# Fuzzy Joins in MapReduce: Edit and Jaccard Distance

Ben Kimmett
University of Victoria, BC, Canada
blk@uvic.ca

Alex Thomo
University of Victoria, BC, Canada
thomo@uvic.ca

Venkatesh Srinivasan
University of Victoria, BC, Canada
srinivas@uvic.ca

***Abstract*—In ICDE'12, Afrati, Das Sarma, Menestrina, Parameswaran and Ullman proposed similarity join algorithms for MapReduce. In this paper, we evaluate and extend their research, testing their proposed algorithms using edit distance and Jaccard similarity. We provide details of adaptations needed to implement their algorithms based on these similarity measures. We conduct an extensive experimental study on large datasets and evaluate the algorithms across several dimensions that define the performance profile in MapReduce.**

***Keywords*—*Fuzzy Join, Similarity Join, MapReduce, Entity Resolution, Record Linkage***

## I. Introduction

Fuzzy join (or similarity join) is a binary operation that takes two sets of elements as input and computes a set of similar element-pairs as output. This is different from exact join where records are matched based on the equality of some fields (cf. [3], [15]). Fuzzy join is very useful in a multitude of applications, such as entity resolution [7], [12], [16], [18] (finding records that refer to the same entity or person), recommender systems [1], [4], [6], [28] (finding similar users in terms of items they have rated), and clustering [5], [17], [19], [20] (grouping items based on similarity). The fuzzy join problem has attracted significant attention from the research community (cf. [2], [13], [24], [25], [26], [27]).

In this paper, we focus on the computation of fuzzy join in MapReduce. More specifically, we report an experimental evaluation of some MapReduce algorithms proposed by Afrati, Das Sarma, Menestrina, Parameswaran and Ullman in [2]. These algorithms have been described and analyzed in [2] from a theoretical point of view only. In [14], we provide an experimental evaluation of the fuzzy join algorithms of [2] using simple Hamming distance. In the current paper, we continue this line of work and consider the more challenging cases of edit and Jaccard distances.

In [2], the authors argue that there is a tradeoff between communication cost and processing cost in the distributed MapReduce setting, and that there is a skyline of the proposed algorithms; i.e. none dominates another. In [14], we showed via experiments that, from a practical point of view, some algorithms are almost always preferable to others. These algorithms were *Splitting* and *Naive*, with the former being faster than the latter when Hamming distance is used. In the current paper, we extend the work in [14] to consider edit and Jaccard distances.

While Naive can be easily extended to handle edit and Jaccard distances, Splitting is more challenging. We describe our algorithmic contributions with respect to Splitting, then embark into the experimental setup and evaluation. We show that for edit and Jaccard distances, both Naive and Splitting continue to be able to handle large datasets in terms of communication and processing cost in MapReduce. However, in contrast to our results for Hamming distance, Naive (in spite of its name) is now more efficient than Splitting.

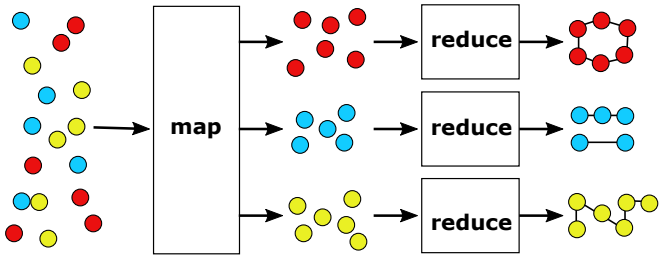


Fig. 1: Graphic representation of the MapReduce process. Input items (circles) are categorized in the map phase, then groups of items are processed in the reduce phase.

Our contributions and results can be summarized as follows. First, we explain the algorithmic features of Naive and Splitting algorithms in MapReduce, and present an algorithmic engineering of the Splitting algorithm for Jaccard distance. Next, we perform extensive experiments for Naive and Splitting using edit and Jaccard distance on large datasets, such as genome sequences and movie ratings. Finally, we provide insights on the practical performance of the two algorithms considered across five dimensions: communication cost, mapper computation cost, shuffle time, reducer computation cost, and total time.

## II. MapReduce

Computations in MapReduce have two primary phases: a map phase, and a reduce phase (see Figure 1). In the map phase, input records are read, and grouped by some characteristic of the record (this varies depending on the algorithm used). The mapper emits key-value pairs, where the key is used to group the items (that is, every item a mapper groups together will have the same key when it is output).

The reduce phase takes the input from the map phase. The key to MapReduce's model is that every key-value pair with the same key will end up at the same reducer instance, and will be able to be processed as a group by that reducer. Once the reducer receives the groups of data, it performs a

processing step on each group (the nature of this step also varies depending on the algorithm used). It then outputs the result of the processing step.

MapReduce also has a 'shuffle phase', which consists of transferring data between the mappers and reducers. It is not shown in Figure 1, but we track it separately in our results graphs.

The bulk of [2] focuses on measuring similarity using Hamming distance, though the paper does mention how some of their algorithms could be applied to edit distance and Jaccard distance. In this paper, we examine two of the algorithms in [2], using edit and Jaccard distance as our similarity measures.

While [2] proposed other algorithms, such as the Ball-Hashing algorithms, the poor performance of these in our previous paper [14] led us to decide not to consider these algorithms in this paper. The bottleneck of the Ball-Hashing algorithms is the amount of intermediate data they generate; this feature remains the same for any distance measure, and may in fact become worse when used with varied text strings or sets (for edit or Jaccard distance) as opposed to the binary bit-strings used to test Hamming distance algorithms.

The algorithms we examine are *Naive*, which compares every element in a set with every other using a rather insightful triangular arrangement of reducers, and *Splitting*, which divides each element in a set in chunks, then checks to see if elements with matching chunks are similar. Determining how to create meaningful chunks for each distance measure is not always easy, and our algorithmic contribution is in showing how to do that for Jaccard distance.

While Splitting was the algorithm with the best communication cost and fastest processing time in our previous work [14], in which we analyzed the Hamming distance algorithms of [2], we show that it does not do as well on edit or Jaccard distance due to how this algorithm handles input.

## III. Algorithms and Implementation

**Input Formats** The distance measures covered in this paper, Jaccard and edit distance, place different requirements on the input that can be given to them. This influences the format of the input in our implementations of these algorithms.

For edit distance, each item in the input is a text string (e.g. a substring of a genome sequence). Substitutions are allowed. For Jaccard distance, each item is a set (e.g. the set of products a user has rated in an e-commerce site). Our implementation of Jaccard distance uses input consisting of pre-sorted sets of integers; each set is a single item in the input. Substitutions are not allowed but, for purposes of implementation, can be represented as an operation equivalent to the cost of one deletion plus one addition.

**Relation of Edit Distance to Jaccard Distance** If input sets are sorted, the Jaccard similarity of two sets can be computed by running an edit distance comparator on them, disallowing substitutions. If a similarity of $J$ is desired, the maximum number of edits that is permissible to turn some set $A$ into another set $B$ is as follows:

$(|A| + |B|) * (1 - \frac{2J}{J+1})$.

To explain this, we start by noting that Jaccard similarity of sets $A$ and $B$ is defined as $\frac{|A \cap B|}{|A \cup B|}$. The size of the union can be defined as $|A|+|B|-m$, where $m$ is the number of elements in both $A$ and $B$ (matching elements). This works because each element in A and B is counted once if we sum the magnitude of each. However, this double-counts matching elements, so subtracting $m$ gives a correct count of the magnitude of the union. Additionally, $m$ is the magnitude of the intersection, $|A \cap B|$. However, while $|A|$ and $|B|$ are known, $m$ is not. To show the relation stated above, we look at the connection of $m$ to the two distance measures.

There is a connection between $m$ and the edit distance of $A$ and $B$ (that is, the number of insertions and deletions needed to turn $A$ into $B$, or vice versa; this will be denoted as $e$). Assuming $|A| \geq |B|$, there must be at least $|A|-|B|$ elements of $A$ which must be deleted; once this is done, there are $|B|-m$ elements of $A$ which must be converted into elements of $B$. However, as substitutions are not allowed in this version of edit distance, the equivalent operation involves deleting each remaining mismatched item from $A$, then adding an item that matches one in $B$. This takes $2(|B| - m)$ edits. The total cost of these two steps, $e$, is $|A|-|B|+2(|B|-m) = |A|+|B|-2m$ edits, implying $\frac{|A|+|B|-e}{2} = m$.

There is also a connection between the Jaccard similarity and the minimum intersection $m$ between two sets, assuming the magnitudes of the sets ($|A|$ and $|B|$) are known. For a pair of sets, $J = \frac{|A \cap B|}{|A \cup B|} = \frac{m}{|A|+|B|-m}$. This can be rewritten as $\frac{J*(|A|+|B|)}{1+J} = m$.

We can now put the two equations together, giving $\frac{|A|+|B|-e}{2} = \frac{J*(|A|+|B|)}{1+J}$, which simplifies to $e = (|A| + |B|) * (1 - \frac{2J}{J+1})$. This also yields another important relation; if the edit distance, $e$, between $A$ and $B$ is known, the Jaccard similarity of the two sets is $\frac{|A|+|B|-e}{|A|+|B|+e}$. The relation between edit and Jaccard distance shown here will be useful when we discuss the splitting algorithm for Jaccard distance later.

**Naive Algorithm**

The Naive algorithm sends a chunk of the input to each (physical) reducer. Each reducer compares each possible pair of items in the input to see if it is within the desired edit distance or Jaccard similarity. If a pair is within this threshold, it is output.

The algorithm that distributes input to reducers is as follows: Let $n = \frac{k*(k+1)}{2}$ be the number of reducers, for some integer $k$. Each reducer is assigned an identifier $(i, j)$, where $0 \leq i \leq j \leq k$; this creates a triangular 'matrix' of reducers. Each item in the input is hashed to some value in $[0, k)$. If an input item has the hash $i$, it is sent to reducer $(i, j)$ or $(j, i)$ (whichever exists) for each $j \in [0, k)$. The input item is thus sent to $k$ reducers.

At this point, each reducer compares the input it receives. As an optimization, each input item keeps track of which half of its destination identifier contains its hash; for a reducer at $(i, j)$, items with a hash of $i$ are only joined with items which have a hash of $j$. This prevents comparing the same pairs of items more than once. If the reducer's identifier is of the form $(i, i)$, the input is joined to itself.

This algorithm is the same as the Naive covered in [14], with the exception that the Hamming distance comparator routine has been replaced with an edit distance comparator to calculate edit distance or Jaccard similarity.

**Splitting Algorithm**
In this algorithm, the mappers break the input into chunks of a specific length. Input items with matching chunks are sent to (logical) reducers; each reducer compares each possible pair of items within the input in the same way reducers for the Naive algorithm do.

In the Hamming distance implementation of this algorithm (as described in [14]), the mappers determine a chunk size, $t$, and break input strings into chunks of that size or smaller. This does not work for the Edit distance and Jaccard similarity implementations.

**Splitting Algorithm - Edit Distance**
In the Edit Distance Splitting algorithm, input strings that are very similar to one another may contain most of the same characters– shifted at some offset. This renders the method of creating fixed-position chunks impossible; as an example, consider the words 'adventurer' and 'misadventure'. The edit distance of these strings is 4 (deleting 'r' and adding 'mis' will convert 'adventurer' to 'misadventure'). However, if each string is split into fixed-position chunks of size (say) 2, the first string will have chunks 'ad','ve','nt','ur', and 'er', and the second string will have the chunks 'mi','sa','dv','en','tu','re', and 'r'– none of which match the chunks of the first words. The only way to have smaller chunks is to have chunks of one character, which may not be feasible for long strings.

Instead, to narrow down the pools of strings to be joined, the mappers for the edit-distance splitting algorithm determine a chunk size, then create a 'sliding window' that emits a substring of that size, starting at every possible position in the string. While the Hamming distance algorithm would produce, given an edit distance threshold $d$ and a string of length $l$, $d+1$ chunks of size $\frac{l}{d+1}$ ([2]), the edit distance algorithm will use the same chunk size and produce chunks of size $c_1 = \lfloor \frac{l-d}{d+1} \rfloor$ (to match strings smaller than the input string) and $c_2 = \lfloor \frac{l}{d+1} \rfloor$ (to match strings larger than the input string). The number of chunks produced of each size will be $(l - c_i) + 1$. However, this increase in the quantity of output drastically reduces the efficiency of the algorithm.

**Splitting Algorithm - Jaccard Distance**
We showed that Jaccard distance can be casted to edit distance, so, we could use the above sliding window technique to perform Splitting for Jaccard distance as well. However, as shown in the following, we can do better than that.

In the Jaccard Similarity Splitting algorithm, input is split into chunks based on its position in the sorted universe. That is to say: Consider a set where the universe of items is integers from 1 to 100. If a set $S$ is $\{1, 3, 5, 6, 10, 22, 54, 55, 56, 57, 92\}$, and we split into chunks of size $c = 5$, we will receive the chunks: $\{1, 3, 5\}, \{6, 10\}, \{22\}, \{54, 55\}, \{56, 57\}, \{92\}$. Each chunk contains *up to* 5 items from the universe; it can be defined as the subset of the universe containing items $(i-1) * c + 1$ to $i * c$ (for some $i$) intersected by $S$.

The splitting of the input sets into chunks also yields an extra constraint on the requested similarity. The chunk size must be set before the map phase starts, i.e. without the luxury of knowing the magnitude of any particular input set. As such, we must assume that every set is as large as possible so that the chunk size will work under any input conditions. This has two effects: it favors smaller chunk sizes than may be absolutely necessary to compare the items of a dataset, decreasing efficiency, and it limits the minimum similarity threshold the algorithm can accept as a parameter.

The reason for the limit on similarity is as follows: As per the relation between Jaccard and Edit distance listed above, the requested Jaccard similarity is converted to edit distance. However, the size of the universe of items ($|U|$) is used in place of both $|A|$ and $|B|$, because $|A|$ and $|B|$ are not known until specific items of input are read. Once a maximum permissible number of edits is known, $d$, the algorithm tries to split every input set into $d + 1$ chunks, like in the Hamming Distance version of the algorithm. However, if the requested similarity threshold is $\leq \frac{1}{3}$, this will require the set be split into more chunks than there are items in the universe! This is impossible, as chunks denote the presence or absence (in the input set) of items in $U$; more chunks than items in the universe implies some chunks would not relate to any items in $U$, a waste of communication cost and processing time. As such, the Jaccard similarity splitting algorithm will fail if a similarity $\leq \frac{1}{3}$ is requested.

## IV. Setup

Our experiments used different datasets for the edit distance and Jaccard similarity portions. The datasets used for the edit distance algorithms were:

- A chunk of DNA from the genome of *Drosophila melanogaster* (the fruit fly), downloaded from the UCSC Genome Browser (http://bit.ly/23mSG1a, August 2014 version, full dataset, chunk 'chr2L'). This dataset came pre-broken into 32-character strings, each character in the string representing a nucleotide. As such, this dataset has a very small alphabet: the characters A, T, C, G, N, where 'N' refers to an unknown nucleotide. The dataset contained 470,275 strings. We also experimented with a random selection of 50% of the above dataset, or 235,135 strings.
- A selection of 1,000,000 words chosen randomly from the Google Ngrams single-word database (http://bit.ly/1S6mTQ2, version 20120701, all files from '1-grams'). The words chosen contained only ASCII alphanumeric characters, to place a limit on the alphabet size. Words were of varying length.

The datasets used for the Jaccard similarity algorithms were:

- The MovieLens 10 Million dataset [11], containing 10,000,054 movie reviews by 71,567 users, on a universe of 10,677 movies. This data was preprocessed to create a sorted set for each individual user, containing numeric identifiers of all the movies that user had rated.
- The MovieLens 20 Million dataset, containing 20,000,263 movie reviews by 138,493 users, on a

universe of 26,744 movies. The data was preprocessed in the same manner as the 10 Million dataset.

For edit distance algorithms, the Drosophila datasets were tested on similarity thresholds from 1 to 8 edits, and the Ngrams dataset was tested on similarity thresholds from 1 to 6 edits (as the average length of strings in the sets were shorter than the Drosophila datasets).

For Jaccard similarity algorithms, a range of similarities from .50 to .95 were tested, incrementing in steps of .05. As stated above, the Splitting algorithm was unable to accept similarities $\leq .33$, which influenced the choice of similarities.

**Hadoop Cluster Configuration.** All data was processed using Hadoop 1.2.1, on an IBM BladeCenter cluster with 33 machines, divided into three chassis of 11, 11, and 9 machines, respectively. Inter-chassis and intra-chassis networking was provided by switches capable of 1 Gbit/second. Each machine had 4 Intel(R) Xeon(R) E5430 @ 2.66GHz processor cores, and 6 GB of memory. The cluster settings permitted 4 map jobs and 4 reduce jobs per machine at any time (effectively, one job per processor core). Each MapReduce child process (which handles a single job) was given 1 GB of memory. There were two 73GB Hot-Swap 3.5" 10K RPM Ultra320 SCSI HDDs per machine, each with a maximum transfer rate of 104 MB/sec.

## V. Results

Here is a summary of our results:

1) The Splitting algorithm did worse than the Naive algorithm, across all datasets (except at very low thresholds under controlled conditions). However, the reasons for the worse performance of Splitting is different for the two distance measures.
2) In the case of edit distance algorithms, the 'sliding window' that was implemented to handle shifted matching segments drastically increases processing time, rendering the splitting algorithm inefficient. The communication cost is lower than the Naive algorithm, however.
3) Nothing in the algorithm itself renders the Jaccard Splitting algorithm inefficient; however, the extremely long nature of the sets, combined with the sparse nature of the data, renders the algorithm impractical by the sheer number of 'set chunks' that must be created, each with their own copy of the original set. This drastically increases communication cost; moreover, the number of resulting chunks renders the processing time of the algorithm extremely prohibitive.

Further details about the experiments are in the rest of the section.

### A. Communication Cost

Each algorithm's communication cost has its own unique behavior:

**Naive Algorithm** The Naive algorithm's communication cost on any dataset does not change, even as the distance measure used or difference threshold changes. Increasing the size of the input does make the cost change, however; this is expected, as the Naive has to emit a fixed number of copies for each item in the input.

**Splitting Algorithm - Edit Distance** The communication cost for the edit distance version of the splitting algorithm is directly related to the number of chunks emitted per item in the input; this is based on the size of the 'sliding window' for any item of input, which is related to the maximum difference threshold.

In the Drosophila datasets (Figures 2, 3), communication cost is higher than that of the equivalent Naive algorithm. There is a dip in the Splitting algorithm's communication cost at difference threshold 2 only. Ignoring this dip, the Splitting cost is lower at threshold 1 than it is at threshold 3 and above.

This behavior is explained by a combination of normal function of the algorithm and the nature of the dataset. In the Drosophila datasets, all input strings are 32 characters long. This means that for a maximum difference threshold of $d$ edits, chunks of size $\lfloor \frac{32-d}{d+1} \rfloor$ and $\lfloor \frac{32}{d+1} \rfloor$ will be emitted for every input string. For a difference threshold of 1, this translates into 35 chunks of lengths 15 and 16; for a difference threshold of 2, 18 chunks, of length 15 only; and for a difference threshold of 3 and above, 37 chunks of lengths 14 and 15.

In the Ngrams dataset (Figure 4), the Splitting algorithm has a better communication cost than the Naive algorithm. In these datasets, input strings are of varying lengths, and are usually far shorter than 32 characters. Because of this, far fewer chunks are emitted on average per string, rendering the communication cost more manageable.

**Splitting Algorithm - Jaccard Distance** As the Jaccard version of the splitting algorithm does not have a 'sliding window' like the edit distance version does, the Splitting algorithm does not create many chunks over a small input item like the edit distance Splitting algorithm does. However, this is balanced out by the nature of the MovieLens datasets; sets of movies a user has rated tend to be both large and sparse, resulting in many chunks being emitted, each containing only a small portion of the set. The communication cost for the MovieLens datasets (Figure 5, 6) is thus significantly higher than the Naive algorithm's cost for the same datasets.

### B. Processing Time

For the processing time of *all* algorithms, the mapper and shuffle phases took trivial amounts of time, typically less than one minute. This implies that the mapper processing is highly efficient, and that transferring the data does not add any load of any kind.

**Naive Algorithm** While the processing time of naive algorithms typically does not tend to vary due to factors other than the size of the input, the processing time of Naive algorithms using edit distance or Jaccard similarity does increase slightly (and approximately linearly) as the difference threshold increases. This is because both algorithms use an edit distance comparator that returns a result immediately once it is able to do so; this is often easier to do with a tighter difference threshold, as non-matching pairs will fail more quickly.

**Splitting Algorithm - Edit Distance** The processing time for edit-distance Splitting algorithms tends to increase at a

higher order of growth than the Naive algorithms. This is likely due to the high number of overlaps generated by the 'sliding window' portion of the algorithm if there is a matching portion of two strings longer than the window size, each of which must be independently checked by the reducer. Notwithstanding, the Splitting algorithm is faster than the Naive algorithm when run on the Drosophila datasets (Figures 2, 3) at low distance thresholds (a maximum of 5 edits and below). This may be due to the low alphabet size of the data (which would lower the number of possible sequences of characters a reducer could receive), or due to a high degree of variation in the 32-character strings of the set (which would cause large chunks to mismatch more often).

For the Ngrams dataset (Figure 4), the Naive algorithm is far faster the Splitting algorithm. This is expected, to an extent; the short length of many words in the Ngrams set means that these words will have their 'sliding window' size be a single character in width, leading to the existence of a pool of all words that contain that character that some unlucky reducer node must join.

**Splitting Algorithm - Jaccard Distance** The processing time for Jaccard similarity Splitting algorithms also increases much faster than the Naive algorithms, even though the algorithm does not use a 'sliding window' method. This is because in MovieLens (Figures 5, 6), the movies a user has rated are in general sparse and tend to create many chunks, thus adding to the potential comparison overhead. Moreover, given the nature of the data, it is possible that matching chunks tend to cluster around popular movies, which may result in many pairs of users that do not fall within the desired similarity threshold having to be compared anyway.

### C. Further Observations

In these results, several points of interest emerge:

- The 'sliding window' approach of the edit-distance Splitting algorithm hampers its efficiency. However, this is only due to the possibility that large, matching chunks of a string could be shifted in a way that would confuse a fixed-chunk algorithm. There would be no need for this approach if a constrained variant of edit distance were used; one that disallows additions and deletions, but allows substitutions. This could lead to a drastic gain in the algorithm's efficiency under these conditions.
- For the Jaccard similarity Splitting algorithm, it would be interesting to test if the algorithm behaves more efficiently than the Naive algorithm when run on dense sets.

## VI. Conclusions

We presented a detailed evaluation of algorithms proposed by Afrati, Das Sarma, Menestrina, Parameswaran and Ullman in [2] using edit and Jaccard similarity. Furthermore, we presented details of algorithmic engineering and adaptations in order to scale up the considered algorithms in practice.

As future work, we would like to study enhanced edit operations with constraints in the style of [8], [9], [10], [22] as well as extending the results to edit distance over XML items (cf. [21], [23]).

## References


[1] M. Abualsaud and A. Thomo. Utilizing favorites lists for better recommendations. In *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*, pages 303–310. IEEE, 2014.

[2] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE'12*, pages 498–509, 2012.

[3] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *Knowledge and Data Engineering, IEEE Transactions on*, 23(9):1282–1298, 2011.

[4] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, 2013.

[5] D. G. Brizan and A. U. Tansel. A. survey of entity resolution and record linkage methodologies. *Communications of the IIMA*, 6(3):5, 2015.

[6] S. Ebrahimi, N. M. Villegas, H. A. Müller, and A. Thomo. Smarterdeals: a context-aware deal recommendation system based on the smartercontext engine. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, pages 116–130. IBM Corp., 2012.

[7] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 411–420. IEEE, 2015.

[8] G. Grahne and A. Thomo. Regular path queries under approximate semantics. *Annals of Mathematics and Artificial Intelligence*, 46(1-2):165–190, 2006.

[9] G. Grahne, A. Thomo, and W. Wadge. Preferentially annotated regular path queries. In *Database Theory–ICDT 2007*, pages 314–328. Springer, 2007.

[10] G. Grahne, A. Thomo, and W. W. Wadge. Preferential regular path queries. *Fundamenta Informaticae*, 89(2-3):259–288, 2008.

[11] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *TiiS*, 5(4):19, 2016.

[12] Y. He, K. Ganjam, and X. Chu. Sema-join: joining semantically-related tables using big table corpora. *Proceedings of the VLDB Endowment*, 8(12):1358–1369, 2015.

[13] H. Kardes, D. Konidena, S. Agrawal, M. Huff, and A. Sun. Graph-based approaches for organization entity resolution in mapreduce. *Graph-Based Methods for Natural Language Processing*, page 70, 2013.

[14] B. Kimmett, V. Srinivasan, and A. Thomo. Fuzzy joins in mapreduce: an experimental study. *Proceedings of the VLDB Endowment*, 8(12):1514–1517, 2015.

[15] B. Kimmett, A. Thomo, and S. Venkatesh. Three-way joins on mapreduce: An experimental study. In *Information, Intelligence, Systems and Applications, IISA 2014, The 5th International Conference on*, pages 227–232. IEEE, 2014.

[16] L. Kolb and E. Rahm. Parallel entity resolution with dedoop. *Datenbank-Spektrum*, 13(1):23–32, 2013.

[17] N. Korovaiko and A. Thomo. Trust prediction from user-item ratings. *Social Network Analysis and Mining*, 3(3):749–759, 2013.

[18] C. Li, S. Mehrotra, and L. Jin. Record linkage: A 10-year retrospective. In *Database Systems for Advanced Applications*, pages 3–12. Springer, 2013.

[19] T. Nie, W.-c. Lee, D. Shen, G. Yu, and Y. Kou. Distributed entity resolution based on similarity join for large-scale data clustering. In *Web-Age Information Management*, pages 138–149. Springer, 2014.

[20] I. Sandler and A. Thomo. Large-scale mining of co-occurrences: Challenges and solutions. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on*, pages 66–73. IEEE, 2012.

[21] M. Shoaran and A. Thomo. Evolving schemas for streaming xml. In *Foundations of Information and Knowledge Systems*, pages 266–285. Springer, 2010.

[22] D. C. Stefanescu, A. Thomo, and L. Thomo. Distributed evaluation of generalized path queries. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 610–616. ACM, 2005.

[23] A. Thomo, S. Venkatesh, and Y. Y. Ye. Visibly pushdown transducers

for approximate validation of streaming xml. In *Foundations of Information and Knowledge Systems*, pages 219–238. Springer, 2008.

[24] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.

[25] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 85–96. ACM, 2012.

[26] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15, 2011.

[27] C. Yan, Y. Song, J. Wang, and W. Guo. Eliminating the redundancy in mapreduce-based entity resolution. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1233–1236. IEEE, 2015.

[28] N. Yazdanfar and A. Thomo. Link recommender: Collaborative-filtering for recommending urls to twitter users. *Procedia Computer Science*, 19:412–419, 2013.

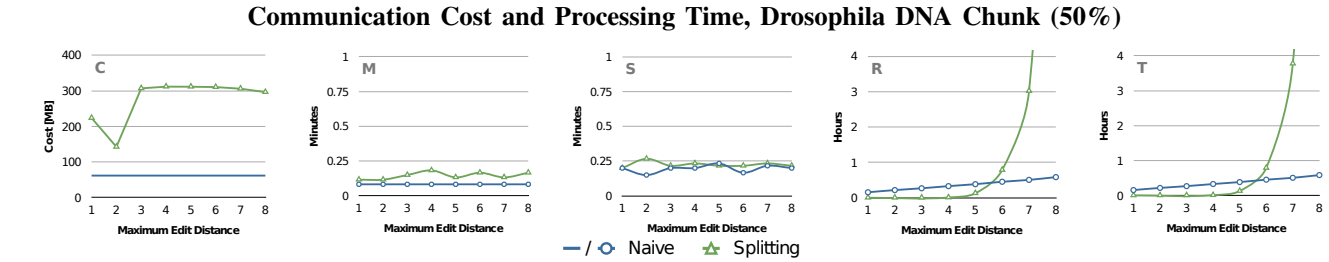


Fig. 2: Communication cost (C), mapper (M), shuffle (S), reducer (R), and total (T) time, from left to right. (For Splitting, distance 8, R and T are 16.30 hrs and 16.31 hrs, respectively. They do not fit in the plot area.)

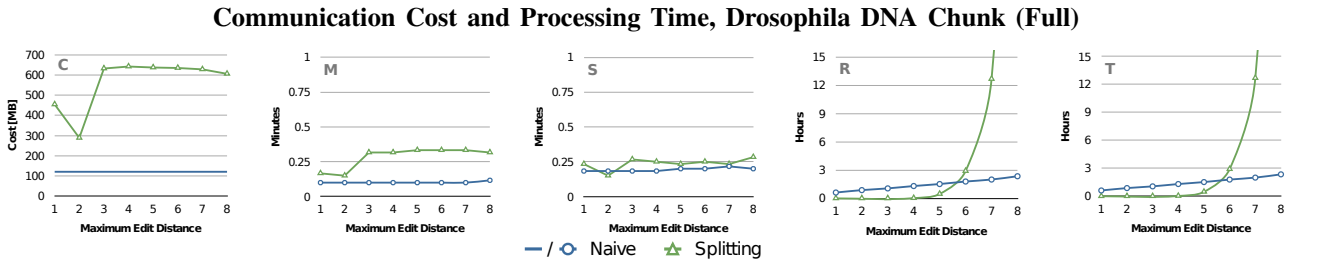


Fig. 3: For Splitting, distance 8, R and T are 64.91 hrs and 64.92 hrs, respectively. They do not fit in the plot area.

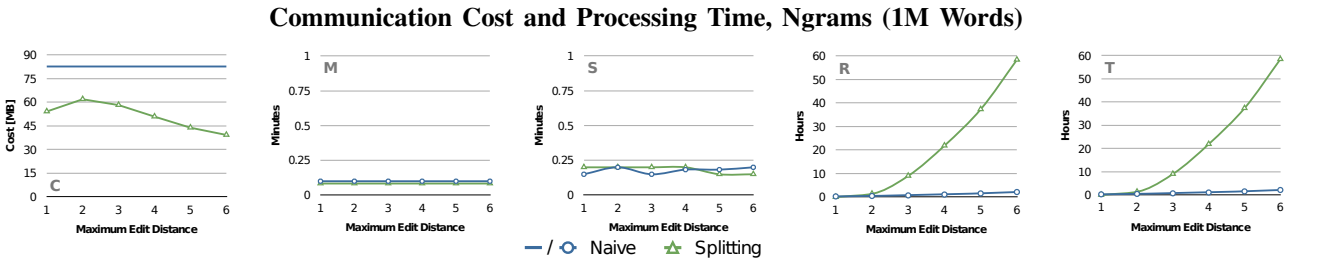


Fig. 4: Communication cost (C), mapper (M), shuffle (S), reducer (R), and total (T) time, from left to right.

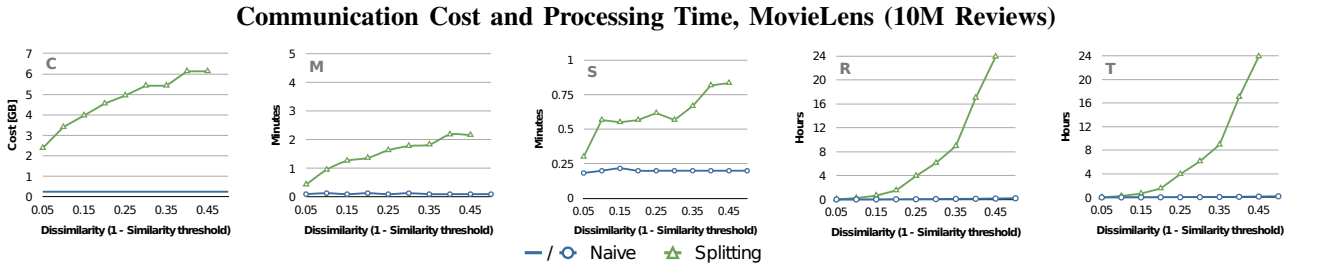


Fig. 5: For Splitting, the run with dissimilarity equal to .5 took longer than 24 hours to finish.

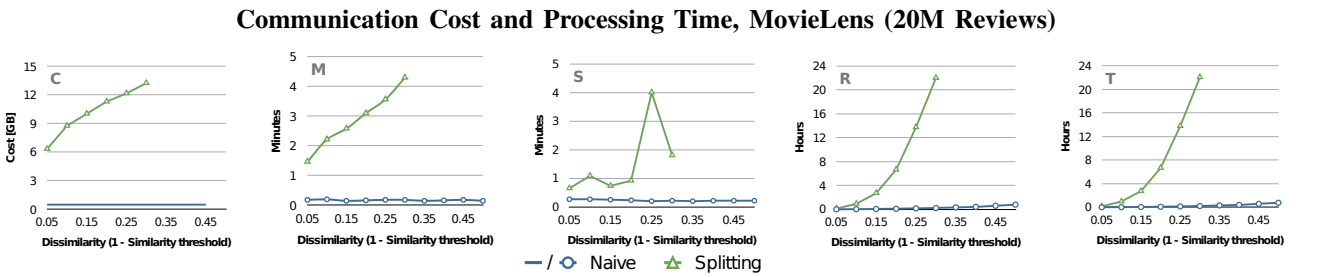


Fig. 6: For Splitting, the runs with dissimilarity greater than .3 took longer than 24 hours to finish.