i.e., with BTO for the first stage, PK for the second stage, and OPRJ for the third stage. This combination could self-join 25 times the original DBLP dataset in around 650 seconds.

6.1.1 Self-Join Speedup

In order to evaluate the speedup of the approaches, we fixed the dataset size and varied the cluster size. Figure 9 shows the running time for self-joining the DBLP$\times 10$ dataset on clusters of 2 to 10 nodes. We used the same three combinations for the three stages. For each approach, we also show its ideal speedup curve (with a thin black line). For instance, if the cluster has twice as many nodes and the data size does not change, the approach should be twice as fast. In Figure 10 we show the same numbers, but plotted on a "relative scale". That is, for each cluster size, we plot the ratio between the running time for the smallest cluster size and the running time of the current cluster size. For example, for the 10-node cluster, we plot the ratio between the running time on the 2-node cluster and the running time on the 10-node cluster. We can see that all three combinations have similar speedup curves, but none of them speed up linearly. In all the settings the BTO-PK-OPRJ combination is the fastest (Figure 9). In the following experiments we looked at the speedup characteristics for each of the three stages in order to understand the overall speedup. Table 1 shows the running time for each stage in the self-join.

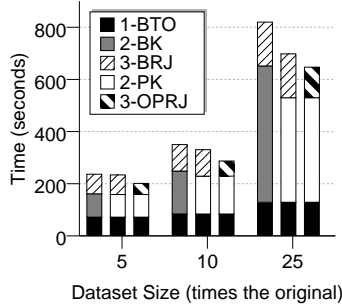**Stage 1: Token Ordering.** We can see that the OPTO approach was the fastest for the settings of 2 nodes and 4

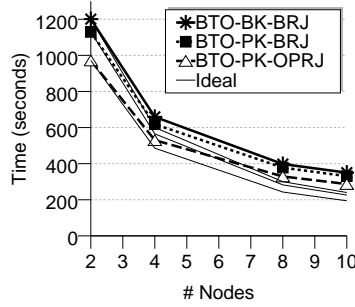**Figure 8: Running time for self-joining DBLP×n datasets (where $n \in [5, 25]$) on a 10-node cluster.**

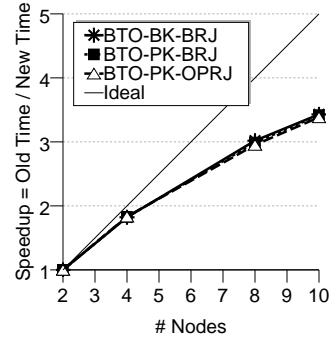**Figure 9: Running time for self-joining the DBLP×10 dataset on different cluster sizes.**

Figure 10: Relative running time for self-joining the DBLP×10 data set on different cluster sizes.

Stage	Alg.	# Nodes			
		2	4	8	10
1	BTO	191.98	125.51	91.85	84.02
	OPTO	175.39	115.36	94.82	92.80
2	BK	753.39	371.08	198.70	164.57
	PK	682.51	330.47	178.88	145.01
3	BRJ	255.35	162.53	107.28	101.54
	OPRJ	97.11	74.32	58.35	58.11

Table 1: Running time (seconds) of each stage for self-joining the DBLP×10 dataset on different cluster sizes

nodes. For the settings of 8 nodes and 10 nodes, the BTO approach became the fastest. Their limited speedup was due to two main reasons. (1) As the number of nodes increased, the amount of input data fed to each combiner decreased. As the number of nodes increased, more data was sent through the network and more data got merged and reduced. (A similar pattern was observed in [10] for the case of general aggregations.) (2) The final token ordering was produced by only one reducer, and this step's cost remained constant as the number of nodes increased. The speedup of the OPTO approach was even worse since as the number of nodes increased, the extra data that was sent through the network had to be aggregated at only one reducer. Because BTO was the fastest for settings of 8 nodes and 10 nodes, and it sped up better than OPTO, we only considered BTO for the end-to-end combinations.

Stage 2: Kernel. For the PK approach, an important factor affecting the running time is the number of token groups. We evaluated the running time for different numbers of groups. We observed that the best performance was achieved when there was one group per token. The reason was that the reduce function could benefit from the grouping conducted "for free" by the MapReduce framework. If groups had more than one token, the framework spends the same amount of time on grouping, but the reducer benefits less. Both approaches had an almost perfect speedup. Moreover, in all the settings, the PK approach was the fastest.

Stage 3: Record Join. The OPRJ approach was always faster than the BRJ approach. The main reason for the poor speedup of the BRJ approach was due to skew in the RID

pairs that join, which affected the workload balance. For analysis purposes, we computed the frequency of each RID appearing in at least one RID pair. On the average an RID appeared on 3.74 RID pairs, with a standard deviation of 14.85 and a maximum of 187. Additionally, we counted how many records were processed by each reduce instance. The minimum number of records processed in the 10-nodes case was 81,662 and the maximum was 90,560, with an average of 87,166.55 and a standard deviation of 2,519.30. No matter how many nodes we added to the cluster, a single RID could not be processed by more than one reduce instance, and all the reducers had to wait for the slowest one to finish.

The speedup of the OPRJ approach was limited because in the OPRJ approach, the list of RID pairs that joined was broadcast to all the maps where they must be loaded in memory and indexed. The elapsed time required for this remained constant as the number of nodes increased. Additional information about the total amount of data sent between map and reduce for each stage is included in [26].
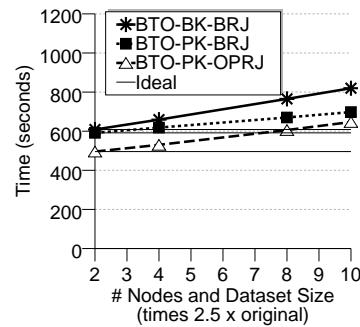
6.1.2 Self-Join Scaleup

Figure 11: Running time for self-joining the DBLP×n dataset (where $n \in [5, 25]$) increased proportionally with the increase of the cluster size.

In order to evaluate the scaleup of the proposed approaches we increased the dataset size and the cluster size together by the same factor. A perfect scaleup could be achieved if the running time remained constant. Figure 11 shows the running time for self-joining the DBLP dataset, increased

from 5 to 25 times, on a cluster with 2 to 10 nodes, respectively. We can see that the fastest combined algorithm was BTO-PK-OPRJ. We can also see that all three combinations scaled up well. BTO-PK-BRJ had the best scaleup. In the following, we look at the scaleup characteristics of each stage. Table 2 shows the running time (in seconds) for each of the self-join stages.

Stage	Alg.	# Nodes/Dataset Size			
		2/x5	4/x10	8/x20	10/x25
1	BTO	124.05	125.51	127.73	128.84
	OPTO	107.21	115.36	136.75	149.40
2	BK	328.26	371.08	470.84	522.88
	PK	311.19	330.47	375.72	401.03
3	BRJ	156.33	162.53	166.66	168.08
	OPRJ	60.61	74.32	102.44	117.15

Table 2: Running time (seconds) of each stage for self-joining the DBLP×n dataset ($n \in [5, 25]$) increased proportionally with the increase of the cluster size.

Stage 1: Token Ordering. We can see in Table 2 that the BTO approach scaled up almost perfectly, while the OPTO approach did not scale up as well. Moreover, the OPTO approach became more expensive than the BTO approach as the number of nodes increased. The reason why the OPTO approach did not scale up as well was because it used only one `reduce` instance to aggregate the token counts (instead of using multiple `reduce` functions as in the BTO case). Both approaches used a single `reduce` to sort the tokens by frequency. As the data increased, the time needed to finish the one `reduce` function increased.

Stage 2: Kernel. We can see that the PK approach was always faster and scaled up better than the BK approach. To understand why the BK approach did not scale up well, let us take a look at the complexity of the reducers. The `reduce` function was called for each prefix token. For each token, the `reduce` function received a list of record projections, and had to verify the self-cross-product of this list. Moreover, a reducer processed a certain number of tokens. Thus, if the length of the record projections list is n and the number of tokens that each reducer has to process is m, the complexity of each reducer is $O(m \cdot n^2)$. As the dataset increases, the number of unique tokens remains constant, but the number of records having a particular prefix token increased by the same factor as the dataset size. Thus, when the dataset is increased t times the length of the record projections list increases t times. Moreover, as the number of nodes increases t times, the number of tokens that each reducer has to process decreases t times. Thus, the complexity of each reducer becomes $O\big(m/t \cdot (n \cdot t)^2\big) = O(t \cdot m \cdot n^2)$. Despite the fact the `reduce` function had a running time that grew proportional with the dataset size, the scaleup of the BK approach was still acceptable because the `map` function scaled up well. In the case of PK, the quadratic increase when the data linearly increased was alleviated because an index was used to decide which pairs are verified.

Stage 3: Record Join. We can see that the BRJ approach had an almost perfect scaleup, while the OPRJ approach did not scale up well. For our 10-node cluster, the OPRJ approach was faster than the BRJ approach, but OPRJ could become slower as the number of nodes and data

size increased. The OPRJ approach did not scale up well since the list of RID pairs that needed to be loaded and indexed by each `map` function increased linearly with the size of the dataset.

6.1.3   Self-Join Summary

We have the following observations:

- For the first stage, BTO was the best choice.

- For the second stage, PK was the best choice.

- For the third stage, the best choice depends on the amount of data and the size of the cluster. In our experiments, OPRJ was somewhat faster, but the cost of loading the similar-RID pairs in memory was constant as the the cluster size increased, and the cost increased as the data size increased. For these reasons, we recommend BRJ as a good alternative.

- The three combinations had similar speedups, but the best scaleup was achieved by BTO-PK-BRJ.

- Our algorithms distributed the data well in the first and the second stages. For the third stage, the algorithms were affected by the fact that some records produced more join results that others, and the amount of work to be done was not well balanced across nodes. This skew heavily depends on the characteristics of the data and we plan to study this issue in future work.

6.2   R-S Join Performance

To evaluate the performance of the algorithms for the R-S-join case, we did a join between the DBLP and the CITE-SEERX datasets. We increased both datasets at the same time by a factor between 5 and 25. Figure 12 shows the running time for the join on a 10-node cluster. We used the same combinations for each stage as in the self-join case. Moreover, the first stage was identical to the first stage of the self-join case, as this stage was run on only one of the datasets, in this case, DBLP. The running time for the second stage (kernel) increased the fastest compared with the other stages, but for the 5 and 10 dataset-increase factors, the third stage (record join) became the most expensive. The main reason for this behavior, compared to the self-join case, was that this stage had to scan two datasets instead of one, and the record length of the CITESEERX dataset was much larger than the record length of the DBLP dataset. For the 25 dataset-increase factor, the OPRJ approach ran out of memory when it loaded the list of RID pairs, making BRJ the only option.

6.2.1   R-S Join Speedup

As in the self-join case, we evaluated the speedup of the algorithms by keeping the dataset size constant and increasing the number of nodes in the cluster. Figure 13 shows the running times for the same three combinations of approaches. We can see that the BTO-PK-OPRJ combination was initially the fastest, but for the 10-node cluster, it became slightly slower than the BTO-BK-BRJ and BTO-PK-BRJ combinations. Moreover, BTO-BK-BRJ and BTO-PK-BRJ sped up better than BTO-PK-OPRJ.

To better understand the speedup behavior, we looked at each individual stage. The first stage performance was identical to the first stage in the self-join case. For the second stage we noticed a similar speedup (almost perfect) as for the self-join case. Regarding the third stage, we noticed
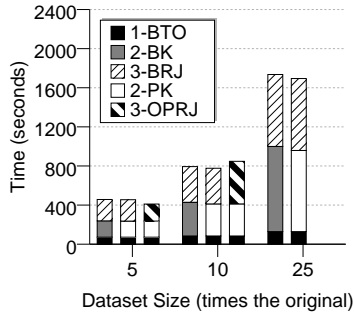
Figure 12: Running time for joining the DBLP×n and the CITESEERX×n datasets (where $n \in [5, 25]$) on a 10-node cluster.
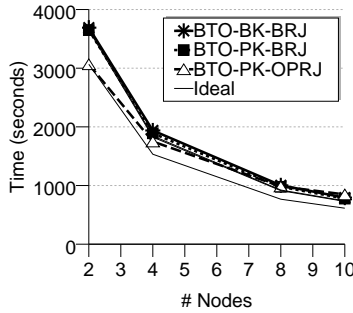
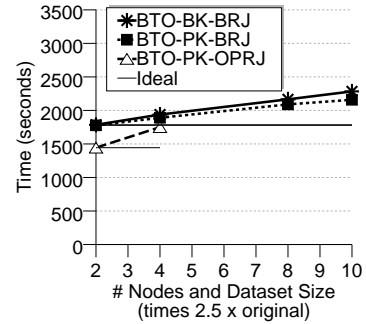**Figure 13: Running time for joining the DBLP×10 and the CITESEERX×10 datasets on different cluster sizes.**

Figure 14: Running time for joining the DBLP×n and the CITESEERX×n datasets (where $n \in [5, 25]$) increased proportionally with the cluster size.

that the OPRJ approach was initially the fastest (for the 2 and 4 node case), but it eventually became slower than the BRJ approach. Additionally, the BRJ approach sped up better than the OPRJ approach. The poor performance of the OPRJ approach was due to the fact that all the map instances had to load the list of RID pairs that join.

6.2.2   R-S Join Scaleup

We also evaluated the scaleup of the R-S join approaches. The evaluation was similar to the one done in the self-join case. In Figure 14 we plot the running time of three combinations for the three stages as we increased the dataset size and the cluster size by the same factor. We can see that BTO-BK-BRJ and BTO-PK-BRJ scaled up well. The BTO-PK-BRJ combination scaled up the best. BTO-PK-OPRJ ran out of memory in the third stage for the case where the datasets were increased 8 times the original size. The third stage ran out of memory when it tried to load in memory the list of RID pairs that join. Before running out of memory, though, BTO-PK-OPRJ was the fastest.

To better understand the behavior of our approaches, we again analyzed the scaleup of each individual stage. The first stage performance was identical with its counter-part in the self-join case. Additionally, the second stage had a similar scaleup performance as its counterpart in the self-join case. Regarding the third stage, we observed that the BRJ approach scaled up well. We also observed that even before running out of memory, the OPRJ approach did not scale up well, but for the case where it did not run our of memory, it was faster than the BRJ approach.

6.2.3   R-S Join Summary

We have the following observations:

- The recommendations for the best choice from the self-join case also hold for the R-S join case.

- The third stage of the join became a significant part of the execution due to the increased amount of data.

- The three algorithm combinations preserved their speedup and scaleup characteristics as for the self-join case.

- We also observed the same data distribution characteristics as for the self-join case.

- For both self-join and R-S join cases, we recommend BTO-PK-BRJ as a robust and scalable method.

7.   RELATED WORK

Set-similarity joins on a single machine have been widely studied in the literature [23, 6, 3, 4, 29]. Inverted-list-based algorithms for finding pairs of strings that share a certain number of tokens in common have been proposed in [23]. Later work has proposed various filters that help decrease the number of pairs that need to be verified. The prefix filter has been proposed in [6]. The length filter has been studied in [3, 4]. Two other filters, namely the positional filter and the suffix filter, were proposed in [29]. In particular, for edit distance, two more filters based on mismatch have been proposed in [28]. Instead of directly using the tokens in the strings, the approach in [3] generates a set of signatures based on the tokens in the string and relies on the fact that similar strings need to have a common signature. A different way of formulating set-similarity join problem is to return partial answers, by using the idea of locality sensitive hashing [12]. It is worth noting most of work deals with values already projected on the similarity attribute and produces only the list of RIDs that join. Computing such a list is the goal of our second stage and most algorithms could successfully replace PPJoin+ in our second stage. To the best of our knowledge, there is no previous work on parallel set-similarity joins.

Parallel join algorithms for large datasets had been widely studied since the early 1980's (e.g., [19, 24]). Moreover, the ideas presented in Section 5 bear resemblance with the bucket-size-tuning ideas presented in [18]. Data partition and replication techniques have been studied in [9] for the problem of numeric band joins.

The MapReduce paradigm was initially presented in [7]. Since then, it has gained a lot of attention in academia [30, 21, 10] and industry [2, 16]. In [30] the authors proposed extending the interface with a new function called "merge" in order to facilitate joins. A comparison of the MapReduce paradigm with parallel DBMS has been done in [21]. Higher-level languages on top of MapReduce have been proposed in [2, 16, 10]. All these languages could benefit from the addition of a set-similarity join operator based on the

techniques proposed here. In the context of JAQL [16] a tutorial on fuzzy joins was presented in [17].

8. CONCLUSIONS

In this paper we studied the problem of answering set-similarity join queries in parallel using the MapReduce framework. We proposed a three-stage approach and explored several solutions for each stage. We showed how to partition the data across nodes in order to balance the workload and minimize the need for replication. We discussed ways to efficiently deal with partitioning, replication, and multiple inputs by exploiting the characteristics of the MapReduce framework. We also described how to control the amount of data that needs to be kept in memory during join by exploiting the data properties. We studied both self-joins and R-S joins, end-to-end, by starting from complete records and producing complete record pairs. Moreover, we discussed strategies for dealing with extreme situations where, even after the data is partitioned to the finest granularity, the amount of data that needs to be in the main memory of one node is too large to fit. Given our proposed algorithms, we implemented them in Hadoop and analyzed their performance characteristics on real datasets (synthetically increased).

9. REFERENCES

[1] Apache Hadoop. `http://hadoop.apache.org`.

[2] Apache Hive. `http://hadoop.apache.org/hive`.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[5] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

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

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

[9] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, pages 443–452, 1991.

[10] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of MapReduce: the Pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[11] Genbank. `http://www.ncbi.nlm.nih.gov/Genbank`.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[13] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[14] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[15] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *JASIST*, 54(3):203–215, 2003.

[16] Jaql. `http://www.jaql.org`.

[17] Jaql - Fuzzy join tutorial. `http://code.google.com/p/jaql/wiki/fuzzyJoinTutorial`.

[18] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, pages 210–221, 1990.

[19] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[20] A. Metwally, D. Agrawal, and A. E. Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *WWW*, pages 241–250, 2007.

[21] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[22] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *WWW*, pages 377–386, 2006.

[23] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[24] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD Conference*, pages 110–121, 1989.

[25] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. In *KDD*, pages 678–684, 2005.

[26] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. Technical report, Department of Computer Science, UC Irvine, March 2010. `http://asterix.ics.uci.edu`.

[27] Web 1t 5-gram version 1. `http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13`.

[28] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. In *VLDB*, 2008.

[29] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[30] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. P. Jr. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD Conference*, pages 1029–1040, 2007.