Fuzzy Joins Using MapReduce

Foto N. Afrati [#1], Anish Das Sarma [⋆2], David Menestrina [⋆3], Aditya Parameswaran [†4], Jeffrey D. Ullman [†5]

[#]*National Technical University Athens* [⋆]*Google, Inc.* [†]*Stanford University*

[1,2,3]{afrati,anish.dassarma,dmenest}@gmail.com [4,5]{adityagp,ullman}@cs.stanford.edu

***Abstract*—Fuzzy/similarity joins have been widely studied in the research community and extensively used in real-world applications. This paper proposes and evaluates several algorithms for finding all pairs of elements from an input set that meet a similarity threshold. The computation model is a single MapReduce job. Because we allow only one MapReduce round, the Reduce function must be designed so a given output pair is produced by only one task; for many algorithms, satisfying this condition is one of the biggest challenges. We break the cost of an algorithm into three components: the execution cost of the mappers, the execution cost of the reducers, and the communication cost from the mappers to reducers. The algorithms are presented first in terms of Hamming distance, but extensions to edit distance and Jaccard distance are shown as well. We find that there are many different approaches to the similarity-join problem using MapReduce, and none dominates the others when both communication and reducer costs are considered. Our cost analyses enable applications to pick the optimal algorithm based on their communication, memory, and cluster requirements.**

I. Introduction

MapReduce is a popular and powerful framework for parallel data analytics. A number of research efforts in recent times have been focused on making the MapReduce paradigm easier to use, including layering a declarative language over MapReduce [1, 2, 3], dealing with data skew [4, 5], and finding efficient MapReduce counterparts of traditional algorithms such as graph algorithms [4, 6, 7, 8] and joins [9]. In what follows, we assume the reader is familiar with how MapReduce works. A tutorial can be found in [10].

MapReduce gives us the ability to leverage many machines working in parallel, thereby letting us process and analyze data sets that are orders of magnitude larger. However, being able to use MapReduce for some of the traditionally sequential algorithms without sending the entire data set to every reducer is not a straightforward task. In the typical MapReduce environment — racks of commodity computers connected by gigabit Ethernet — communication is a significant cost. Thus, replication of data at many processors needs to be avoided to the extent possible. In particular, figuring out which records need to be processed together at the same reducer for a given algorithm is especially tricky. In addition, in order to compare two MapReduce algorithms, one needs to reason about many system parameters all at once, including network latency, processor speeds, number of reducers, main memory size, and others, making theoretical analysis difficult.

The goal of this paper is to look at a nontrivial problem—*fuzzy joins*— on MapReduce. We shall define the problem formally in Section III, but intuitively, given a set of records we are interested in finding all pairs of records that are within some distance from each other according to some distance measure. There are many ways to apply MapReduce to this problem, but all except the most trivial (and generally least efficient) algorithm requires replication of data at many reducers. Moreover, we find that there is a tradeoff between communication cost and processing cost, and there are at least five significantly different approaches that belong to the "skyline"; i.e., none dominates another. One of the trickiest aspects of this problem is avoiding duplicate outputs from different reducers, and we develop a general technique involving lexicographic orderings of several types to solve this problem for each of our algorithms.

Fuzzy joins arise in many applications, including entity resolution, collaborative filtering, and clustering, for example. We consider a number of distance measures, but concentrate on Hamming distance because it is in a sense the simplest measure and lets us offer the clearest view of the various algorithmic approaches. We then show how to apply the same ideas to edit distance and Jaccard distance as well.

Our focus, in this paper, is to provide a theoretical analysis of various MapReduce-based similarity join algorithms, and compare them in terms of various parameters, including map and reduce costs, number of reducers, and communication cost. We present a suite of algorithms that span the spectrum of tradeoffs between each of these parameters, thereby enabling an application to determine the most suitable algorithm based on our analysis.

II. Related Work

MapReduce Versions of Algorithms: Much recent research on MapReduce has focused on developing MapReduce versions of standard algorithms. Lattanzi et. al. [6] develop approximate versions of graph algorithms based on filtering (a form of sampling) that run in a small number of MapReduce rounds. Other work has considered counting triangles (which is useful for computing clustering coefficients in social network graphs) [4], joins [9], evaluating joins with skewness considerations [3], matching advertisers to users while obeying capacity constraints [8] and approximation algorithms for max-cover [7], minimum spanning trees [11] and flow-shop scheduling [12].

Models of Computation: Recent work [11] has suggested a model of computation for MapReduce that is inspired by the PRAM model. This model, denoted $\mathcal{MRC}$, enforces a limited amount of storage per processor, as well as a limited number of processors. While this model has counterparts for each of the costs that we consider, it is less explicit about the various costs, namely map cost, communication cost (or shuffle/network

cost) and reduce cost. We focus on computation on a certain number of reducers, quantifying the amount of computation in each stage of the MapReduce job. Note that all our algorithms run in a single MapReduce job. Moreover, note that the amount of storage needed per reducer is nothing but a proxy for our communication cost split evenly over the number of reducers. Our analysis is more along the lines of Afrati et. al. [13].

An alternative model is the one recently proposed in [3], where the model used is leveraged to answer conjunctive queries. They include possible communication between mappers to deal with data skew, which we do not consider, since it is not in the basic MapReduce model. In addition, they ignore the computation cost at the reducers, while we explicitly consider all costs. Their primary emphasis is on reducing the number of rounds.

Fuzzy Joins: There are two approaches to fuzzy or set-similarity joins that have been considered in the past. One approach uses *approximate matching* techniques such as locality-sensitive hashing [14], which work especially well for low similarity thresholds. The other approach is *exact matching* techniques. These have the additional desirable property of always returning the correct output. In this paper, we focus on exact matching techniques.

There has been some recent work on fuzzy joins using MapReduce [15, 16]. Vernica et al. [15] tries to identify similar records based on the Jaccard similarity of sets, using the length/prefix-based methods of Chaudhuri et. al. [17], combined with the positional and suffix filtering techniques of Xiao et al. [18], and then parallelizes these techniques. The problem of Jaccard similarity of sets is effectively reduced to the problem of overlap of sorted strings. This approach to Jaccard similarity is known to be good when the similarity threshold is high. Experiments in [15] have shown that on some real data the PPJoin+ algorithm [18] can be efficiently parallelized. We compare this algorithm with ours in more detail in Section VII, after the details of our algorithms are explained.

Baraglia et. al. [16] show improvements over the Vernica [15] approach using a two-MapReduce-phase approach of indexing the prefix of every record, and computing the similarity of only those records that share a token in their prefix (as opposed to the Vernica [15] approach which broadcasts copies of the document for every token in the prefix.) This approach is inspired by an earlier one proposed in Elsayed et. al. [19]. We, on the other hand, use a single MapReduce phase, since the setup time for a MapReduce job is known to be costly.

The two papers mentioned above present one technique each that is optimized for one similarity function, only for the case where the universal set of tokens is large, and only when the similarity threshold large as well (enabling the pruning optimizations). See Section VII where we analyze algorithms based on the [18] in the same terms as the algorithms we propose.

We perform a more principled analysis of algorithms, applicable to a number of distance measures, comparing them in terms of various parameters, including map and reduce costs, number of reducers, communication cost. In fact, we present a suite of novel algorithms that span the spectrum of tradeoffs between each of these parameters, thereby enabling an application to determine the most suitable algorithm based on our analysis.

Finally, a recent paper [20] studies the problem of performing arbitrary theta-joins in a single map-reduce step. The main idea in the paper is to distribute the pairwise comparison of every tuple uniformly across a given set of reducers. Our focus, on the other hand, is to minimize the number of pairwise comparisons performed by explicitly looking at the fuzzy join criterion, i.e., only compare pairs of tuples that may be in the result.

III. Problem Definition

We start by formally introducing the problem we address in this paper, and enumerating the various parameters used to analyze each algorithm.

A. Fuzzy Join

Let $\mathcal{D}$ be the domain of all possible records.

Definition 3.1 (Similarity Function): A *similarity function* is a function $Sim : \mathcal{D} \times \mathcal{D} \rightarrow \mathbb{R}$.

Some examples of similarity functions are:

1) **Strings:** Some well-known string similarity measures include edit-distance, Jaro-Winkler, and other functions from the commonly used string similarity package SecondString [21]. For deduplication, string similarity measures often include a table of "transformations" between strings like "Bob" is the same as "Robert", "Blvd." and "Boulevard" are the same.
2) **Sets:** The most commonly used set-similarity measures include Jaccard, and other intersection-based similarities.
3) **Numeric:** A simple example of numeric similarity is the difference between two numbers.

Definition 3.2 (Fuzzy-Join Predicate): A fuzzy-join predicate $F = (Sim, \tau)$ is defined by a similarity function Sim and a threshold τ. The result of applying F to a set of records $\mathcal{S} \subseteq \mathcal{D}$ is $F(R) = \{(x, y) \mid x, y \in R, Sim(x, y) \geq \tau\}$. For $(x, y) \in F(R)$ we say $F(x, y) = 1$.

Thus a fuzzy-join is stated using a *distance measure* used to define the similarity, where we are required to find all pairs (x, y) with a distance of at most some pre-specified threshold. We wish to find algorithms that can efficiently return all the $(x, y) \in F(R)$ using MapReduce.

B. Costs of Map-Reduce Algorithms

For each map-reduce algorithm, we consider the following costs:

- Total map or preprocessing cost across all input records (M).
- Total communication cost (C) of passing data from the mappers to the reducers.
- Total computation cost of all reducers (R).

These costs are expressed in terms of the following parameters.

- The input dataset $\mathcal{S}$ and its size $|\mathcal{S}|$.
- The threshold defining the similarity function, which we shall normally express as a maximum distance d.
- Properties of the input data elements, especially the length of strings when the input is a set of strings, but

also other parameters such as the alphabet size for edit distance.
- The number of reducers K.

With the exception of the "naive algorithm" to be discussed in Example 3.4, what we refer to as the "number of reducers" K is really the number of keys and their associated lists of values, i.e., the maximum possible number of Reduce tasks. These key-list pairs must each be fed to a Reduce task, but the actual number of Reduce tasks may be less than K. The total execution time of the reducers does not depend on how many keys each Reduce tasks gets. In practice, because there is overhead associated with each Reduce task, we would want to use a number of Reduce tasks that is only a small multiple of the number of compute nodes available.

The number of mappers is never considered. We assume that the algorithm uses as many mappers as is appropriate to handle the input. Since the mappers typically operate on one input element at a time, the total map cost is not really affected by the number of mappers, although if we use too few, then the finishing time of the entire job will be unnecessarily high.

C. (M, C, R)-map-reducible algorithms

Next we define the notion of (M, C, R)-map-reducible algorithms, which allows us to compare different map-reduce algorithms for fuzzy joins.

Definition 3.3: Given a dataset $\mathcal{S}$, a join predicate F is (M, C, R)-*map-reducible* if there is a one-to-many mapping G from $\mathcal{D}$ to a domain $\mathcal{D}' = \{1, \ldots, K\}$ (where K is the number of reducers to be used) such that the following holds.

1) If $F(x, y) = 1$, then $G(x) \cap G(y) \neq \emptyset$.
2) Computing $G(r_i)$ for all $r_i \in \mathcal{S}$ takes $O(M)$ time.
3) The total communication cost $\sum_{r_i \in \mathcal{S}} |G(r_i)| \leq C$.
4) Finding all pairs (x, y) at each reducer i where $i \in G(x)$, $i \in G(y)$, and $F(x, y) = 1$ takes $O(R)$ time.

Item 1 in the definition implies that every pair of similar records are hashed to at least one reducer in common. The remaining three items describe the three cost measures by which we evaluate algorithms. Item 2 represents the *total preprocessing cost* at the mappers. Note that computation of G for a given r_i is independent of any other $r_j \in \mathcal{S}$. (i.e., the mappers do not know which other records are present in $\mathcal{S}$.

Item 3 gives the total amount of data transferred to the reducers. It is referred to as *total communication cost* in [9]. It represents the total amount of network resources needed for the computation.

Item 4 is the total processing time at the reducers. Sometimes, we have to compare each pair of elements sent to the same reducer, but there are algorithms that allow us to avoid many of these comparisons. In addition, it is necessary to make sure that each pair of similar inputs is produced by only one reducer. Often, this requirement is met easily, but in some cases the computation needed to avoid duplicates is significant. When the computation is implemented on a remote commercial web site $M + C + R$ is proportional to the rent the user has to pay for the resources.

An algorithm that evaluates F over MapReduce, with costs M, C and R, is as good or better than one that can be represented with costs (M', C', R') provided $M' = \Omega(M)$, $C' = \Omega(C)$, and $R' = \Omega(R)$. Our overall goal is to find MapReduce procedures along a skyline of (M, C, R) for each of the important fuzzy-join predicates.

Example 3.4: Let us begin with an example algorithm that we shall refer to as the *Naive Algorithm*. This algorithm works for any type of data and any similarity function. We shall assume that the similarity test takes unit time; if that is not the case, then the reducer cost must be multiplied by whatever time the similarity test takes.

Suppose our input is a set $\mathcal{S}$ of elements of some type. We shall arrange our K reducers in a triangle so that every pair of elements of $\mathcal{S}$ appears at exactly one reducer and is compared there. To form the triangle, each reducer is identified by a pair (i, j) such that $0 \leq i \leq j < J$ for some constant J. The number of reducers is $K = \binom{J+1}{2} = J(J+1)/2$. Note that J is proportional to $\sqrt{K}$.

Members of the input set $\mathcal{S}$ are hashed to J buckets from $0 \ldots J-1$. An element that hashes to i is sent by its mapper to all reducers whose identifying pair is (i, j) or (j, i) for some j. Note that each string is sent to exactly J reducers, since only one of reducers (i, j) and (j, i) actually exists (unless $i = j$, in which case they are the same reducer anyway). Thus, the cost for all the mappers as well as the communication cost is $O(|\mathcal{S}|J) = O(|\mathcal{S}|\sqrt{K})$.

Each reducer compares each pair of elements it receives. Assuming that the hash function distributes elements evenly, we may assume each reducer receives approximately the same number of elements. The total number of elements sent to the K reducers is $J|\mathcal{S}|$, so each reducer has $|\mathcal{S}|J/K = 2|\mathcal{S}|/(J+1)$ elements. The work done at each reducer involves comparing $\binom{2|\mathcal{S}|/(J+1)}{2}$ pairs of elements, so the work per reducer is $O(|\mathcal{S}|^2/K)$. As there are K reducers, the total reducer cost is $O(|\mathcal{S}|^2)$. In summary:*

$$\text{Number of reducers } = K$$

$$\text{Total map cost and communication, } M = C = |\mathcal{S}|\sqrt{K}$$

$$\text{Total reducer cost, } R = |\mathcal{S}|^2$$

That is, the Naive Algorithm is a $(|\mathcal{S}|\sqrt{K}, |\mathcal{S}|\sqrt{K}, |\mathcal{S}|^2)$-map-reducible algorithm.

In the following subsections, we offer a variety of more sophisticated algorithms that have all costs, M, C, and R, below $|\mathcal{S}|^2$. In Appendix A we prove that any algorithm that compares all sets of elements at some reducer must have a reduce cost at least $|\mathcal{S}|^2$. However, note that some of our algorithms, notably "Ball-Hashing-1" (Section IV-A) and "Anchor Points" (Section IV-D) use reducers that compare only a subset of the pairs of elements they receive.

D. Simplifications

In the rest of this paper, we focus on finding algorithms for the following fuzzy join predicates.

- Hamming Distance, in Section IV.
- Edit Distance, in Section V.
- Jaccard Distance, in Section VI.

*Here, and throughout, we omit the big-oh expression when describing counts of map cost, reducer cost, and communication cost.

There are a number of simplifications to the model that we shall make for the sake of easy exposition. We summarize them here.

- As in Example 3.4, we assume that the input set $\mathcal{S}$ is random and that no element appears more than once. As an important consequence, when we hash inputs and send them to reducers, each reducer receives the average number of elements. Note that this number may be more than $|\mathcal{S}|/K$, since input elements typically need to be sent to more than one reducer.
- We assume that input elements can have simple operations performed on them in unit time. These operations include copying (communication), comparing, and hashing.
- All algorithms are assumed to have a map phase that operates independently on input elements. We shall therefore often state the map cost on a per-element basis. To get the total map cost M, we may simply multiply by $|\mathcal{S}|$.
- There are several different ways we could estimate the costs associated with each algorithm. We have chosen to compute the average cost, on the assumption that each reducer receives the same amount of data. This assumption is realistic if data is distributed by a random mechanism (typically hashing). Since we are comparing algorithms, the particular assumptions about input are less important than the fact that we use the same assumptions for each algorithm.

IV. Hamming Distance

In this section we analyze algorithms for similarity join based on Hamming Distance:

> Given a set $\mathcal{S}$ of b-bit strings, and an integer $0 \leq d \leq b$, find the set
>
> $$\{(s_1, s_2) \mid \text{HD}(s_1, s_2) \leq d\}$$
>
> where $\text{HD}(s_1, s_2)$ denotes the *Hamming distance* (the number of positions in which the two strings disagree) between s_1 and s_2.

We use $B(b, d)$ to denote the number of b-bit strings that can be obtained by flipping the value of at most d bits of any given b-bit string. That is:

$$B(b, d) = \sum_{k=0}^{d} \binom{b}{k}$$

Therefore, $B(b, d)$ is roughly $b^d/d!$, assuming d is much smaller than b. When b is clear from the context, we shall just use $B(d)$ to denote the expression above, and call it the "ball of radius d."

Below, we present each algorithm and consider the map-cost M, the number K of reducers used, the total communication between mappers and reducers C, and the total processing cost R on the reducers. Table I summarizes these parameters for each of the algorithms. Note that the Naive Algorithm is the approach of Example 3.4 applied to Hamming distance.

Observe that in many, but not all, algorithms, $M = C$, since the job of the mappers is simply to distribute the input among the reducers. However, there are algorithms where $M > C$ because the mappers do more complex work.

Recall from Section III-D that copying, comparing, or moving a b-bit string can be done in unit time, and we do not include a factor b in running times simply to account for the lengths of strings. However, in several algorithms to be discussed, we need to do something with each bit of a string. In that case, we do properly include a factor b in running times.

A. Ball-Hashing Algorithms

First, we consider two different algorithm that each use $n = 2^b$ reducers, one for each possible b-bit string.[†] These algorithms use a similar idea of hashing balls around any input string:

Ball-hashing Algo-1: In this algorithm, there is one reducer for each of the n possible strings of length b. The mappers send each string s to all b-bit strings at a distance of at most d from it. Thus the map-cost is $B(d)$ per input element. To be precise, from the string s the mapper generates the key-value pair $(s, -1)$ and all key-value pairs (t, s) such that $t \neq s$ is in the ball of raduis d around s.

Each of the n reducers then checks if the string s corresponding to that reducer is one of its input strings. Note that this test is fast, since -1 will be the first element on the list associated with key s if s is in $\mathcal{S}$, and -1 will not appear otherwise. Call a reducer that received the -1 *active*. Each of the (at most $|\mathcal{S}|$) active reducers outputs all pairs consisting of s and one of the other strings that hashed to this reducer.

The total number of strings sent to all the reducers is $|\mathcal{S}|B(d)$. Since there are n reducers, the average number of strings received by a reducer is $|\mathcal{S}|B(d)/n$. Recall that we perform our analyses assuming all reducers receive the average amount of data. Thus, the total work done by all $|\mathcal{S}|$ active reducers is $|\mathcal{S}|^2B(d)/n$. We shall omit the (typically negligible) term n that accounts for the cost of each reducer deciding whether or not it is active. In summary:

$$\text{Number of reducers } = n$$

$$\text{Total map cost and communication, } M = C = (|\mathcal{S}|B(d)$$

$$\text{Total reducer cost, } R = |\mathcal{S}|^2B(d)/n$$

As stated, each pair $\{s, t\}$ is produced twice, once from the reducer for s and once from the reducer for t. If we order strings, say lexicographically, we can inhibit producing the pair $\{s, t\}$ at the reducer for s if t preceeds s in the order.

Ball-hashing Algo-2: Again we use one reducer for each of the n bit strings of length b. However, in this algorithm, we hash to a reducer only those strings of distance $d/2$ or less.[‡] In this algorithm, all n reducers are "active." The reason is that each reducer not only has to compare the string to which

[†]That may well be a huge number of reducers. However, we should remember that for all the algorithms discussed except the Naive Algorithm, a "reducer" is a single key and its associated value list, not a Reduce task or a compute-node. It is permissible, and indeed necessary in certain circumstances, to execute more than one reducer at a single compute node.

[‡]Here and elsewhere, we use fractions like $d/2$ that may not be an integer as if they were rounded up to an integer. That is, to simplify notation, we omit the ceiling function in places where it is obviously needed.

Algorithm	Map-cost Per Element	#Reducers	Communication (C)	Processing (R)
Naive	J (approx. $\sqrt{K}$)	K (arbitrary)	$\lvert\mathcal{S}\rvert\sqrt{K}$	$\lvert\mathcal{S}\rvert^2$
Ball-hashing-1	$B(d)$	n	$\lvert\mathcal{S}\rvert B(d)$	$\lvert\mathcal{S}\rvert^2 B(d)/n$
Ball-hashing-2	$B(d/2)$	n	$\lvert\mathcal{S}\rvert B(d/2))$	$\lvert\mathcal{S}\rvert^2\big(B(d/2)\big)^2 b/n$
Splitting	$d+1$	$(d+1)n^{\frac{1}{d+1}}$	$(d+1)\lvert\mathcal{S}\rvert$	$(d+1)\lvert\mathcal{S}\rvert^2/n^{\frac{1}{d+1}}$
Hamming Code	$b^2 \log b$	$n/(b+1)$	$b\lvert\mathcal{S}\rvert$	$\frac{b^2}{b+1}\lvert\mathcal{S}\rvert$
Anchor Points	$\min\big(B(2d), \frac{n}{B(d)}\big)$	$\frac{n}{B(d)}$	$\lvert\mathcal{S}\rvert\frac{B(2d)}{B(d)}$	$\lvert\mathcal{S}\rvert B(d)/n$

TABLE I

SUMMARY OF THE MAP COST PER INPUT ELEMENT, NUMBER OF REDUCERS, COMMUNICATION COMPLEXITY, AND THE TOTAL REDUCER COST FOR VARIOUS HAMMING-DISTANCE JOIN ALGORITHMS. ALL EXPRESSIONS OMIT A BIG-OH; I.E., THEY ARE ORDER-OF-MAGNITUDE.

it corresponds (its key) with all the other strings that arrive at the reducer, but it must compare all the strings that arrive there with each other. As a result, each pair at distance d or less is produced by many reducers, and we need to inhibit the output at all but one of these to avoid duplicates.

Example 4.1: Suppose strings s and t have Hamming distance exactly d. Then there are $\binom{d}{d/2}$ reducers that will be distance $d/2$ from both s and t; these reducers corresponding to each of the strings that are formed by flipping exactly half the bits in which s and t differ.

To begin, the mappers send each string s to the reducer for each input string at a distance at most $d/2$ from s. Thus the map-cost is $B(d/2)$ per input string. The average number of strings received by a reducer is $|\mathcal{S}|B(d/2)/n$. Each pair of these strings must be compared, so the total number of pairs compared at all the reducers is $|\mathcal{S}|^2\big(B(d/2)\big)^2/n$. A pair $\{s,t\}$ is output by the reducer if and only if this reducer corresponds to a string u that is lexicographically first among all the strings that are distance at most $d/2$ from both s and t.

It is not hard to examine s and t and compute that lexicographically first string. In particular, it can be done in time $O(b)$, where as usual, b is the length of strings. Here is how to find the lexicographically first intermediate.

Suppose s and t are at distance $e \leq d$. Starting with s, we can change 1's to 0's, from the left, at positions where s and t disagree. We can change up to $d/2$ of these to get the intermediate u. However, if $e < d$, then we have another option. We can change up to $(d-e)/2$ 1's to 0's in positions where s and t both have 1. These positions are then changed back to 1's when we go from u to t. Thus, to construct u from s, scan s from the left. If we encounter a 1 where t has a 0, change that 1 to a 0. If we encounter a 1 where t is also 1, change the 1 to a 0 provided we have not already encountered $(d-e)/2$ positions where both s and t are 1. Stop when we have changed $d/2$ 1's to 0's. The resulting string is the lexicographically first u that is distance at most $d/2$ from both s and t.

Example 4.2: Let
$s = 101101001100$
$t = 101001101001$
and let $d = 6$. Since s and t have distance 4, we have $e = 4$ and $(d-e)/2 = 1$. That is, we can flip one 1 that s and t share in common.

Start scanning s from the left. The first position has 1, and t agrees. So we change that 1 to 0 and we have exhausted our budget of 1's that may be flipped even though s and t agree. Thus, we cannot flip the 1 in position 3. However, in position 4, s has 1 and t has 0, so we flip that 1. It is not until the 10th position that we encounter another 1 in s where t has 0. Thus, u is constructed from s by replacing 1's by 0's in positions 1, 4, and 10. That is, $u = 001001001000$. Note that t is distance $d/2 = 3$ from u. We obtain t from u by flipping positions 1, 7, and 12.

$$\text{Number of reducers } = n$$

$$\text{Total map cost and communication } M = C = |\mathcal{S}|B(d/2)$$

$$\text{Total reducer cost, } R = |\mathcal{S}|^2\big(B(d/2)\big)^2 b/n$$

B. Splitting Algorithm

In the splitting algorithm the b-bit strings are divided into $(d+1)$ equal-length substrings.[§] Note that if $\text{HD}(s,t) \leq d$, then at least one of the $(d+1)$ corresponding substrings of s and t match exactly. For each of the $d+1$ substrings of length $b/(d+1)$, we have a family of reducers that correspond to the $2^{b/(d+1)}$ possible values of that substring. If string s consists of substrings $s_1 s_2 \cdots s_{d+1}$, then a copy of s is sent to the reducer in the ith family that corresponds to s_i. More precisely, a key-value pair is created where the key is (i, s_i) and the value is s. Note that the number of reducers is $(d+1)2^{b/(d+1)}$.

Each reducer compares all the strings it receives to find those pairs at distance at most d. We shall assume that the work done by each reducer is proportional to the square of the number of strings received, although there are locality-sensitive hashing techniques (see [10]) that can reduce this cost. On our assumption that the work of the reducers is equal, each reducer receives $|\mathcal{S}|/2^{b/(d+1)}$ strings, and so does work $O(|\mathcal{S}|^2/2^{2b/(d+1)})$. Since the number of reducers is $(d+1)2^{b/(d+1)}$, the total work of the reducers is $O((d+1)|\mathcal{S}|^2/2^{b/(d+1)})$. Since $n = 2^b$, we can simplify these expressions by replacing $2^{b/(d+1)}$ by $n^{1/(d+1)}$.

A final point is that we again need to avoid outputting the same pair more than once. However, the solution is simple in this case. When a reducer in the ith family finds that s and t are at distance d or less, it checks that there is no $j < i$ for in which s and t are also equal in their jth substrings. If there is no such j, then the pair $\{s,t\}$ is sent to the output. In summary:

$$\text{Number of reducers } = (d+1)n^{1/(d+1)}$$

$$\text{Total map cost and communication, } M = C = (d+1)|\mathcal{S}|$$

$$\text{Total reducer cost, } R = (d+1)|\mathcal{S}|^2/n^{1/(d+1)}$$

The algorithm above can be further refined by applying "recursive" splitting as in the Hamming-distance search algorithm presented in [18]

[§]If $d+1$ does not divide b evenly, then some of the last pieces will be one shorter than the first pieces. In our discussion we shall assume an equal division.

C. Hamming Codes

As an introduction to the algorithm called Anchor-Points, let us consider a very special case, where $d = 1$ and b is one less than a power of 2. In this case, there is a subset of the binary strings of length b called a *Hamming code* [22] with several unusual properties:

1) The number of strings in the Hamming code is $n/b + 1$.
2) Every string of length b is either in the Hamming code or at distance 1 from a unique member of the Hamming code.
3) Given a string s of length b, we can find efficiently whether s is in the code or, if not, which member of the code is at distance 1. The time required to make this determination is $O(b \log b)$.

We can find strings at distance 1 in an input set $\mathcal{S}$ by using $n/(b+1)$ reducers, each corresponding to one member of the Hamming code. Any pair at distance 1 either consists of a *codeword* (member of the Hamming code) and another word at distance 1, or it consists of two non-codewords, s and t. In the latter case, s will be distance 1 from a unique codeword, and t will be distance 2 from the same codeword. We therefore require the mappers to send to each reducer all the strings in $\mathcal{S}$ at distance up to 2 from its codeword.

The mappers take each input string s and determine whether it is in the code. If so, s is sent to the reducer for s. If not, then the mapper finds the string t in the code at distance 1 from s and sends s to the reducer for t. Then, the mapper flips, in turn, each bit of s other than the one that turns s into t. These $b-1$ strings are the other strings at distance 1 from s. For each of these strings t, find the codeword at distance 1 from t and send s to the reducer for that codeword.

Note that the work done by the mapper for each input string is $O(b^2 \log b)$. The reason is that there are b strings (s and its neighbors at distance 1) for which we must determine the nearest codeword, and that determination takes $O(b \log b)$ time. On the other hand, the communication is only b, since each string is sent to b reducers.

The reducer for code word s first builds an index of received words, so it can look them up in $O(1)$ time. It then checks whether it has received s. If s was received, then it outputs all pairs consisting of s and one of the other received strings at distance 1. Also, for each string t at distance 1 from s that was received, the reducer finds all strings u at distance 1 from t that were received, and outputs the pair $\{t, u\}$, provided t precedes u lexicographically. Note that the pair $\{t, u\}$ will be discovered twice, once at the reducer for the codeword at distance 1 from t and once at the reducer for the codeword at distance 1 from u. We therefore need to avoid emitting this pair twice. On the other hand, pairs containing a codeword are discovered only at the reducer for that codeword, and thus must be output regardless of which is lexicographically first.

On our assumption that each reducer receives an average number of strings, a reducer receives $b|\mathcal{S}|/n$ strings at distance 1 from its codeword and $\binom{b}{2}|\mathcal{S}|/n$ strings at distance 2. The index of strings received, lets us test the presence of any given string in $O(1)$ time. Thus, a reducer requires $O(b)$ time to determine which of the strings at distance 1 from its codeword are present, since it must flip each bit in turn and do a lookup in the index. Then, for each of the $b|\mathcal{S}|/n$ strings at distance 1 from the codeword, it does a similar task by flipping each of the $b-1$ bits in turn that do not lead to the codeword and testing membership of the resulting string. Thus, the total work done at each reducer is $O(b^2|\mathcal{S}|/n)$. In summary:

$$\text{Number of reducers } = n/(b+1)$$

$$\text{Total map cost, } M = b^2 \log b|\mathcal{S}|$$

$$\text{Total communication, } C = b|\mathcal{S}|$$

$$\text{Total reducer cost, } R = \frac{b^2}{b+1}|\mathcal{S}|$$

D. Anchor-Points Algorithm

The Anchor-Points algorithm generalizes the method of Section IV-C to the extent possible. In place of the Hamming code, it uses a set $\mathcal{A}$ of *anchor points* such that all b-bit strings are within d-bits from at least one anchor point. Since one anchor point can only "cover" $B(d)$ strings, the minimum number of anchor points we need is $n/B(d)$. A set of anchor points meeting this bound is called a *perfect code*. Unfortunately, there are very few perfect codes; the Hamming codes are almost the only example. We discuss perfect codes in Appendix C. On the other hand, we can find anchor-point sets with not too many more strings than the theoretical minimum, as we shall show in Section IV-D1. Thus, in our presentation of the Anchor-Points algorithm we shall take $n/B(d)$ as the size $|\mathcal{A}|$ of the set.

Let us assume that each mapper has available to it the set $\mathcal{A}$, indexed so we can test membership of a string in $\mathcal{A}$ in $O(1)$ time. Since the number of anchor points is small for large d, this assumption is reasonable, although strictly speaking we should add a term $|\mathcal{A}|$ times the number of mappers to the total communication cost to account for sending copies of $\mathcal{A}$ to each mapper. There is a reducer for each anchor point.

The mappers send each string s in $\mathcal{S}$ to every anchor point at a distance of at most $2d$ from s. There are two ways we can identify those anchor points for a string s.

1) We can generate from s all those strings at distance at most $2d$ and test their membership in $\mathcal{A}$. The work of doing so is clearly $O\big(B(2d)\big)$ per input string, since we have an index to make each membership test in $O(1)$ time.
2) We can consider all anchor points and compare them with s to see if they are at distance $2d$ or less from s. This approach takes time $O(|\mathcal{A}|) = O\big(n/B(d)\big)$ per input string.

In [23] these are called the "query expansion" and "linear scan" approaches. Their third approach, "table expansion," is not useful in the Map phase of a MapReduce algorithm. Since either approach could be more efficient, depending on $|\mathcal{A}|$, b, and d, we shall take the map cost per input string to be the minimum.

There is one more detail regarding the mappers that is essential to assure that pairs are not output more than once. We need to assign a unique *home* reducer to each string in $\mathcal{S}$. The home reducer for s is the one that corresponds to the closest anchor point to s, and in case of ties the lexicographically first such anchor point. The mapper that handles s, while searching

for nearby anchor points, determines the home for s, and when it sends s to that reducer, it tags s with the label "home." Note that if s and t are at distance at most d, then t will surely be sent to the home reducer of s, and vice-versa. We can avoid generating $\{s,t\}$ twice by requiring not only that each pair generated by a reducer involve at least one string of which it is the home, but by insisting that the home string lexicographically precede the non-home string.

For the communication cost, observe that each of the $|\mathcal{S}|$ input strings is sent to all the anchor points in the ball of radius $2d$ around the string. The probability that a given string is an anchor string is $|\mathcal{A}|/n$. Since we assume $|\mathcal{A}| = n/B(d)$, this probability is $1/B(d)$, and the expected number of points to which we send an input string is $B(2d)/B(d)$. Thus, the communication cost is $|\mathcal{S}|B(2d)/B(d)$.

The number of reducers is $n/B(d)$. The number of strings sent to reducers by all the mappers is $|\mathcal{S}|B(2d)$; this figure is also the communication cost. Thus, the average number of strings per reducer is $|\mathcal{S}|B(d)B(2d)/n$. Of these the expected number of strings of which the reducer is the home is $|\mathcal{S}|B(d)/n$.

However, we do not have to compare all pairs of these strings. We need only consider the strings s for which this reducer is home and, for each s, all strings t of distance at most d from s. The expected number of strings s is $|\mathcal{S}|B(d)/n$, as mentioned above. The number of possible strings t at distance up to d from s is $B(d)$. Each of these must be looked up in the index to see whether they are present at the reducer. If present and lexicographically after s, the pair $\{s,t\}$ is emitted. The total work at each reducer is thus $|\mathcal{S}|\big(B(d)\big)^2/n$. Since there are $n/B(d)$ reducers, the total work at all reducers is $|\mathcal{S}|B(d)$. In summary:

$$\text{Number of reducers } = n/B(d)$$

$$\text{Total map cost, } M = |\mathcal{S}|\Big(\min\big(B(2d), \frac{n}{B(d)}\big)\Big)$$

$$\text{Total communication, } C = |\mathcal{S}|B(2d)/B(d)$$

$$\text{Total reducer cost, } R = |\mathcal{S}|B(d)/n$$

1) Finding Good Sets of Anchor Points: A simple randomized algorithm can be used to find a covering set of anchor points with around $\frac{n \log n}{B(d)}$ points. In case of a "perfect covering", we would need $\frac{n}{B(d)}$ anchor points, so the number of points found by the randomized algorithm is more by at most a logarithmic factor. Proofs of these observations can be found in the appendix.

E. Comparison of Algorithms via an Example

Example 4.3: Recall that the key parameters for each of the algorithms were summarized in Table I. The functions given there omit constant factors, so they cannot be treated as absolutes, but only as expressions of how these paramaters grow as the data size grows. Nevertheless, it is interesting to use these expressions and see how the algorithms compare for a concrete example. We choose $b = 20$, so $n = 2^{20}$ or about one million. We shall use $d = 4$, so $B(d) = 6226$, $B(d/2) = 211$, and $B(2d)$ = 263,980. We also take $|\mathcal{S}|$ to be 100,000. For the naive algorithm, where the number of reducers is not fixed, we take K = 10,000.

Table II shows the values for the various costs. It is interesting to note that no algorithm dominates another. That is, in order of communication cost, the preference order of the algorithms is Splitting, Anchor, Naive, Ball-2, Ball-1, while for reducer cost, the preference order is Ball-1, Ball-2, Anchor, Splitting, Naive. Using these two orders, the only possible domanances are that Splitting and Anchor are better than Naive. However, Naive does have the ability to adjust the number of reducers, and it is easy to see that with $K = 1$ (i.e., run the entire algorithm on a single processor), the communication cost for Naive would be only 10^5. That cost is less than the communication cost for any of the other algorithms, including Splitting and Anchor.

V. Edit Distance

Given a set of edit operations on strings, with associated costs, the edit distance between a pair of strings is defined as the least-cost path from one string to the other:

Definition 5.1 (Edit Distance): Consider a space of edit operations $\mathcal{E}$ and an associated weight function $w : E \to \mathbb{N}$. The *edit distance* between two strings s_1 and s_2 of lengths b_1 and b_2 is $e(s_1, s_2)$ if and only if:

1) There is a sequence $e_1, \ldots, e_k$ of operations transforming s_1 to s_2 with total cost $\sum_{i=1}^{k} w(e_i) = e(s_1, s_2)$
2) No sequence of operations gives a total cost less than $e(s_1, s_2)$.

We consider the edit operations of *insertion* and *deletion* of a character at an position, where each of these operations is of unit cost. This is the most common case of edit distance.

In the remainder of this section, we study map-reducability of edit distance. We start by giving a generic mapping that shows how our techniques for Hamming distance can be applied for edit distance (Section V-A). It turns out that the naive, ball-hashing-1, and ball-hashing-2 algorithms carry over directly, as we state in Section V-B. The splitting algorithm and the anchor-points algorithm need some technical modifications, and are considered in Section V-D and V-F respectively. We also discuss another algorithm called the subsequence algorithm which is not interesting in the case of Hamming distance in section V-C. All our results on edit distance are summarized in Table III.

A. Counting the Number of Edits

For insertion and deletion with unit cost, the following is well known:

Property 5.2: The edit distance between strings s_1 and s_2 of lengths b_1 and b_2 is given by $e(s_1, s_2) = (b_1 + b_2 - 2l)$, where l is the length of the longest common subsequence of s_1 and s_2.

In what follows, we assume that there is a predefined bound on the maximum length of a string and a fixed alphabet of size q, so the space of possible strings is finite. We define three kinds of *balls* of radius d around a string s of length l:

- Balls that result from s by performing at most d deletions or insertions; the size of such a ball is denoted $B_s(d, l)$.

Algorithm	Map-cost Per Element	#Reducers	Communication (C)	Processing (R)
Naive	100	10^4	10^7	10^{10}
Ball-hashing-1	6226	10^5	6.2×10^7	6.2×10^8
Ball-hashing-2	211	10^5	2.1×10^7	4.5×10^8
Splitting	5	80	5×10^5	6.3×10^8
Anchor Points	160	160	4.2×10^6	6.2×10^8

TABLE II
VALUES OF EXPRESSIONS FROM TABLE I WHEN $b = 20$, $d = 4$, $|\mathcal{S}| = 10^5$, AND $K = 10^4$.

- Balls that result from s by performing at most d deletions; we denote their size by $B_s^{del}(d, l)$.
- Balls that result from s by performing at most d insertions; we denote their size by $B^{ins}(d, l)$, since, as we shall see, the size of this ball depends only on the length of s and not on the particular symbols comprising s.

Unlike the case of Hamming distance, when we deal with edit distance, the sizes of balls B and B^{del} depend on the particular string s, not just its length. That is why we included the string s as a subscript in those cases. (When the string s is understood, we shall omit the subscript s, and if the length l is also understood, we shall omit that argument.)

Example 5.3: Consider the strings 000 and 010 with a binary alphabet and $d = 1$. The length is $l = 3$ is both cases. However $B_{000}^{del}(1, 3) = 2$, since the only possible strings that can result from 000 after up to one deletion are 000 and 00. Also, $B_{010}^{del}(1, 3) = 4$, since any of the strings 010, 01, 11, or 10 can result from up to one deletion.

When we delete up to d positions of a string of length l, we cannot get more than $\sum_{i=0}^{d} \binom{l}{i}$ different strings. Note that this expression is what we called $B(d)$ in Section IV. Typically, some of these deletions will yield the same string, as we saw in Example 5.3. nevertheless, the upper bound $B^{del}(d, l) \leq \sum_{i=0}^{d} \binom{l}{i}$ will serve. To be exact, the number of ways we can delete depends on the number of *runs* (a run is a maximal sequence of identical symbols) a string has. The number of ways we can make one deletion from a string equals the number of runs it has. That is why, in Example 5.3, when $s = 000$, a string with only one run, we could only produce one string, 00, by deletion. However, with $s = 010$, a string with three runs, a single deletion can produce the three different strings 10, 00, and 01.

It turns out that $B^{ins}(d, l)$ is independent of the string; its size is given by

$$B^{ins}(d, l) = \sum_{i=1}^{d} \sum_{j=0}^{i} \binom{l + i - j}{i - j} q^j (q - 1)^{i-j}$$

For $B(d, l)$ we can give a tight upper bound, which we shall use as the size of a ball when needed in the algorithms that follow:

$$B(d, l) \leq \binom{l + d}{d} q^d q^d$$

Both these formulas are proven in Appendix E.

Next we consider turning all our Hamming-distance algorithms into algorithms for edit distance. Most of the ideas for Hamming distance carry over, now that we have defined a notion of a ball for strings with the edit-distance operations. If strings have widely varying lengths, then we define disjoint sets of reducers, each set collecting strings of length within d of a certain number and break the problem into disjoint problems each with smaller input.

B. Ball-hashing-1, Ball-hashing-2

The ball-hashing algo-1 depends only on the fact that there is a finite set of domain elements at distance d from a specific element and there is an algorithm for computing the specific distance. So, the algorithm itself carries over as is. The part that concerns not outputting the same string twice can also be implemented here, since we can again define a lexicographic order on the input strings. The analysis of the algorithm is different only as concerns the size of the balls. Now the balls may be of different sizes since the size depends on the string that corresponds to each reducer. Thus the performance measures of this algorithm will be given in terms of the upper bound on the size of the balls: $B(d) = \binom{b+d}{d} q^d \geq B(d, l)$, where l is the size of any string whereas b is the maximum size of an input string. Besides this change the counting remains the same as in the Hamming distance case.

For the same reason the ball-hashing algo-2 could remain the same (with the balls being again of different sizes) if we didn't care about outputting the same pair more than once. Thus, we have to explain how we avoid this in the case of edit distance. Observe that if we have two strings at edit distance less than or equal to d then they will both be included in the reducers which correspond to all their longest common subsequences. The reducer that finally will output the pair $\{s, t\}$ is the subsequence chosen as follows:

1) Choose which of s and t precedes the other; say it is s.
2) Find the common subsequence of s and t, of the correct length, that is leftmost in s. Given the two strings of length up to b we can find this particular common subsequence in time $O(bd)$ as explained in Appendix F.

Thus in the cost measures, the only difference (besides the difference as regards the size of the balls that was pointed out for the ball-hashing algo-1 above) with the Hamming distance algorithm is that the reducers processing cost is multiplied by d.

C. Subsequence Algorithm

Define a k-subsequence for a string to be any subsequence of length k found within the string. Suppose all strings are of length b. (Our techniques can be adapted easily for the case where strings are of varying length, see Section V-E.) We index all $(b - \frac{d}{2})$-subsequences of any input string. We claim that at least one of the subsequences will appear intact in any string of edit distance d from a particular string. We can prove this claim as follows: Suppose the length of a longest common subsequence is l_0 then we have that $2b - 2l_0 < d$; i.e., $l_0 > b - \frac{d}{2}$. Hence if none of the subsequences is common between the strings, then the length of a longest common subsequence will be less than $b - \frac{d}{2}$, leading to a contradiction.

We can now analyze the subsequence algorithm based on creation of all subsequences as described above. In this case the number of the reducers is $q^{b-d/2}$, which is the maximum

possible number of subsequences. (If we have fewer subsequences, we need fewer reducers.) The total communication cost is $C = |\mathcal{S}|\binom{b}{b-\frac{d}{2}}$, because we need to consider all $(b-d/2)$-subsequences of each string, and their number is $\binom{b}{b-\frac{d}{2}}$. The map-cost per element is the same $\binom{b}{b-\frac{d}{2}}$. The processing cost in the reducers is calculated as follows: We have $q^{b-d/2}$ reducers, hence each reducer contains $|\mathcal{S}|\binom{b}{b-\frac{d}{2}}q^{d/2-b}$ elements. Hence the total processing cost in the reducers is (remember we have an extra cost bd per pair to avoid including the same pair twice in the output):

$$|\mathcal{S}|^2\binom{b}{b-\frac{d}{2}}^2 q^{d-2b}bdq^{b-d/2} = |\mathcal{S}|^2\binom{b}{b-\frac{d}{2}}^2 q^{d/2-b}bd$$

The algorithm to let a reducer decide whether or not to output a discovered pair, so that each pair is output only once, is again the algorithm explained in Appendix F.

For Hamming distance we didn't include an algorithm based on subsequences because it will be always beaten by the Ball-hashing algorithm-1.

D. Splitting Algorithm

We assume all strings are of equal length. (See Section V-E.) Thus if strings s and t have edit distance d, we go from s to t by deleting $d/2$ characters and inserting $d/2$. This will use d positions of string s (we can imagine that we insert first to s and then delete – it is easy to see that we never delete what we have inserted). Thus at least one substring of length $b/(d+1)$ will appear intact in both.

Thus we hash according to substrings of length $b/(d+1)$. There are $q^{b/(d+1)}$ such substrings. Hence we have $q^{b/(d+1)} = n^{1/(d+1)}$ reducers. The map-cost is equal to the number of substrings in each string which is $b - b/(d+1)$. Each string is hashed to $b - b/(d+1)$ reducers, hence communication cost is equal to $|\mathcal{S}|(b - b/(d+1))$. Each reducer receives $(b-b/(d+1))q^{b/(d+1)}|\mathcal{S}|/n$ strings on average because there are $q^{b-(b-b/(d+1))}$ strings which begin with the given shingle and if we move the string to all its possible $b - b/(d+1)$ positions inside a string of length b then we get that all the strings that contain at least once the specific substring are $(b - b/(d+1))q^{b/(d+1)}$ and after canonicalization we get the result. Thus the work in all the reducers is $q^{b/(d+1)}((b-b/(d+1))q^{b/(d+1)}|\mathcal{S}|/n)^2 = (b - b/(d+1))^2|\mathcal{S}|^2/n^{2-3/(d+1)}$.

How do we avoid including the same pair twice in the output? Again we compute for each pair $\{s,t\}$ the leftmost shingle in the lexicographically preceding string. The algorithm is similar with the algorithm for subsequence only simpler and its cost is $O(b)$ per pair. Thus the total work in the reducers is $b(b - b/(d+1))^2|\mathcal{S}|^2/n^{2-3/(d+1)}$.

E. Datasets with Strings of Varying Length

In the two algorithms described above, the subsequence and the splitting algorithm, we gave the details for the case where all strings in the dataset are of equal length. We discuss briefly below the modifications for those algorithms when there are strings of varying length.

Now each reducer is characterized by two parameters, one corresponding to the length of the string under consideration and one to the substring (or subsequence respectively, depending which of the two algorithms we are modifying). The same argument that set the suitable length for substring (or subsequence) works again, only now we have $l_1 + l_2$ instead of $2b$. However the mapper considers only one string and it should decide on the basis that it does not know the other string, hence it does not know one of l_1 or l_2. But the bucketing into classes of lengths helps here. In particular we do the following: We construct intervals $[0, 2d], [d, 3d], [2d, 4d], [3d, 5d], \ldots, [jd, (j+1)d]$, where the last interval $[jd, (j+1)d]$ is such that the string of maximum length in the dataset falls in this interval. The hash is a pair (i, j), where i takes the values of the interval the length of the string belongs to (i.e., two values) and j is one of the substrings (or subsequences) of appropriate length L. L is calculated based on the length of the string and in particular on the interval this length belongs. For interval $[kd, (k+2)d]$, L is set to the same value as the case of equal length if we replace b by kd. The algorithm that allows us not to produce the same pair twice in the output works here without additional complications.

Now the map-cost remains the same, but for the other three costs, we need to sum up for each different interval by setting for each interval the value for b equal to kd (for the k-th interval). The resulting costs will be lower than the costs in the case all strings were of equal length equal to the maximum length in the dataset. If the strings in the dataset are such that in each interval we have a constant number of strings (say 10) then the communication cost and the processing cost in the reducers are both equal to $|\mathcal{S}|$ and the number of reducers is equal to the number of intervals multiplied by a small constant which is roughly again equal to $|\mathcal{S}|$.

F. Anchor-Points Algorithms

Before describing the anchor-points algorithm, we show a connection between the generic edit distance described above and the edit distance using only deletions as operations (called *edit distance by deletion*):

Proposition 5.4: If the edit distance from u to v is less than d, and the edit distance by deletion from u to c (the anchor point) is less than d, then the edit distance by deletion from v to c is less than $2d$.

Proof: Suppose a longest common subsequence of u and v is w and suppose that the distance from w to u is d_1 by deletion and the distance from w to v is d_2 by deletion. It is easy to see that $d_1 + d_2 = dist(u, v) < d$. A way to derive c from v is by deleting d_2 to get w then inserting d_1 to get u and deleting $d_3 < d$ to get c. Thus, in the worst case, we need to delete all the d_1 insertions and also do all the d_2+d_3 deletions in order to get c from v. Thus the distance by deletion from v to c is less than $d_1 + d_2 + d_3 < 2d$. ■

The above result allows us to use strings such as c above as anchors. We know that the anchor point is a string that can be reached by deletions only (as show in the claim above). We say that an anchor "covers" all strings formed from it by up to d insertions. If all strings are covered, and we send every string to all anchors that it can reach by up to $2d$ deletions, then all pairs of strings at edit-distance at most d will be sent to a at least one common anchor.

In analogy with the Hamming distance, we can compute the anchor point in two ways, either by generating all strings at distance by deletion at most $2d$ from the given string (that will cost $B^{del}(2d)$) or by considering all anchor points and check which is at distance at most $2d$ from it (that will cost $\frac{n}{B^{ins}(2d)}$. Thus the map-cost per element is the minimum of the two costs. The total number of reducers is $\frac{n}{B^{ins}(2d)}$. The total communication cost is $|\mathcal{S}|B^{del}(2d)$. Notice that we only need to use the size of the ball that results by deletions and this reduces the communication cost considerably. The total reducer cost is $|\mathcal{S}|B^{ins}(2d)$. Since the size of the ball that is produced by insertions only and the size of the ball that is produced by both deletions and insertions are the same order, in the table we will use $B(d)$ for $B^{ins}(d)$ but we will keep $B^{del}(d)$ since it is much smaller. When we compute the costs, whenever the B^{ins} appears in the denominator it actually gives an upper bound because according to what we also explain in the subsection below using again the Hungarian argument the number of anchor points is expected to be a bit larger than that.

1) Finding Good Sets of Anchor Points: Here we can again apply a Hungarian argument as in Section IV-D1. The calculations remain the same but notice that now we are considering a ball around a point that results from deletions only hence the final result is:

$$x > \frac{n \log n}{B^{del}(d)}$$

We have summarized the results on edit distance in Table III.

VI. Jaccard Similarity

Finally, we briefly consider the Jaccard similarity measure: Given two sets S_1, S_2, the jaccard similarity is given by $J_{1,2} = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$. The Jaccard distance between the same sets is $d_J = 1 - \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$.

We present a brief description for Jaccard distance here, with details in the appendix. The algorithms we propose for Jaccard distance are essentially the edit-distance algorithms applied to the sorted string representation of sets that comes from Chaudhuri et al. [17] and Xiao et al. [18].

VII. Analysis of Sorted-String Algorithms

In this section we analyze the performance of the algorithm proposed by Xiao et al. [18] in terms that best approximate the analyses we did for our proposed edit-distance algorithms. The conclusion is that for large universal sets, [18] is very good, but not so if the universal set is small. Recall that this algorithm uses sorted strings to represent sets.

Suppose that the limit on edit distance of these strings is d, and all strings are of length b. Let q be the size of the universal set, i.e., the size of the alphabet from which strings are constructed. The reducers correspond to the symbols representing an element of a set. Thus, there are q reducers. A string s is sent to $d+1$ reducers, those that correspond to the symbols appearing in the first $d+1$ positions of s. However, attached to s is an integer indicating the position in which the symbol appears.

If there are q reducers, then the average reducer receives $|\mathcal{S}|(d+1)/q$ strings. It does not have to compare all pairs of received strings; only those where the positions of the symbol associated with that reducer sum to at most $d+2$ are compared. However, that effect only cuts down the number of comparisons by a factor of 2 on the average. Thus, the total work of the reducers is $O(|\mathcal{S}|^2 d^2/q^2)$. If we compare with the estimates in Table III we see that this expression has a low value when q is large, but is higher than the costs associated with our proposed algorithms if q is small.

Unfortunately, there is another factor that makes this analysis of [18] too low, but for which we cannot offer an exact modification. Since the strings are sorted, the earliest symbols in the order can be expected to appear preferentially in the prefixes of strings. To what extent this is true depends on details of the population of strings that we cannot characterize easily. However, we do not expect the assumption of uniform distribution of data to reducers will hold. Rather, the reducers corresponding to early symbols in the order will get larger sets of strings to compare. Since the cost of comparisons is quadratic in the number of strings a reducer receives, this effect can be significant and argues against the algorithms based on [18].

Interestingly, the approach of [18] can be used for edit distance as well as Jaccard distance. In that case, the strings are not sorted, so the skew in the populations of strings at the reducers does not necessarily occur (but could still be present if certain characters were more popular, as "e" might be expected to occur more frequently than "z" in English text). However, edit-distance applications tend to have a small alphabet size, in which case the algorithms we propose would be preferable. In particular, our algorithms generate many more reducers than there are characters. Thus, even if there is skew, we have ample opportunity to group large numbers of "reducers" (i.e., key/value-list pairs) into a smaller number of Reduce tasks, or allow each reducer to be a Reduce task and distribute them evenly to a smaller number of compute nodes.

VIII. Conclusions

In this paper, we developed techniques for performing similarity joins in a single map-reduce step. We present algorithms for similarity joins based on the hamming distance, edit distance, and jaccard distance measures. We compare our algorithms based on three components: map cost, reducer cost, and communication cost. Interestingly, we show that there are multiple non-dominated algorithms, which enables applications to pick the most suitable application based on our cost analysis.

References

[1] C. Olston et. al., "Pig latin: a not-so-foreign language for data processing," in *SIGMOD '08*.

[2] A. Thusoo et. al., "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, August 2009.

[3] P. Koutris and D. Suciu, "Parallel evaluation of conjunctive queries," in *PODS '11*.

[4] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *WWW '11*.

[5] K. Morton, M. Balazinska, and D. Grossman, "Paratimer: a progress indicator for mapreduce dags," in *SIGMOD '10*.

[6] S. Lattanzi et. al., "Filtering: a method for solving graph problems in mapreduce," in *SPAA '11*.

Algorithm	Map-cost Per Element	#Reducers	Communication (C)	Processing (R)
Naive	J (approx. $\sqrt{K}$)	K (arbitrary)	$\lvert\mathcal{S}\rvert\sqrt{K}$	$\lvert\mathcal{S}\rvert^2$
Ball-hashing-1	$B(d)$	n	$\lvert\mathcal{S}\rvert B(d)$	$\lvert\mathcal{S}\rvert^2 B(d)/n$
Ball-hashing-2	$B(d/2)$	n	$\lvert\mathcal{S}\rvert B(d/2))$	$\lvert\mathcal{S}\rvert^2 \big(B(d/2)\big)^2 bd/n$
Subsequence	$\binom{b}{b-\frac{d}{2}}$	$q^{b-d/2}$	$\lvert\mathcal{S}\rvert\binom{b}{b-\frac{d}{2}}$	$\lvert\mathcal{S}\rvert^2\binom{b}{b-\frac{d}{2}}^2 q^{d/2-b}bd$
Splitting	$b-b/(d+1)$	$n^{1/(d+1)}$	$\lvert\mathcal{S}\rvert(b-b/(d+1))$	$b(b-b/(d+1))^2\lvert\mathcal{S}\rvert^2/n^{2-3/(d+1)}$
Anchor Points	$\min\big(B^{del}(2d), \frac{n}{B^{ins}(d)}\big)$	$\frac{n}{B^{ins}(d)}$	$\lvert\mathcal{S}\rvert\frac{B^{del}(2d)}{B^{ins}(d)}$	$\lvert\mathcal{S}\rvert B(d)/n$

TABLE III
SUMMARY OF THE MAP COST PER INPUT ELEMENT, NUMBER OF REDUCERS, COMMUNICATION COMPLEXITY, AND THE TOTAL REDUCER COST FOR VARIOUS EDIT-DISTANCE JOIN ALGORITHMS. ALL EXPRESSIONS OMIT A BIG-OH; I.E., THEY ARE ORDER-OF-MAGNITUDE. BY q WE DENOTE THE SIZE OF THE ALPHABET.

[7] F. Chierichetti, R. Kumar, and A. Tomkins, "Max-cover in map-reduce," in *WWW '10*.
[8] G. D. F. Morales, A. Gionis, and M. Sozio, "Social content matching in mapreduce," *PVLDB*, vol. 4, no. 7, 2011.
[9] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *EDBT*, 2010.
[10] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. e-book, 2010. [Online]. Available: http://i.stanford.edu/ ullman/mmds/
[11] H. J. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *SODA '10*.
[12] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *SPAA '11*.
[13] F. N. Afrati and J. D. Ullman, "A new computation model for cluster computing," Technical Report, December 2009.
[14] T. Ito, "An implementation of locality sensitive hashing with mapreduce," Tech. Rep. [Online]. Available: http://code.google.com/p/likelike/
[15] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *SIGMOD '10*.
[16] R. Baraglia, G. D. F. Morales, and C. Lucchese, "Document similarity self-join with mapreduce," in *ICDM '10*.
[17] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE '06*.
[18] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *WWW '08*, 2008.
[19] T. Elsayed, J. J. Lin, and D. W. Oard, "Pairwise document similarity in large collections with mapreduce," in *ACL '08*.
[20] A. Okcan and M. Riedewald, "Processing theta-joins using mapreduce," in *SIGMOD '11*.
[21] W. Cohen, P. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *IJCAI '03*.
[22] W. W. Peterson and E. J. Weldon, Jr., *Error-correcting Codes*. MIT, 1972.
[23] A. X. Liu, K. Shen, and E. Torng, "Large scale hamming distance query processing," in *ICDE '11*.
[24] A. Tietavainen, "On the nonexistence of perfect codes over finite fields," *SIAM-J-APPL-MATH*, vol. 24, no. 1, Jan. 1973.
[25] M. J. E. Golay, "Notes on digital coding," *Proc. IRE*, vol. 37, 1949.
[26] T. S. Baicheva et. al., "Binary and ternary linear quasi-perfect codes with small dimensions," *IEEE Transactions on Information Theory*, vol. 54, no. 9, 2008.
[27] J. W. Hunt and M. D. McIlroy, "An algorithm for differential file comparison," Technical Report, June 1976.

APPENDIX

A. A Lower Bound for Naive Algorithms

We can show that the algorithm described in Section III-C is, to within a constant factor, the best we can do without some way of avoiding comparing all pairs of strings at the reducers. Suppose that there are K reducers, and each one receives x strings. Since all $\binom{|\mathcal{S}|}{2}$ pairs of strings must be compared, and each reducer is only capable of comparing the $\binom{x}{2}$ pairs of strings that it has received, we must have

$$K\binom{x}{2} \geq \binom{|\mathcal{S}|}{2}$$

It follows easily that $x = \Omega(|\mathcal{S}|/\sqrt{K})$. Thus, $\binom{x}{2} = \Omega(|\mathcal{S}|^2/K)$. The latter expression is the work performed at each reducer. As there are K reducers, we have the total reduce cost $R = \Omega(|\mathcal{S}|^2)$.

B. Hamming Codes: details

In Section IV-C we mentioned the properties of Hamming codes. Here we give a short introduction to the Hamming codes, how they are constructed and how the decoding is done. For any $r \in \mathbb{N}$ a Hamming code exists of distance $b = 2^r - 1$. It is a linear perfect code. We start by considering an $r \times b$ *parity-check matrix* H: The columns of H consist of all non-zero vectors of length r (in any order). The set of all codewords of length b are given by all vectors v such that $H \cdot v = 0 \mod 2$. That is, the set of all codewords is given by:

$$\mathcal{C} = \{v | H \cdot v = 0 \mod 2\}$$

It is easy to see that the number of codewords is 2^k where $k = (2^r - r - 1)$. In fact, we can easily obtain the generator matrix G whose rows are the basis of the code subspace if we construct H such that $H = [I_r|M]$ where I is the identify matrix of size r and M is a $r \times (2^r - r - 1)$ matrix consisting of all the remaining non-unit vectors. Then, we can construct $G = [M^T|I_{2^r-r-1}]$, a $(2^r - r - 1) \times b$ matrix. The size of the code, i.e., the number of codewords $|\mathcal{C}|$

$$|\mathcal{C}| = 2^k = 2^{2^r-r-1}$$

A perfect code should satisfy the lower bound which indeed is the case for the code we constructed:

$$LB = \frac{2^b}{1 + \binom{b}{1}} = \frac{2^{2^r-1}}{1 + 2^r - 1} = 2^{2^r-r-1} = |\mathcal{C}|$$

The decoding is done as follows: Suppose we want to decode vector v. Then we multiply v by the matrix H. If $v = c + e$ where c a codeword and e a vector with all "0"s or with only one "1" and all "0", then $H \cdot v = H \cdot c + H \cdot e = H \cdot e$. Thus the result of the multiplication will be a vector which will be either zero or will be identical with some column of H, say the j-th column. Actually j is also the position of the "1" in e. Thus, we decide that the codeword which is closer to v is the one which is produced by flipping the j-th position of v.

The following theorem is a consequence of what we explained above.

Theorem A.1: For a given r, there is a Hamming code of length $b = 2^r - 1$ with a $r \times (2^r - 1) = \log b \times b$ parity-check matrix. The most expensive computation of the decoding algorithm is the multiplication of a vector with H, hence the processing cost of the decoding is $\log b \times b$. This multiplication creates a vector of length $\log b$ which is either all-0 (in which case the given vector is a codeword itself) or is identical with one of the columns of H.

Example A.2: As an example take $r = 3$. Then the parity-check matrix H is:
$$\begin{matrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{matrix}$$
This matrix is a 3×7 matrix and creates a code of length $2^3 - 1 = 7$.

Suppose we take the vector $v = [1101000]$. If we multiply this vector (viewed as a column) by H we get:

$H \cdot v = 0$

Observe that this multiplication is equivalent to adding up the the first column of H with its second column with its fourth column (because the "1"s in v appear in the first, second and fourth position). Thus we conclude that v is a codeword. Now let us take $v' = [1100000]$. Now we need to add the first and the second column of H getting:

$H \cdot v' = [110]$

Again the $[110]$ is actually a column vector. Now, we observe that this vector is identical to the fourth column of H. Hence the closer codeword to v' is derived if we flip the digit in the fourth position of v' and is $v = [1101000]$.

C. Perfect codes

The problem of finding a good set of anchor points under Hamming distance has been investigated in Coding Theory, since such sets are also good error-correcting codes. In this subsection we present what is known and how we can take advantage of it.

An (n, k, t) *binary code* is a set of k codewords, which are $(0, 1)$-vectors of length n and have the property that the balls of radius t around each codeword are disjoint. A code (like the Hamming code in Section IV-C) where its set of codewords cover the whole space of words is called *perfect*. A *quasi-perfect* code is one in which the balls of radius t centred on codewords are disjoint and the balls of radius $t + 1$ cover the space, possibly with some overlaps.

A complete classification of the parameters for which perfect codes over Galois fields [22] exist is available since the early 1970's (see e.g., [24]). The nontrivial perfect codes are very few:

- the Hamming codes (with parameters as in previous section)
- the binary (23, 2048, 7) Golay code [25] and the ternary (11, 729, 5) Golay code.

For quasi-perfect codes, we don't have a complete characterization. However, it appears that, like perfect codes, quasi-perfect codes exist only for small radii. For covering radius more than 3 only few non-trivial examples of binary quasi-perfect codes exist (see e.g., [26]).

D. Finding Good Sets of Anchor Points: Details

Here we provide details for our short discussion in Section IV-D1 on finding good sets of anchor points. Suppose we have x codewords (anchor points), and each string needs to be within distance d from some anchor point. Say that an anchor point *covers* string s if it is within distance d of s. With $n = 2^b$, there are $\binom{n}{x}$ ways of picking x anchor points. Some of these will fail to satisfy the condition that every string is covered by some anchor point. Focus on a particular string s. If a selection of x anchor points fails to cover s, then the selection of anchor points must be made from the strings outside the ball of radius d around s. The number of selections of x anchor points that do not cover s is $\binom{n-B(d)}{x}$. Since there are n possible strings s, the total number of selections of x anchor points that *fail* to cover some string is at most $n\binom{n-B(d)}{x}$. As long as this quantity is strictly less than $\binom{n}{x}$, then there must be some selection of x anchor points that covers every string. That is, we know there is a set of x anchor points that covers every string provided

$$n\binom{n - B(d)}{x} < \binom{n}{x} \tag{1}$$

We shall assume that x is small compared with n or $n - B(d)$. If so, then $\binom{n-B(d)}{x}$ is approximately

$$\big(n - B(d)\big)^x / x!$$

and $\binom{n}{x}$ is approximately $n^x/x!$. On this assumption, Equation 1 becomes

$$x \log n > \log n + x \log\big(n - B(d)\big)$$

or

$$x > \frac{\log n}{\log \frac{n}{n-B(d)}}$$

where the logarithms are natural logs.

We shall further assume that $B(d)$ is small compared with n. If so, then $\log \frac{n}{n-B(d)}$ is approximately $B(d)/n$, whereupon Equation 1 holds as long as

$$x > \frac{n \log n}{B(d)}$$

Thus, the number of anchor points needed is only a little more (a factor of $\log n$, or b) than what would be needed if there were a perfect coding (which would have given $x = \frac{n}{B(d)}$).

The above argument is only an existence proof. However, if we double the left side of Equation 1:

$$2n\binom{n - B(d)}{x} < \binom{n}{x}$$

then we can be certain that a randomly chosen set of x points has a 50% probability of covering every point. The above is satisfied provided $x > \frac{n \log 2n}{B(d)}$. As long as we pick x to satisfy the above, we can test random selections of anchor points, and the expected number of selections we must test is two. In fact, we could multiply the left side of 1 by 1000, which would only require that $x > \frac{n \log 1000n}{B(d)}$ (i.e., a small fraction more than the minimum x implied by Equation 1 when n is much larger than 1000), and know that a random selection of x points covered all points with probability .999 without testing.

E. Upper Bounds on B^{ins} and B for Edit-Distance

The formula for B^{ins} is somewhat tricky. Let us begin by considering how many different strings we can obtain by inserting exactly i symbols from an alphabet of size q into a string s of length l. Notice that it is never necessary to insert a symbol x immediately to the left of another x. We could instead insert the same symbol immediately to the right of a run of x's. Thus, if we insert anywhere but at the right end of s, we should choose from only $q-1$ of the q symbols. If we make this restriction, then every string that results from s by i insertions can be constructed in a unique way. To count the number of these strings, begin by observing that the resulting string t has $l+i$ positions, of which i are inserted positions. These positions can be ordered in $\binom{l+i}{i}$ ways. However, we must distinguish between inserted positions at the end of t and those that have at least one original position to their right.

The string t will have a *tail* of j inserted positions at the right end, where $i \geq j \geq 0$. These positions can be filled with any of the q possible symbols. Positions not in the tail can only be filled with one of $q-1$ symbols, all symbols of the alphabet except for the symbol in the position that follows, regardless of whether the following position is original or inserted. Thus, the number of possible strings t that can be formed by inserting i symbols, with j of those symbols in the tail is $\binom{l+i-j}{i-j}q^j(q-1)^{i-j}$.

If we sum this count for $i = 1, 2, \ldots, d$ and $j = 0, 1, \ldots, i$, then we get the expression for B^{ins}:

$$B^{ins}(d,l) = \sum_{i=1}^{d}\sum_{j=0}^{i}\binom{l+i-j}{i-j}q^j(q-1)^{i-j}$$

Notice that this quantity is always larger than $B^{del}(d,l)$, and except for the case $q=2$, it is much larger. The largest term of the sum is $\binom{l+d}{d}q^d$, and as long as d is small compared with l, that one term is, to within a constant factor, an approximation for the whole sum.

The value of $B_s(d,l)$, involving combinations of insertions and deletions is naturally larger than $B^{ins}(d,l)$. However, since there are always more possible insertions than deletions on any given string, the order of magnitude of $B_s(d,l)$ and $B^{ins}(d,l)$ are the same. We can get an upper bound on $B(d,l)$ for any string s by considering d "markers" representing edit operations interspersed with the l positions of the string s. There are $\binom{l+d}{d}$ ways to order the original symbols and markers.

Markers can be replaced by one of the q symbols of the alphabet or by "delete." The effect of the latter is to delete the following symbol of the string if it was one of the original symbols. If the following symbol was inserted or another "delete," then "delete" has no effect. Note that this policy lets us obtain any string that can be formed by d or fewer edits.

A marker that is part of the tail of markers at the right end can be replaced by any of the q alphabet symbols. However, as discussed above in connection with counting B^{ins}, a marker that is followed by an original symbol of s or an inserted symbol need not be replaced by that same symbol, since a run of x's can always be increased by insertions at the right end. Thus, markers not in the tail can also be replaced in q ways: either "delete" or one of $q-1$ alphabet symbols. We have thus proved that

Proposition A.3: $B(d,l) \leq \binom{l+d}{d}q^d q^d$.

F. Finding Leftmost Subsequences

The following is a simple variant of the Hunt-McIlroy algorithm [27], which is the one implemented in the UNIX command `diff`. Suppose we have two strings s and t with a common subsequence u of length l. We want to check whether there is any other subsequence of the same length that is "to the left" of u with respect to string s. That is, each common subsequence corresponds to an increasing list of positions in both s and t. We can compare two such sequences of positions lexicographically, and the first is said to be *to the left*. More formally, if the two sequences are $p_1, p_2, \ldots$ and $q_1, q_2, \ldots$, then the first is to the left of the second if for some $i \geq 1$, we have $p_j = q_j$ for $0 \leq j < i$, while $p_i < q_i$.

Example A.4: Let $s = acba$ and $t = abca$. There are two longest common subsequences aca and aba. The former is to the left, since its position list in s is $[1,2,4]$ while the position list for aba in s is $[1,3,4]$.

To find the leftmost subsequence whose length is u, we construct a matrix $M[i,j]$, where i ranges from 1 up to $|t|$ and j ranges from 0 up to $|t|-|u|$. The value of $M[i,j]$ is the leftmost list of positions of s that forms a subsequence of length $i-j$ with the first i symbols of t. The value is undefined if there is no sequence of positions meeting the requirements.

To start, let $M[1,1]$ be the empty list and $M[1,0]$ be the leftmost position of s that agrees with position 1 of t (undefined if no such position exists). For the induction, suppose we have computed M for first arguments up to i, and let us compute the entries for fist argument $i+1$. The list for $M[i+1,j]$ can be constructed in two possible ways. Assume the symbol in position $i+1$ of t is x.

1) Take the list $M[i,j]$ and concatenate it with the first position of s after the end of this list that holds x.
2) Take the list $M[i,j-1]$ and do not match x against any position of s.

Example A.5: Let $s = cbacbbbc\cdots$ and $t = abcabcb\cdots$. $M[6,3]$ is $[1,2,4]$, since cbc is the leftmost common subsequence of length $6-3=3$ between s and the first six positions of t. $M[6,2]$ is $[1,2,5,8]$, since $cbbc$ is the leftmost sequence of length $6-2=4$ between s and the first six positions of t.

The seventh position of t holds b. To form $M[7,3]$ we can either copy $M[6,2] = [1,2,5,8]$ or start with $M[6,3] = [1,2,4]$ and append the leftmost position of s that follows position 4 (the last position of the list $M[6,3]$) and holds b. That position is position 5, so the second option gives us the list $[1,2,4,5]$. Since the letter precedes $[1,2,5,8]$ lexicographically, we set $M[7,3] = [1,2,4,5]$.

Notice that $M[i,j]$ is always identical to or to the left of $M[i,j-1]$. For if not, then we could use the first $i-j$ positions of the list $M[i,j-1]$ as $M[i,j]$. Thus, if $M[i,j]$ is strictly to the left of the first $i-j$ positions of $M[i,j-1]$ we always choose option (1) unless there is no later position of s that matches x. If the last position of $M[i,j]$ equals the next-to-last position of $M[i,j-1]$ (in which case it is easy to prove that $M[i,j]$ equals the first $i-j$ positions of $M[i,j-1]$), then we pick option (1) if and only if the next position of s that holds symbol x is to the left of the last position of $M[i,j-1]$.

After computing $M[|t|, |t| - |u|]$, we examine this list. If in s these positions form subsequence u, then u is the leftmost subsequence of its length within s, and the reducer doing this calculation emits the pair $\{s, t\}$. Otherwise, this reducer does not output this pair, knowing the reducer associated with the leftmost subsequence will output the pair instead.

The running time of this algorithm on strings of length b with subsequences of length $b - d/2$ is $O(bd)$. To argue the case we must show how to compute each element of M in $O(1)$ time, since there are only $bd/2$ elements that must be computed. That, in turn, requires we set up some data structures, after which the $O(1)$ work per element claim should be clear.

a) We must index the symbols of s. There is a list for each symbol that appears one or more times, and a pointer a position on the list for each symbol x is accessible in $O(1)$ time via the index, given x. The total length of all these lists is $|s| = b$. For each symbol x, its pointer indicates the list entry corresponding to the first unexamined position holding x. When we need to find the first position of s that holds x and is to the right of some given position p, we find the pointer for x and advance it down the list until we find a position greater than p or reach the end of the list (in which case no such p exists). As the sum of the lengths of the lists is b, these searches take time $O(b)$ total.
b) We cannot represent $M[i, j]$ by the complete list, or copying lists would become too expensive. However, we can represent $M[i, j]$ by the last position on the list and a pointer to a place where the first $i - j - 1$ positions of the list are represented. To copy a list, simply copy its position and pointer. To append a position p to a list L and thereby form form $M[i, j]$, store as the value of $M[i, j]$ the position p and a pointer to a place where L is represented (in practice, that would always be the entry $M[i - 1, j]$).

G. Jaccard Distance

For this section we assume that the elements of the sets are drawn from a universe of q distinct elements and that each element can appear at most once. (Results can be extended easily for the case where elements may appear multiple times, up to a maximum cardinality of b.)

1) Mapping Jaccard Distance to Hamming Distance: We compute the size of the ball of radius d around a set S Suppose the threshold on the Jaccard distance is d. Then the Jaccard similarity threshold is $J = 1 - d$. Suppose we have a set S of size p. Recall we have q distinct elements in our universe. Let S' be a set with intersection with S of size i, i.e., $|S \cap S'| = i$. Then S' is at Jaccard distance less than d from S if the assymetric difference $S' - S$ is of size at most $i/J - p$. The number of different such sets S' is:

$$B_i = \binom{p}{i} \sum_{x=1}^{\lceil i/J - p \rceil} \binom{q-p}{x}$$

if $i/J - p \geq 1$ and B_i is equal to 0 otherwise.

Now the size of the ball is $\sum_{i=1}^{p} B_i$.

Below, first we present a simple algorithm and then we present algorithms that exploit the balls or certain radius around a point in a similar way as in previous sections for Hamming and edit distance.

2) Simple algorithm: We pre-define a total order on all the q elements. We can now represent each set as a string (over an alphabet of size q), by listing its elements in sorted order. These string representations of sets satisfy the following salient properties:

1) No character appears more than once in any string.
2) If two characters appear in two different strings, then they appear in the same order in both strings.

Next we show how the string representations can be exploited for the similarity join: Consider sorting (or bucketing) the set of strings based on length. Then, for each string s, we will compare it with all strings t that follow s in the sorted set of strings; but we can restrict the set of strings to compare s with based on the length of s: We only consider strings t that are not much longer than s, as shown below.

Suppose the required lower bound on the Jaccard similarity between the original sets is J. Let $L(x)$ denote the length of any string x. In our approach $L(s) \leq L(t)$. Since the intersection of the sets corresponding to s and t have at most $L(s)$ elements in common, and at least $L(t)$ elements in the union, we have an upper bound on their Jaccard similarity: $J_{s,t} \leq \frac{L(s)}{L(t)}$.¶ Therefore, we only consider strings t with $L(t) \leq \frac{L(s)}{J}$.

Using the bucketing technique described above, we consider pairs of strings which satisfy the length requirement; this is achieved by a map-phase that groups strings into buckets, with each string being mapped to buckets where it must be compared with other strings. Thereafter, we compare the strings, and if their intersection is above the required threshold, we output the pairs.

¶Note that this lower bound is appropriately revised if each element may appear up to b times in each set.