M.S. THESIS

PLF-Join: An Efficient MapReduce Algorithm for Vector Similarity Join

PLF-Join: 벡터 유사 조인을 위한 효율적인 맵리듀스 알고리즘

FEBRUARY 2015

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hyun Joon Kim

PLF-Join: An Efficient MapReduce Algorithm for Vector Similarity Join

PLF-Join: 벡터 유사 조인을 위한 효율적인 맵리듀스 알고리즘

지도교수 이상구

이 논문을 공학석사학위논문으로 제출함

2014 년 12월

서울대학교 대학원

전기.컴퓨터공학부

김현준

김현준의 석사학위논문을 인준함

2014 년 12 월

Abstract

PLF-Join: An Efficient MapReduce Algorithm for Vector Similarity Join

Hyun Joon Kim
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

Vector similarity join is a problem of finding all pairs of vectors which has a similarity measure that exceeds a given threshold from a set of vectors. *Vector similarity join* is used in many applications such as near duplication detection in web pages, recommendation, and mining social data. However, it requires $O(n^2)$ complexity where n is the number of vectors. This impractical time complexity makes it hard to utilize *Vector similarity join* on many real world problems. Hence, a lot of the Hadoop MapReduce algorithms were proposed to quickly compute *Vector similarity join.* The state-of-the-art algorithm considers prefix filtering and length filtering methods to reduce the time taken for *Vector similarity join* operation. To even further reduce this time complexity, we propose a variation of an algorithm that can be used to reduce the overhead involved in the network I/O cost. Along with a MapReduce algorithm we propose an efficient pre-processing technique which facilitates *Vector similarity join* calculation.

Keywords: Vector Similarity Join, Prefix Filtering, Length Filtering, All-Pair

Similarity Search, Standard Deviation

Student Number: 2012-23207

Contents

List of Figures

Chapter 1

Introduction

Vector similarity join is a problem of finding all pairs of vectors which exceeds a given threshold from a set of vectors. Similarity join is used in many applications such as near duplication detection in web pages, recommendation, and mining social data. Essentially the problem requires $N(N-1)/2$ pairs examined where N is the number of vectors. Since all of the dimensions need to be checked to see if each pair is similar, total complexity grows to $D * N(N-1)/2$ where D is the number of dimensions in the dataset. To resolve this time complexity issue, a lot of algorithms were proposed. There are many approaches in solving similarity join problem. One of the approaches is the filtering technique. By filtering it means to filter out dissimilar pairs without actually calculating the similarity value. For instance, recent works such as PPJoin+ and PNJoin works with combination of prefix filtering, length filtering and suffix filtering. These methods are all part of filtering techniques and they do their best to reduce the number of candidate pairs from naïve algorithm's $O(n^2)$.

Prefix filtering is one of the methods used most widely in similarity join algorithms. It exploits the fact that for a given threshold, to find another vector which is similar above that threshold, we do not have to look at all the elements. Using this fact we can create a partial inverted index and calculate similarities by comparing vectors that are gathered by the vector's dimensions. Details of the algorithm will be explained in Chapter 2.

Length filtering is another algorithm proposed by [2]. It is applied to prefix filtering technique to even further reduce the number of candidate pairs. Since all vectors have length, we can normalize all the vectors and compare their values. By this way we can drive a formula which uses length information and prefix element information to get better discriminative power.

Since the advent of the MapReduce framework, many algorithms tried to find a way to efficiently solve similarity join with MapReduce. VCL[4], V-SMART[5], and Bjoin[8, 3] are the recent researches in similarity join in MapReduce framework. VCL exploits prefix filtering technique. Since MapReduce gathers data by distinct key value, VCL algorithm uses dimension of the vectors as key and send all the vector elements to that dimension and do join algorithm within same dimension. V-SMART is another algorithm that uses inverted index for similarity join. It does not use any filtering technique, simply gathering all the vectors to its' dimensions where each element is non-zero value. Lastly, Bjoin is an algorithm that uses both prefix filtering and length filtering. It uses prefix filtering to first filter out the candidate pairs and gathers each vector to respective dimensions up to prefix. Then by using length filtering it further filters out the candidate pairs. After that the algorithm finally joins candidate pairs and checks if indeed the candidate pairs have similarity value that is higher than given threshold. Bjoin is a good algorithm that uses both prefix and length filtering, but it suffers from its base design of requiring network cost in $O(p^2)$ where p is a number of prefix elements in a vector.

In this paper we propose an algorithm that is based on Prefix and Length Filtering called PLF-Join. We efficiently halved network overhead of Bjoin from $O(p^2)$ to $\frac{O(p^2)}{2}$. Eventually leading to less filtering of dissimilar pairs. We will explain our algorithms in Chapter 5. This may seem small improvements in the mathematical terms, but in practical experiments, the performance difference is big. We will show the experimental results in Chapter 6

Chapter 2

Preliminary

In this section we explain the basic prefix and length filtering techniques. Section 2.2 refers to the contents from Bjoin[3] paper.

2.1 Problem Definition

According to definition from [9], Vector Similarity Measure has four properties. Positivity, Self-similarity, Maximality and Symmetry. In this paper we assume non-negative vectors and cosine similarity measure.

Definition 1 ***Vector Similarity Join*** *With given a number of vectors* $\mathcal{V}$ *and a threshold value* t*, vector similarity join is a problem of finding all vector pairs* x, y *such that* $x, y \in \mathcal{V}$ *where a similarity measure between them* $Sim(x, y) \geq t$

The definition 1 states the formal definition of vector similarity join problem. As we explained earlier, the time complexity of naive vector similarity join is $O(N^2)$. Instead of this naive algorithm, if we use inverted index join the time complexity becomes $O(N^2 * D)$ where N is the number of vectors and D is the number of dimensions. It looks like the time complexity increases, but since usually the real world datasets are sparse and not all vectors have same distribution of dimensions, actually it can be used to reduce the real time taken for total calculation. Addition to this, if we use prefix and length filtering we can further reduce the time taken for the vector similarity join.

Definition 1 is a general definition for different similarity measures. We can use any of measures such as Cosine, Dice, Tanimoto, and Jaccard. In this paper we will use Cosine similarity measure.

$$\begin{aligned} Sim(x,y) &\geq t \\ CosSim(x,y) = \frac{D(x,y)}{\|x\|\|y\|} &\geq t \end{aligned} \tag{2.1}$$

Equation 2.1 shows the objective function in this paper. From Definition 1, $Sim(x,y) \geq t$ was the objective function and since we are assuming cosine similarity measure in this paper, it can be rewritten as $CosSim(x,y) = \frac{D(x,y)}{\|x\|\|y\|} \geq t$

2.2 Filtering Predicate

We assume all vectors are normalized before we calculate the similarity. If the vectors are normalized we can calculate the cosine similarity as (2.2).

$$\frac{D(x,y)}{\|x\|\|y\|} = D(x,y) \tag{2.2}$$

$\|x\|$ denotes the length of a vector x. Since this is 1 in normalized vectors, $\|x\|\|y\|$ simply becomes 1 in Equation (2.2). $D(x,y)$ denotes the dot product of vector x and y. Dot product is a commutative function. So we can decompose $D(x,y)$ into smaller parts. In other words, $D(x_{p+s}, y_{p+s}) = D(x_p, y_p) + D(x_s, y_s)$. Using this property we can set a certain dimension as a point where we break a vector into two parts. In our algorithm this will be determined by the given threshold t and we will call this point as a $prefix$ *point* and denote it as p_i. Subscript i is denoting vector i. x_p denotes the prefix portion of the vector x. For example, in Figure 2.1, we see that there are two

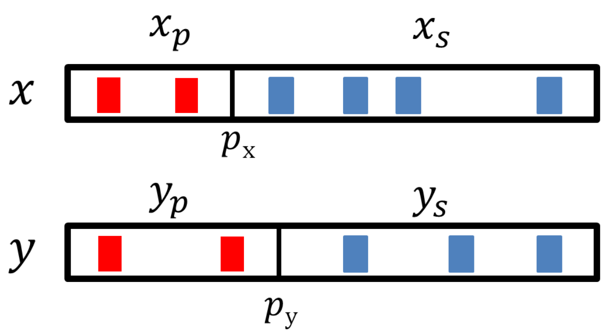


Figure 2.1: Filtering terms

non-zero elements in the prefix of vector x. x_s is a suffix part of vector x. Suffix is all the non-zero elements that come after the prefix point. For the Figure 2.1, x_s has four elements.

2.2.1 Prefix Filtering

Prefix Filtering utilizes an inverted index and the given threshold to reduce the candidates. In inverted index join, we need to calculate all the elements inside the vector. For example, in the Figure 2.1, we have total 6 non-zero elements for vector x and 5 non-zero elements for vector y. For inverted index join, we need to invert all 6 elements for x and 5 elements for y to dimensions and then calculate the similarity. This incurs too much time and network overhead that sometimes can take longer than the time needed for naive pairwise join.

On the other hand, given a threshold, prefix filtering can reduce the number of elements in the vector to be processed. In the Figure 2.1, the given threshold is 0.9. So we need to find the length of the prefix of the vector that exceeds 0.1. In both x and y, length of the prefix after 2 elements exceeds 0.1 ($\|x_p\| > 0.1$). This means we have to consider two elements before the prefix point p_x and we don't need to care about what comes after the prefix point p_x. Because the suffix lengths of the vectors are less than or equal to 0.9 ($\|x_s\| \leq 0.9$). So in prefix filtering, if any non-zero elements' dimension overlaps with other vector, the pair becomes a candidate pair. In order to safely conclude that the pair is dissimilar, if nothing matches before the prefix point, we will show that $D(x_s, y) < t$ in such case.

$$\begin{aligned} \frac{\|x_s\|}{\|x\|} &< t \\ \Rightarrow \|x_s\| &< t\|x\| \\ \Rightarrow \|x_s\|\|y\| &< t\|x\|\|y\| \\ \Rightarrow D(x_s, y) \leq \|x_s\|\|y\| &< t\|x\|\|y\| \\ \Rightarrow D(x_s, y) &< t\|x\|\|y\| \\ \Rightarrow D(x_s, y) &< t \end{aligned} \tag{2.3}$$

Note that at the first line of Equation 2.3, we set the length of prefix point part greater than the threshold that is why the suffix length divided by whole vector length is less than the given threshold. The fourth line comes from Cauchy-Schwarz inequality. The fifth line's lengths of two vectors becomes 1 because we are assuming normalized vectors. So we concluded that if we set the length of prefix point larger than the threshold, dot product of the vector's suffix part and any other vector will not be greater than t. VCL algorithm uses this fact to filter out dissimilar pairs. However this technique alone, can not differentiate the vectors that share same prefix part but are not similar. For example, consider we have two vectors with same one prefix dimension but with different magnitudes. Then even they have non-zero elements at same dimension that differs greatly in magnitude that will make them dissimilar, they will pass this prefix filtering. However, this can be dealt with the length filtering technique. We will describe this in the next section.

2.2.2 Length Filtering

Length Filtering is a technique proposed by [2]. This technique adds on to the prefix filtering method and even further discriminates the candidate pairs. In prefix filtering we found out that if we make prefix point greater than the threshold, no matter how much overlap happens at the suffix of the vector, if nothing overlaps at the prefix part, the similarity value will be below the threshold. So in length filtering we will even examine these candidate pairs that passed the prefix filtering predicate and filter them. Length filtering utilizes an inequality from equation 2.4.

$$D(x_s, y_s) \leq \|x_s\|\|y_s\| \tag{2.4}$$

The equation 2.4 comes from Cauchy-Schwarz inequality to set an upper bound. So if the sum of $D(x_p, y_p)$ and $\|x_s\|\|y_s\|$ is less than a threshold t, then $Cos(x, y) \leq t$. Equation 2.5 is the length filtering predicate equation. We can see that the first line is the approximation of the second line which is cosine similarity value of vectors x and y. Since representing length of a vector requires much less data compared to the actual vector elements, we can reduce data required for the filtering by this approximation.

$$\begin{aligned} D(x_p, y_p) + \|x_s\|\|y_s\| &< t\|x\|\|y\| \\ \Rightarrow D(x_p, y_p) + D(x_s, y_s) &< t\|x\|\|y\| \\ \Rightarrow D(x, y) &< t\|x\|\|y\| \\ \Rightarrow Cos(x, y) &< t \\ &\Rightarrow DissimilarPair \end{aligned} \tag{2.5}$$

There is one important thing to note in the prefix filtering and length filtering. That is the prefix point p does not necessarily mean same thing between these two filtering techniques. In prefix filtering, p means that all dimensions smaller than p should be checked to judge whether the candidate pair is similar or not. So p should satisfy first line of the Equation 2.3.

On the other hand, the prefix point p of length filtering should not follow this condition. Every p from length filtering should follow Equation 2.5. Vectors have different prefix points because each vector has different dimensions and

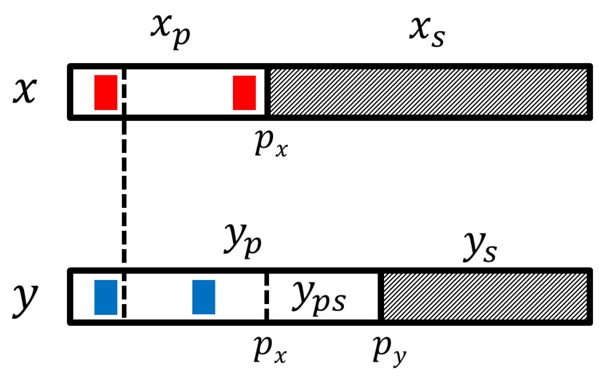


Figure 2.2: Length Filtering

values. In Figure 2.2, we see different prefix points for vectors x and y, $p_x < p_y$. In order to apply length filtering predicate, we need to extend y_s so that at least it covers y_{ps} part. This is because we don't know how vector x will match with y_{ps}. In some cases, this part might contain the very similar object which contains large portion of the vector length. But if we don't include y_{ps} to the suffix length of y, the pair would be considered not similar. So we need to be sure to cover these cases. If we have information about the value of y_{ps}, then we can exactly pin point the location of dimension p_x on y. So when we make $\|y_s\|$ to include $\|y_{ps}\|$, we can use length filtering predicate in Equation 2.5.

Chapter 3

Related Works

3.1 V-SMART Algorithm

V-SMART algorithm is proposed at VLDB 2012, which is earlier than VCL algorithm which is discussed at Section 3.2. However, since this algorithm has different characteristics from the other algorithms, we will introduce V-SMART algorithm first. The algorithm uses the fact that a lot of similarity measures are composed of Unilateral and Conjunctive functions of vectors. With this fact the authors of the algorithm proposed an algorithm that makes a virtual inverted index and calculates actual similarity value of all pairs of vectors on Hadoop MapReduce framework.

V-SMART is rather a simple algorithm. It is consisted of two MapReduce jobs. The first job is responsible for building virtual inverted index and send each value of vectors to the respective dimensions. The second job gathers pairs of vector ids and calculates actual similarity value of pairs from inverted index for confirming similar pairs. For example, one key for the second job would be $< x, y >$ and the values would be dot product of elements that share same dimensions between vectors x and y.

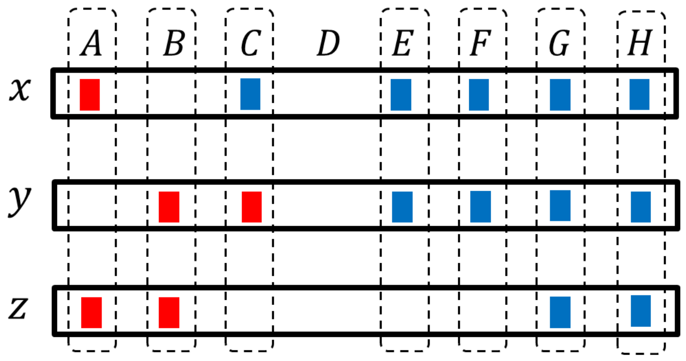


Figure 3.1: An example for V-SMART algorithm

Refer to the Figure 3.1, x, y, z denotes the vectors and A to H denotes the dimensions for each vector. Note that some places don't have any boxes. This indicates the dimensions contain zero value element for that particular vector. For the first job's Mapper, V-SMART algorithm outputs *Unilateral* values, vector id, and dimension value to each non-zero element appears in the vectors. These values gather at the first Reducer. Then the Reducer generates vector pair (e.g. $< x, y >$) and sends previous *Unilateral* values and vector information. In our example, note that there are 11 pairs of vectors generated. For the second job's Mapper, does not do anything and it is called *Identity-Mapper*. At the second job's Reducer, each vector elements are gathered at the pairs. So the algorithm can calculate the similarity value of two vectors.

V-SMART suffers from the performance because the overhead of making the inverted index is a lot. This incurs a lot of I/O cost because typically MapReduce spill out all data to the disk between jobs. The most inefficient part is $Reduce_{First}$, because the number of outputs is proportional to the square of the number of inputs in worst case. Also unlike filtering algorithms, performance of *V-SMART* algorithm does not change for varying threshold. Because *V-SMART* is not filtering based and it simply creates the inverted index of vectors, it is incapable of elastically respond to different threshold values. This can be critical issue in some applications such as duplicate detection. Duplicate detection requires fairly high threshold for the data cleansing purpose. However, *V-SMART* can't efficiently handle the high threshold unlike the other filtering based algorithms. Our experiments in Chapter 6 show that *V-SMART* algorithm is good for only low threshold.

3.2 VCL Algorithm

In this section we describe another vector similarity join algorithm proposed by Vernica at el. *VCL* which is proposed at SIGMOD 2010 [4]. *VCL* is an algorithm which uses prefix filtering predicate we described in 2.2.1. It generates partial virtual indexes for prefixes rather than the entire vectors. The vectors gather at the prefix dimensions so that the matching vectors in the dimensions can calculate similarity value. *VCL* assumes set similarity filtering and it uses Jaccard similarity for similarity measure.

Similar to the Equation 2.3, the algorithm needs to check at least the prefix of the vectors. If there is a match in the prefix of the vector, the vectors would not be filtered out. If there is no match, $Jac(x, y)$ cannot exceed a threshold, which means they are not similar.

How *VCL* algorithm works is as follows. In Figure 3.2, blocks before the bold line represent prefix part of the vector and blocks after the bold line represent suffix part of the vector. Only prefixes, A, B, and C are virtually indexed. For example, *V-SMART* generates 11 pairs (3 distinct pairs) from 4 dimensions in Figure 3.1. *VCL* generates only 2 pairs from 3 dimensions in Figure 3.2. *VCL* usually performs better in most conditions because it generates much less virtual indexes and conducts the filtering technique. Also it does perform well with the high threshold values.

First job's Mapper generates vector ID, x, and vector data, x_{Data}, for each non-zero elements of the prefix of vector. First job's Reducer can get vector IDs that have common non-zero elements in the prefixes and calculate similarities of pairs using x_{Data} came with x. The second job just removes duplicated pairs

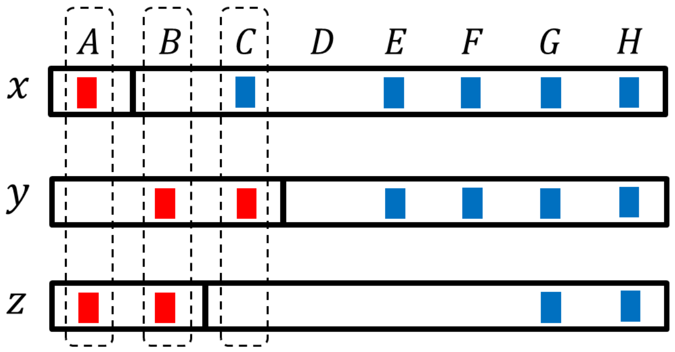


Figure 3.2: An example for VCL algorithm

from the similar pairs result of first job. *VCL* is a simple algorithm that works well with the MapReduce system. Since it requires only two jobs for the similarity calculation, it minimizes the intermediate data in MapReduce system. Since MapReduce requires all intermediate data to be spilled to the persistent storage, this often incurs much disk I/O.

However, *VCL* has some inefficiency in vector data duplication. Since it only uses prefix filtering, many vectors do pass this filtering technique, eventually leading to a lot of network I/O with vectors that are actually not similar. When first job's Mapper of *VCL* reads a vector, it outputs $\langle x, x_{Data} \rangle$ for every prefix elements. For instance, assume that a vector x has 100 elements, and the prefix has 20 elements. *VCL* needs to replicate the vector x as many as the number of prefix elements. *VCL* copies the vector with 100 elements by 20 times and produces 2,000 elements. This is significantly more overhead compared to *Bjoin* algorithm.

3.3 Bjoin Algorithm

Bjoin algorithm was proposed in [3]. The paper extends the length filtering method proposed in [2] and proposes a MapReduce algorithm. The length filtering method efficiently filters dissimilar pairs.

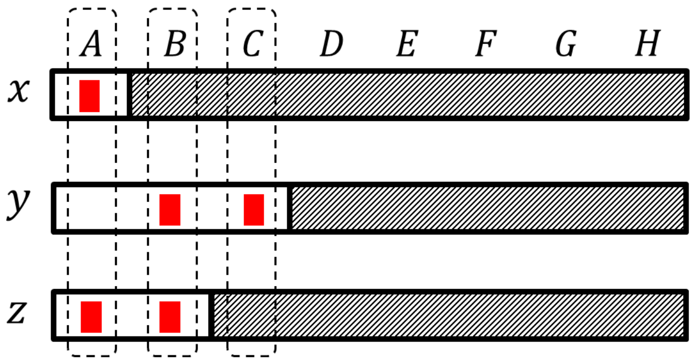


Figure 3.3: An example for Bjoin algorithm

Bjoin works as follows. In the Figure 3.3, rectangles with diagonal lines represent the suffix part of the vectors. Like *VCL* algorithm Bjoin also virtually indexes A, B, and C dimensions. However, unlike *VCL* algorithm, *Bjoin* does not index the whole vector but only indexes prefix part of the vector to reduce the network overhead. After creating the index, using the length filtering method presented in 2.2.2, Bjoin algorithm filters out the candidate pairs which passed the prefix filtering. This results in reducing the candidate pairs even further than prefix filtering. After this process, the algorithm needs to check if the candidate pairs which passed the prefix and length filtering actually have similarity measure higher than the threshold. So the algorithm actually performs the join operation for the pairs and discard pairs that do not have similarity value above the threshold.

Bjoin's first job's Mapper generates vector ID, x, prefix part of the vector, and length of suffix part of the vector for each non-zero elements of the prefix of vector. First job's Reducer receives vector IDs that non-zero element is in the same dimension of the prefix. From here, Bjoin algorithm applies length filtering method and using the information about length of the suffix and prefix elements. Using Equation 2.5, the algorithm filters out the dissimilar pairs. After this job, both prefix and length filtering techniques are applied and we have a list of candidate pairs. However, the list does contain some duplicate records, so *Bjoin* removes these duplicate records with a one more MapReduce job. The third job needs to access the original input vector data again. Since MapReduce framework is not capable of holding the data between the stages, *Bjoin* re-accesses the input data from Hadoop File System and match it with the previous job's output (Candidate pair list). The fourth job is responsible for the actual calculation of the similarity pairs. If their similarity values exceed the given threshold value, *Bjoin* outputs the value and the pairs.

Bjoin achieves much better filtering power because it is not only using prefix filtering, but also length filtering. However, the extension requires two more additional MapReduce jobs which incurs much overhead. Basically, Bjoin algorithm requires network cost of $O(p^2)$ at the first MapReduce job because it is sending prefix elements to each prefix dimension. For the most of datasets, this is the most time consuming MapReduce job and bottleneck for the performance issue. So we changed this bottleneck part and reduced the network overhead to $\frac{O(p^2)}{2}$. Our algorithm will be presented in Chapter 5

Chapter 4

Pre-Processing

Vector pre-processing is also vitally important process for filtering based vector similarity join algorithms. Algorithms such as VCL and Bjoin applies pre-processing step to the dataset before it performs the join operation. The major intuition behind this pre-processing is to reduce the number of vector pairs for each dimension. The pre-processing orders the dimensions by its frequency. Which means we have less frequent dimensions in the front for the prefix filtering. This reduces the number of pairs greatly and only requires $O(n)$ for time complexity where n is the number vectors. In this chapter, we present a method called StdSort which even further improve the existing pre-processing method.

4.1 Pre-Processing Method of Previous Researches

Recent researches [4, 3], used frequency as the ordering method of the dimensions. We will call this as $sparsity$ method. In $sparsity$ method, dimensions are sorted with the frequency of dimensions. In other words, the less popular dimensions go to the front of the vectors. As the popular dimensions go to the back of vector, it will be unlikely to have those popular dimensions in the prefix of vector. Therefore prefix filtering will eventually have less candidate pairs and reduce the time needed for the algorithm. As dimensions are sorted by their frequencies, it only requires time complexity of $O(n)$ where n is the number of vectors. This pre-processing benefits overall computation time compared to the non-ordered dataset.

4.2 StdSort : Sorting with Standard Deviation

In this section, we will explain intuition for *StdSort* technique. It is a method of utilizing the standard deviation values of the dimensions. It is applied to the dataset only once and then future similarity joins can benefit from the already ordered dataset.

Standard deviation sorting method may be added on to the *sparsity* ordering explained in Section 4.1. We will first sort the dimensions by frequency (Number of vectors in that dimension) of these dimensions. Afterwards, if we have a tie for frequency, we will sort those by standard deviation value from dimensions. However this standard deviation sorting can only be applied along with the length filtering technique for the best performance. Because the prefix filtering technique does not consider the values of the vector, the variation of values does not mean anything to prefix filtering technique alone. *StdSort* will have same performance as *sparsity* in prefix filtering method alone because in *StdSort*, dimensions are sorted by frequency and then by standard deviation value if we have any tie. The main intuition behind the technique is that if a dimension has a high standard deviation value, it will be likely to have values that spread out more. It means that the dimensions will have high and low values. This is beneficial factor for both prefix filtering and length filtering, because in prefix filtering, it is better if we have less vector elements in the prefix. In achieving this, we need a dimension that contains high values to be placed at the front of the vector. For length filtering, it is better to have low value

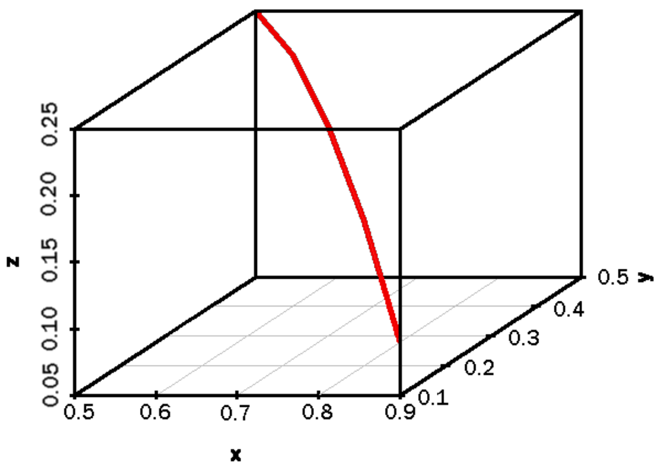
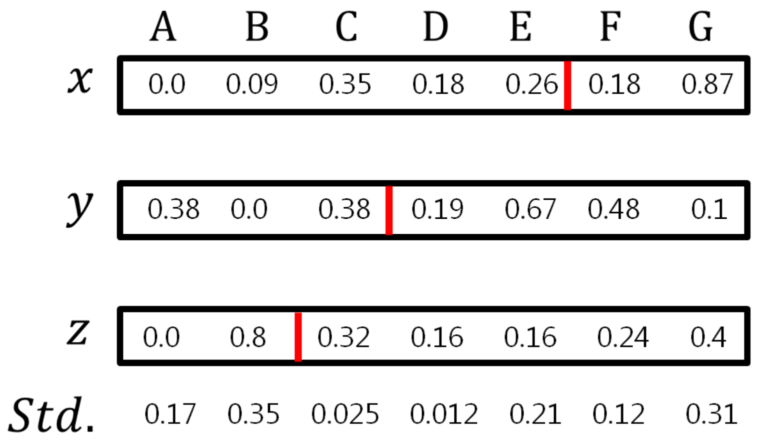


Figure 4.1: Variation of Dot Product Value

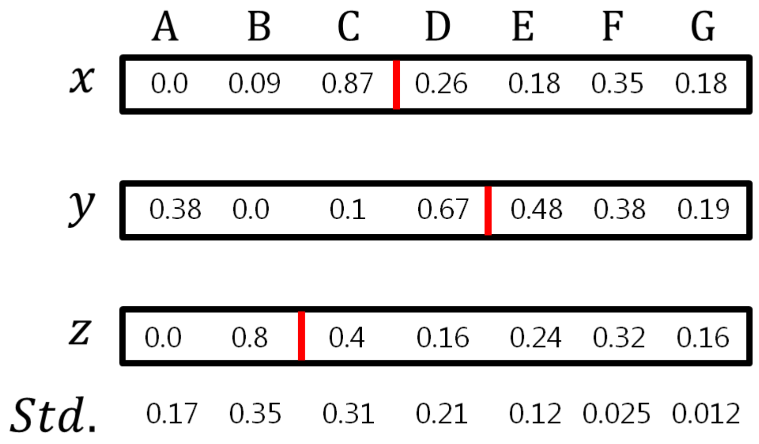
as possible because from Equation 2.5, if $D(x_p, y_p) + \|x_s\|\|y_s\| > t$, we need to confirm that candidate pair if it really has a similarity value greater than t. So for length filtering technique, it is better to have more weight towards $D(x_p, y_p)$ than $\|x_s\|\|y_s\|$, because that means we have more discriminative power.

Therefore, in order to satisfy these two contradicting standards from prefix and length filtering technique, we use standard deviation value for sorting method. It gives the dimensions more weight for having less vector elements inside the prefix. At the same time those values are likely to be different because the dimensions have high standard deviation values. Please refer to Figure 4.1. We plotted a graph of how different X and Y values can affect dot product value. In this graph we assumed 1 as the sum value for X and Y. X and Y plots denotes values for each two variables and Z plot denotes the dot product $(D(X, Y))$ of X and Y. Note that when the two variables differs more, we have lower Z value. For example, dot product of 0.9 and 0.1 is 0.09, whereas dot product of 0.5 and 0.5 is 0.25. So even the value of two variables add up to

1, if their values are more apart from one another, the resulting dot product value is less. This is the reason for using standard deviation for the ordering the dimensions. As more values in same dimension have different values, it would have high standard deviation value, resulting in less value for $D(x_p, y_p)$ from Equation 2.5.


A B C D E F G
x 0.0 0.09 0.35 0.18 0.26 0.18 0.87
y 0.38 0.0 0.38 0.19 0.67 0.48 0.1
z 0.0 0.8 0.32 0.16 0.16 0.24 0.4
Std. 0.17 0.35 0.025 0.012 0.21 0.12 0.31

(a) Sparsity Example


A B C D E F G
x 0.0 0.09 0.87 0.26 0.18 0.35 0.18
y 0.38 0.0 0.1 0.67 0.48 0.38 0.19
z 0.0 0.8 0.4 0.16 0.24 0.32 0.16
Std. 0.17 0.35 0.31 0.21 0.12 0.025 0.012

(b) StdSort Example

Figure 4.2: Vector Pre-Processing Examples

Please refer to the Figure 4.2. In this figure, we explain the difference between *Sparsity* and *StdSort*. Figure 4.2(a) is an example of *Sparsity* method and Figure 4.2(b) is an example of *StdSort* method. Each example contains three vectors. Letters A through G denote the dimensions and *Std.* denotes the standard deviation value of the each dimension. Red bold lines between dimensions denote the prefix dimension for each vector. Note that dimensions are sorted by the frequency first and then standard deviation value in Figure 4.2(b). Accordingly, dimension A is on the front of the vector because it is the only dimension that has one non-zero element. All other dimensions contains at least two non-zero elements. Since it is the sparsest dimension, it is brought to the front. Note from the dimension C, all the dimensions have three non-zero elements. Hence we will consider the standard deviation value for sorting from dimension C through G. As the standard deviation value increases, the values from different vectors vary more. Therefore the dot product will be likely to be less than dimensions with low standard deviation value.

For example, consider the example from this Figure 4.2 with the threshold 0.9. We will consider how many elements will be examined by prefix and length filtering for each pre-processing methods. For Figure 4.2(a), vector x has four non-zero elements to be examined until the prefix dimension. For vector y, there are two elements before the prefix and for vector z one element needs to be examined. Seven elements need to be considered for prefix filtering in total. For Figure 4.2(b), a sum of six elements needs to be considered. The difference arises because dimensions with high standard deviation tend to have elements with high values inside the dimension. Note that for vector y, prefix dimension increased from *sparsity* example. This is because for vector y, high standard deviation dimension brought low value 0.1 before the prefix dimension therefore 0.67 had to be examined for the proper length of prefix.

In length filtering, consider only the pair of x and y. Firstly, we will see how *Sparsity* example works with the length filtering. For any candidate pair to work with length filtering technique, we need to make same prefix point, since vector x's prefix dimension is larger than y's prefix dimension. Therefore, x's new prefix dimension will be between C and D. Accordingly, the suffix length of vector x will become 0.93 and length of vector y is 0.84. Now we can apply the length filtering principle from Equation 2.5. $D(x_p, y_p) = 0.13$ and $\|x_s\|\|y_s\| = 0.78$. So the sum of these two is 0.91, which is greater than threshold 0.9. Hence *sparsity* method will classify the vector pair x and y as a candidate pair and proceed for actual similarity calculation of the pair. However, the actual similarity value of these two vectors is 0.5, which is less than 0.9, therefore not similar. On the other hand, *StdSort* method does not judge this x and y pair as candidate pair. For *StdSort*, vector y's prefix dimension is larger than that of vector x. So we need to make prefix dimension of y to fit that of x. Then, $D(x_p, y_p) = 0.08$ and $\|x_s\|\|y_s\| = 0.45$. Sum of these two is only 0.53. Which is less than 0.9 and therefore this pair is not considered as a candidate pair in *StdSort* method. When we actually calculate the similarity value for each pairs, we need to examine $O(d)$ where d is the number of dimensions for the dataset. We see that in this example, *sparsity* will have fifteen elements to be examined whereas *StdSort* will have only six elements to be examined.

Chapter 5

PLF-Join

In this chapter, our MapReduce algorithm PLF-join will be presented. Compared to Bjoin[3], PLF-join efficiently reduces network cost from $O(p^2)$ to $\frac{O(p^2)}{2}$ where p is the number of prefix elements in the vector. Also PLF-join is composed of three MapReduce jobs compared to Bjoin's four MapReduce jobs. The first job of PLF-join is for filtering out dissimilar pairs with prefix filtering and length filtering. With the candidate vector pairs from the first job, the second job re-accesses the dataset vectors and emits vector elements by their pairs. The third job calculates the actual similarity value from the pairs and emits them if they are above the threshold.

5.1 Job 1 : Filter Dissimilar Pairs

This job is where we achieved major improvements compared to Bjoin. The difference comes from the line 6. Unlike Bjoin, PLF algorithm starts from the end of prefix and starts to append prefix. This is a big difference in terms of the network cost. Since Bjoin was originally sending prefix to every prefix dimensions, the cost was $O(p^2)$. However, in PLF algorithm, we are reducing this amount by aggregating the prefix and it becomes $\frac{O(p^2)}{2}$. We will explain about this with the detailed example at Figure Algorithm 1 is the mapper function of Job1. The algorithm submits dimension ids that are less than or equal to the prefix point p in Line 4 - 9. Notice that we start from the end of prefix and then start to aggregate prefix as we go to the front of the vector. This is because at the front part of the dimensions require later part of dimensions of prefix in order to fix the prefix point if the other vector matched has less prefix point than this vector (e.g. Figure 2.2).

Algorithm 1: MAPPER OF *Job 1. Filter Dissimilar Pairs*

```
Input: Key: Vector ID
       Value: Vector data.
       Format: $\langle v_{id} \rangle, \langle d_1 : val_1, \ldots, d_k : val_k \rangle$
Output: Key: Dimension id
        Value: Vector id, Prefix point, Prefix, Suffix length
        Form: $\langle d_k \rangle, \langle x_{.id}, p, x_p, \|x_s\| \rangle$
1 $p \leftarrow getPrefixPoint(x)$
2 $\|x_s\| \leftarrow getSuffixLength(x, p)$
3 $i = p$
4 while $i \geq 0$ do
5     $Key \leftarrow \langle d_k \rangle$
6     $x_p \leftarrow appendPrefix(x_p, val_i)$
7     $Value \leftarrow \langle x_{.id}, p, x_p, \|x_s\| \rangle$
8     $write(Key, Value)$
9     $i - -$
```

Algorithm 2 explains the Reducer of Job 1. The input of this reducer are dimension id and related values of vector information such as prefix elements, prefix dimension, suffix length of the vector, and vector id. Gathered vectors

Algorithm 2: REDUCER OF *Job 1. Filter Dissimilar Pairs*

```
Input: Key: Dimension id
    Value: A list of vector information
    Format: $\langle d_k \rangle, \langle x_{id}, p, x_p, \|x_s\| \rangle^*$
Output: Key: Candidate Pair
    Value: null
    Format: $\langle x_{id}, y_{id} \rangle, \langle - \rangle$
1 foreach element $x$ in the value list do
2     foreach element $y$ in the value list do
3         if $x_{id} < y_{id}$ then
4             $dotValue \leftarrow D(x_p, y_q)$
5             $mulValue \leftarrow getSuffixLenMul(x_p, y_p, p_x, p_y, \|x_s\|, \|y_s\|)$
6             if $dotValue + mulValue \geq t$ then
7                 $Key \leftarrow \langle x_{id}, y_{id} \rangle$
8                 $Value \leftarrow null$
9                 write(Key, Value)
```

inside these key dimensions are all candidates of the similar pair because they passed the prefix filtering predicate Equation 2.3. If this was the VCL algorithm, we wouldn't do anything else but join the gathered vectors, but we will apply length filtering algorithm. So for all pairs of vectors, we apply the predicate Equation 2.5. We will calculate the dot product of prefix part of the two vectors and then multiply the suffix lengths of them. However, as we explained earlier, there are cases like in Figure 2.2 where p_x and p_y doesn't match for any arbitrary vector x and y. So in this case, we will have to make the longer prefix to be same or shorter than the shorter prefix and include the left over length to suffix length (y_{ps} in Figure 2.2). In Algorithm 2, $getSuffixLenMul$ function at Line 5 handles this part.

5.2 Job 2 : Re-import the Vectors

A list of candidate vector id pairs which passed the prefix and length filtering is the input of the Algorithm 3. With the list of these candidates we need to actually calculate similarity value to if their similarity value really exceed the given threshold value. Since Hadoop MapReduce doesn't have the capability of sharing the data between the jobs, we need to import the data from HDFS(Hadoop File System) again. This is the reason why the second job is named as "Re-import the Vectors". So there are two types of inputs for the Algorithm 3. One is the candidate pairs from previous job and the other is re-imported vector data from HDFS. For calculating the similarity value, we need to match the candidate vector id pairs to the each vector data. Algorithm 3 does this job. At Line 1, it first distinguishes input data type from Candidate Pairs from Vector Data. Note that the Candidate Pairs does contain the duplication so at Line 2, it checks if the pair was already output by the algorithm before. This is done by checking the data structure that maintains list of output pairs. If the input data type is not Candidate Pairs, it goes to the Line 8. Here we know that the input is Vector Data. So we just use its id for the key and vector

Algorithm 3: MAPPER OF *Job 2. Re-import the Vectors*

```
Input: Candidate Pairs (With Duplicates)
    Format: $\langle x_{id}, y_{id} \rangle$
Output: Candidate Pairs
    Format: $\langle x_{id}, \langle x_{id}, y_{id}, - \rangle \rangle$ and $\langle y_{id}, \langle x_{id}, y_{id}, - \rangle \rangle$
Input: Vector Data
    Format: $\langle x_{id}, x_{Data} \rangle$
Output: Candidate Vector Data
    Format: $\langle x_{id}, \langle -, -, x_{Data} \rangle \rangle$

1  if Candidate Pair then
2      if Pair x and y are not already written then
3          $Key \leftarrow x_{id}$
4          $Value \leftarrow \langle x_{id}, y_{id}, - \rangle$
5          $write(Key, Value)$
6          $Key \leftarrow y_{id}$
7          $write(Key, Value)$
8  else
9      $Key \leftarrow x_{id}$
10     $Value \leftarrow \langle -, -, x_{Data} \rangle$
11     $write(Key, Value)$
```

data as their value. So after Algorithm 3, each candidate pairs and the related vector data can gather at same reducer.

Algorithm 4: REDUCER OF *Job 2. Re-import the Vectors*

```
Input: Candidate Pairs
        Format: $\langle x_{id}, \langle x_{id}, y_{id}, - \rangle \rangle$
Input: Candidate Vector Data
        Format: $\langle x_{id}, \langle -, -, x_{Data} \rangle \rangle$
Output: Half Vector Data
        Format: $\langle \langle x_{id}, y_{id} \rangle, \langle x_{Data}, - \rangle \rangle$

1  foreach element in the value list do
2      if value is type of $x_{Data}$ then
3          $\mathrm{x}_{Data} \leftarrow element.x_{Data}$
4  foreach element in the value list do
5      if $element.pair.firstID == x_{id}$ then
6          $Key \leftarrow element.pair$
7          $Value \leftarrow \langle x_{Data}, - \rangle$
8          $write(Key, Value)$
9      else if $element.pair.secondID == x_{id}$ then
10         $Key \leftarrow element.pair$
11         $Value \leftarrow \langle -, x_{Data} \rangle$
12         $write(Key, Value)$
```

Algorithm 4 handles partially aggregated data and generate pair key for the final similarity calculation. We have two input types one is the candidate pairs and the other is candidate vector data. What we need to do is send the vector data to the candidate pair as a key. So that for all candidate pairs, related vector data can gather and we can perform the similarity calculation. From Line 1 - 3, the algorithm iterates through the list of values and retrieves vector data first. After that for each Candidate Pairs, the algorithm sends Candidate Pair as a key and vector data as a value. This is done in Line 4 - 12.

5.3 Job 3 : Calculate Similarity

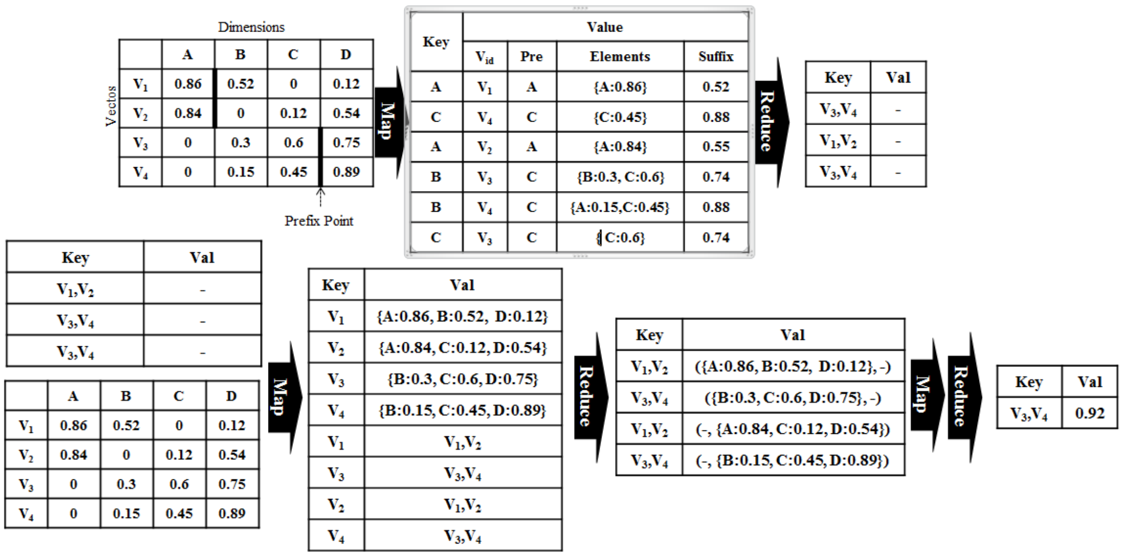
In this job, actual similarity values of the vector pairs are calculated. Everything we need for the calculation is already gathered from Algorithm 4. So in this job we do not need any mapper. So for Job 5.3, we only have a reducer. Algorithm 5 is the responsible reducer for similarity calculation. It simply calculates the similarity value at Line 3 and if the value is greater or equal to t, it outputs the pair as the similarity pair.

Algorithm 5: REDUCER OF *Job 3. Calculate Similarity*

```
Input: Half Vector Data
        Format: $\langle\langle x_{id}, y_{id}\rangle, \langle x_{Data}, -\rangle\rangle$
Output: Similarity Value
        Format: $\langle\langle x_{id}, y_{id}\rangle, \langle Sim(x,y)\rangle\rangle$
1 $x_{Data} \leftarrow value.first$
2 $y_{Data} \leftarrow value.second$
3 $Similarity\ Value \leftarrow Cosine(x_{Data}, y_{Data})$
4 if $Similarity\ Value \geq t$ then
5     $Key \leftarrow \langle x_{id}, y_{id}\rangle$
6     $Value \leftarrow Similarity\ Value$
7     $write(Key, Value)$
```

5.4 Example of PLF-join

In Figure 5.1, we have four vectors with four dimensions in input dataset. In this example the threshold is set to 0.9 and all the vectors are normalized. In the table of ⟨*key*, *value*⟩, V_{id}, P, *Prefix*, and *SuffLen* represents the ID of



	A	B	C	D
V_1	0.86	0.52	0	0.12
V_2	0.84	0	0.12	0.54
V_3	0	0.3	0.6	0.75
V_4	0	0.15	0.45	0.89

Key	Value			
	V_{id}	Pre	Elements	Suffix
A	V_1	A	{A:0.86}	0.52
C	V_4	C	{C:0.45}	0.88
A	V_2	A	{A:0.84}	0.55
B	V_3	C	{B:0.3, C:0.6}	0.74
B	V_4	C	{A:0.15,C:0.45}	0.88
C	V_3	C	{C:0.6}	0.74

Key	Val
V_3,V_4	-
V_1,V_2	-
V_3,V_4	-

Key	Val
V_1,V_2	-
V_3,V_4	-
V_3,V_4	-

	A	B	C	D
V_1	0.86	0.52	0	0.12
V_2	0.84	0	0.12	0.54
V_3	0	0.3	0.6	0.75
V_4	0	0.15	0.45	0.89

Key	Val
V_1	{A:0.86, B:0.52, D:0.12}
V_2	{A:0.84, C:0.12, D:0.54}
V_3	{B:0.3, C:0.6, D:0.75}
V_4	{B:0.15, C:0.45, D:0.89}
V_1	V_1,V_2
V_3	V_3,V_4
V_2	V_1,V_2
V_4	V_3,V_4

Key	Val
V_1,V_2	({A:0.86, B:0.52, D:0.12}, -)
V_3,V_4	({B:0.3, C:0.6, D:0.75}, -)
V_1,V_2	(-, {A:0.84, C:0.12, D:0.54})
V_3,V_4	(-, {B:0.15, C:0.45, D:0.89})

Key	Val
V_3,V_4	0.92

Figure 5.1: Example of PLF-join

a vector, prefix point, prefix, and suffix length of the vector, respectively. The suffix length is used when *PLF-join* performs length filtering. For simplicity, we omit the shuffling of MapReduce in the figure.

The mapper of Job 1 reads the input dataset of vectors. For each vector, it searches for a prefix point p, by finding a dimension where $\|x_s\| < t$, i.e., the suffix length is less than t. For example, the prefix point of vector V_3 is C since $\sqrt{0.75^2} = 0.75 < 0.9$. Note that $\sqrt{0.75^2 + 0.6^2} = 0.96 > 0.9$, so dimension B cannot be the prefix point. Once we determined the prefix point and suffix length of the vector, the algorithm will iterate through the elements before the prefix point and output each element's dimension as key and V_{id}, p, *Prefix*, and *SuffLen* as value. The mapper outputs $\langle key, value\rangle$ pairs. One of them is $\langle$B,$\langle V_3$,C,{B:0.3, C:0.6},0.74$\rangle\rangle$.

The reducer then inputs$\langle key, value\rangle$ pairs, and makes candidate pairs of vectors, during which we perform the length filtering technique. By Equation 2.5 in the section 2.2.2, we calculate $D(x_p, y_p) + \|x_s\|\|y_s\|$. If the value is greater than or equal to the threshold, we hold the pair. Otherwise, we can safely discard the pair. For example, consider a pair V_3 and V_4 which share the same key value. Then the value is $0.315{+}0.6675 = 0.9825 > t = 0.9$. So this pair $\langle V_3, V_4\rangle$ remains candidate. On the other hand, consider another pair V_3, V_5. $D(x_p, y_p)+\|x_s\|\|y_s\|$ for V_3 and V_5 is 0.3. Because $D(x_p, y_p) = 0.3$ and $\|x_s\|\|y_s\| = 0$ ($\|y_s\| = 0$). Since $D(x_p, y_p) + \|x_s\|\|y_s\|$ ¡ 0.9, the pair V_3, V_5 is discarded. If we were using VCL algorithm, this pair would have survived to the last job because it passed the prefix filtering. However, PLF-join efficietly filters this dissimilar pair out by length filtering technique. The reducer uses *key* to pass the pair candidacy to the next job.

Job 2 has two inputs. One is the candidate pairs output from Job 1, and the other is original input vectors. The candidate pairs are split into vector ids, and output as *keys*. In the figure, $\langle V_3, V_4\rangle$ is split to V_3 and V_4. The reducers join candidate pairs and vector elements by common vector id, combine vector ids, and output it as a key and the related data. For example, $\langle V_3$, $\langle\{$B:0.3,C:0.6,D:0.75$\}\rangle\rangle$ and $\langle V_4$, $\langle\{$B:0.15,C:0.45,D:0.89$\}\rangle\rangle$, and $\langle V_3$, $\langle V_3, V_4\rangle\rangle$, $\langle V_4$, $\langle V_3, V_4\rangle\rangle$ are joined to output the records $\langle\langle V_3, V_4\rangle$, $\langle\{$B:0.3,C:0.6,D:0.75$\}, -\rangle\rangle$ and $\langle\langle V_3, V_4\rangle$, $\langle -, \{$B:0.15,C:0.45,D:0.89$\}\rangle\rangle$.

At Job 3, the mappers do nothing but shuffle. This is called Identity-Mapper and this is why we didn't put anything between Map and Reduce in the Figure 5.1. The reducers calculate the similarity of each candidate pairs. If the similarity exceeds the given threshold, it outputs the pair with its similarity value. In the figure, $\langle\langle V_3, V_4\rangle, 0.92\rangle$ is the final output of *PLF-join*.

Chapter 6

Experiment

We show performance differences between PLF-join algorithms and other baseline algorithms. We experimented with multiple dataset with multiple threshold values on our algorithm and other algorithms.

6.1 Experiment Setup

We used two datasets in this experiment.

- `UKBench-` UKBench dataset is a dataset that contains visual information as a word of bag. Each vector represents an image and each dimension represents a visual word. The features (dimensions) are extracted by Scale-Invariant Feature Transform (SIFT) algorithm [10]. One of the applications of this dataset using similarity join algorithm can be finding similar images. The dataset is expanded to 7 times. Total size of this dataset is 667 MB.
- `MovieLens-` This dataset is a movie review dataset from a number of users. Each vector represents a user and each dimension represents a movie. A user rates a movie on 5 discrete numbers from 1 to 5. It is collected and processed by GroupLens, a research group at the University of Minnesota

[7]. In recommendation systems, finding similar users or items (movies) is a very frequently used operation. The size of dataset is 87.6 MB.

- `LiveJournal-` LiveJournal is an online social network service that operates with almost 10 million users. Users can create journals and group blogs. It also enables users to have friendship with each other. A user is represented as a vector and dimensions are other users. So the values inside the vector represents friendship relationships between users. Similar users are applied in recommending friends or detecting communities from the network. The dataset is from *SNAP: Network datasets* [6].

We used Hadoop 1.1.2 version and 12 nodes for the experiment. Each node is configured with i7-3820 3.60 GHz CPU, 4GB DDR3 RAM, and 2TB WD HDD. Since each CPU has 4 cores, we ran 4 reducers for each node.

6.2 Time Performance

We measured the time performance of the algorithms on different threshold values. Figure 6.1 shows the result of the experiment. We tested the four different thresholds which are 0.3, 0.5, 0.7, and 0.9. Each sub-figure in Figure 6.1 represents an experiment with different dataset. Note that the *V-SMART* has same performance over all threshold because the algorithm is not able to respond to the differing threshold. Except that case, all other algorithms show generally reduced time when we have higher threshold. This is because higher threshold results in less or equal number of similar vector pairs to the lower threshold. As we have more similar pairs to process, they will pass the filtering process and takes more time. Thus it is important to make the algorithm react to different threshold elastically. For UKBench and MovieLens, the performance improvement of PLF-join algorithm is a lot. However, in LiveJournal experiment at Figure 6.1(c), All three algorithms except V-SMART achieves similar performance. The reason behind this is that compared to other two datasets, LiveJournal dataset contains more similar pairs. Since both PLF and Bjoin have more MapReduce jobs, more similar pairs will result in more intermediate data between the jobs. This results in the bottleneck of the algorithm. Still PLF-join achieves reasonable performance with LiveJounrnal dataset, but if a dataset contains a lot of similar pairs, VCL or other algorithms could perform better than PLF-join.

6.3 StdSort Experiment

In this section we will describe the experiment with *StdSort* that was introduced in Chapter 4. We used the Gowalla dataset from SNAP[6]. It is a social network dataset that contains the user information about check-ins to local locations. We transformed the dataset into vector representations. Each dimension represents the location and each vector represents the user. Value inside the vector represents the frequency of visits from each user to a particular location. For the experiment we used similarity threshold 0.9 and the system was same as our other experiment.

In this experiment we have three methods. *No sort* which does not employ any pre-processing sort methods. Here we use the ordering of the vectors as given from dataset. *Sparsity* method uses only the sparsity of the dimensions. Lastly, *StdSort* utilizes both sparsity and standard deviation for sorting.

Unlike *No sort* method, we need to take into account the time need for sorting for *Sparsity* and *StdSort*. The time complexity of sorting the dimensions is just $O(n)$. In this experiment it took only 9 seconds in a single machine to sort the dimensions for this dataset.

First experiment is the Time experiment for measuring the time taken for the whole similarity join operation. As expected, *StdSort* showed best performance among all three methods. Second experiment is the Filtering power effectiveness experiment. Main point of this experiment is to check if sorting methods does affect filtering power of prefix and length filtering algorithms.

If we have fewer candidates, we will have less pairs to calculate for similarity value.

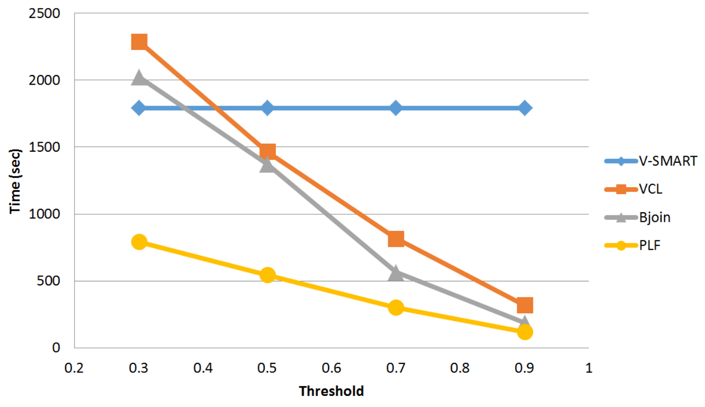
Refer to the Figure 6.2(a). As expected the *No sort* method takes the most time. It is as expected result because the dataset is not ordered for prefix filtering. *Sparsity* method is slightly better method than *No sort*. This technique does contribute for the prefix filtering technique so it gives little more advantage compared to *No sort*. *StdSort* is the best method because it not only utilizes sparsity for the prefix filtering, but also standard deviation value for the length filtering technique. 6.2(b) shows the experimental result for filtering experiment of each method. Note that it is in the log scale. Filtering experiment also shows that *StdSort* efficiently reduces number of candidate pairs for vector similarity join.

6.4 Combining PLF-join and StdSort

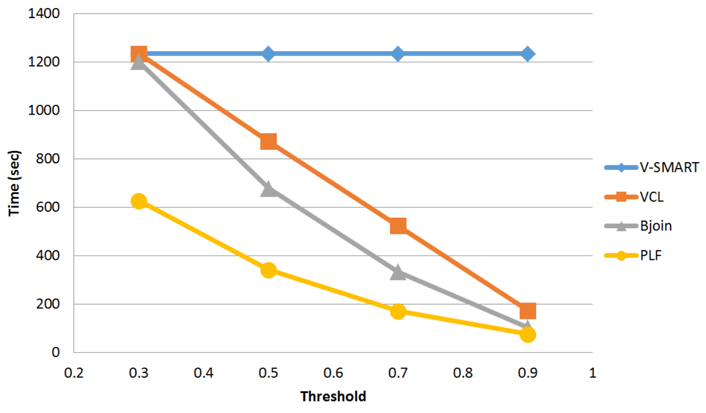
In this section, we present a combined experimental results of PLF-join and StdSort and discuss meaning of the results.

Refer to Figure 6.3. The graph shows the time taken for joining MovieLens dataset on different threshold values. Bjoin+Std denotes Bjoin algorithm with StdSort applied for the pre-processing. Similarly, PLF+Std denotes PLF-join algorithm with StdSort applied for the pre-processing. For other algorithms, Sparsity pre-processing is applied.

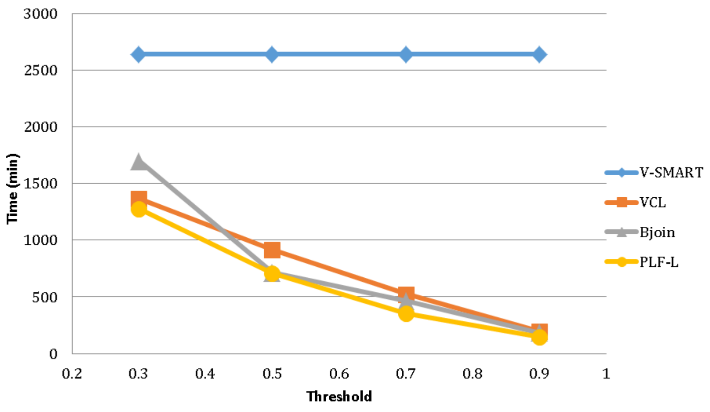
Note that StdSort sometimes improves the performance and sometimes not. For example, compare Bjoin and Bjoin+StdSort at threshold 0.5. In this case StdSort provides positive effect and speeds up the process. However, PLF+StdSort at threshold 0.3 slows down the process compared to PLF alone. To understand the reason behind this, we need to know about the dataset. MovieLens is a dataset that contains information of users' review on movies. So each vector represents a user and each dimension represents a movie. Movie rating is one of the values from 1, 2, 3, 4, 5. This value distribution is the main reason StdSort doesn't work well with the dataset. Since all values are distributed to only 5 discrete numbers, standard deviation values of the dimensions are not so different from each other. Hence StdSort doesn't have great impact on this dataset. If we have a dataset with great range, we think StdSort can contribute for the faster vector similarity join.



(a) UKBench

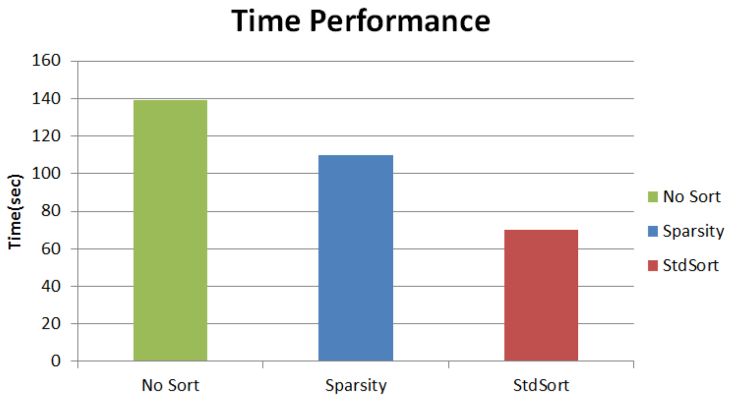


(b) MovieLens

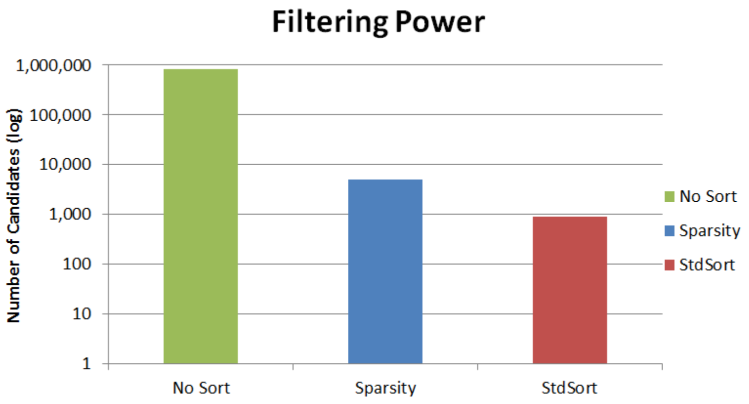


(c) LiveJournal

Figure 6.1: Time Performance Experiment



(a) Time Experiment



(b) Filtering Experiment

Figure 6.2: StdSort Experiment

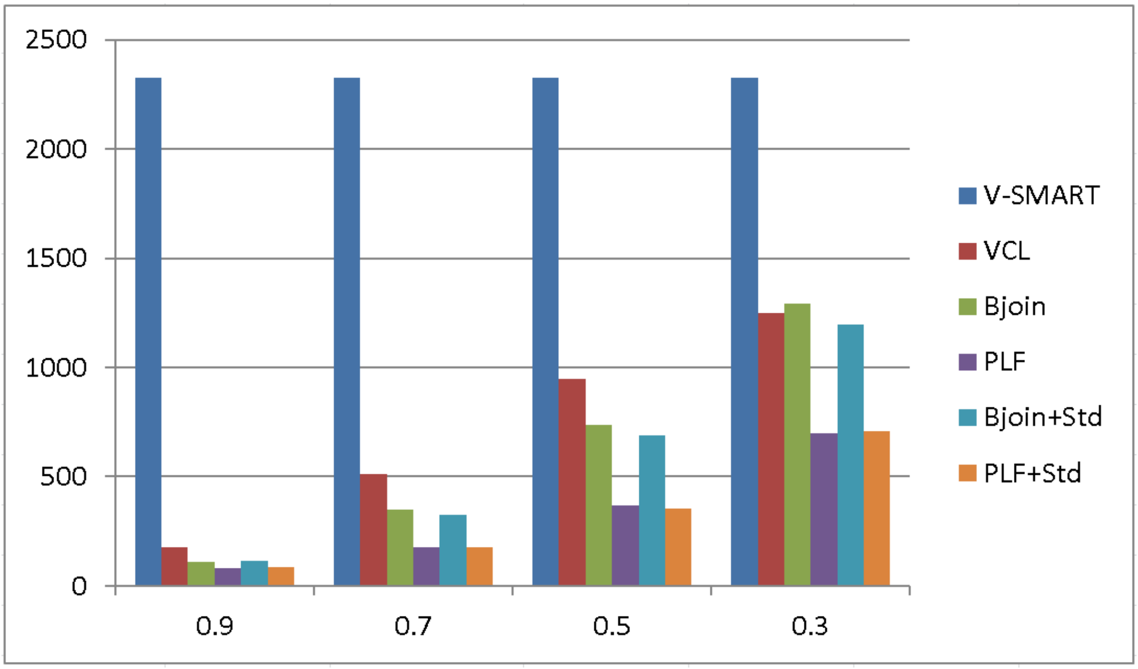


Figure 6.3: Time Experiment of PLF-join and StdSort

Chapter 7

Conclusion

Vector Similarity Join is a fundamental operation that is widely used from collaborative filtering to graph analytics. However, its $O(n^2)$ computational load made it impractical to use in real world problems. Even a lot of researches have been conducted after the MapReduce framework was introduced, it is still difficult to achieve a practical time complexity. Previous researches tried to achieve better performance through adapting the MapReduce distributed framework. But there are still many inefficiencies remained. For example, *V-SMART* wasn't able to elastically respond to the different threshold, which made it impractical to some applications that require high thresholds. *VCL* algorithm uses prefix filtering, but it wasn't able to efficiently filter out dissimilar pairs that shared same prefix dimensions. *Bjoin* utilizes both prefix and length filtering techniques, but it suffers from large prefix duplication network overhead and more numbers of MapReduce jobs. So we proposed MapReduce algorithm PLF-join which reduced the network overhead of the previous algorithm. The experiments show that our algorithm achieved about 2 times faster than the existing algorithm. Also we proposed an efficient pre-processing algorithm which utilizes standard deviation values of the vector. The experiments show that it is

57% faster than the existing pre-processing algorithm. Although we achieved much performance gain, as the Big Data community is moving toward to more in-memory processing, these vector similarity join algorithms should also be able to process large data using in-memory system to be more practical for real world problems.

Bibliography

[1] D. Lee, J. Park, J. Shim, and S. goo Lee. “An Efficient Similarity Join Algorithm with Cosine Similarity Predicate,” In *DEXA (2)*, pages 422-436, 2010.

[2] D. Lee, “An Efficient Filtering Framework for Vector Similarity Joins,” PhD thesis, Seoul National University, Seoul, South Korea, Aug, 2011.

[3] B. Yang, ”A MapReduce-based Filtering Framework for Vector Similarity Joins”, Master’s thesis, Seoul National University, Seoul, South Korea, Feb, 2013.

[4] R. Vernica, M. J. Carey, Michael J. and C. Li, ”Efficient parallel set-similarity joins using MapReduce”, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495-506 New York, NY, USA, 2010. ACM.

[5] A. Metwally, and C. Faloutsos, ”V-SMART-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors” , Proc. VLDB Endow., 5(8):704-715, Apr. 2012.

[6] J. L. E. Cho, S. A. Myers. Friendship and mobility: Friendship and mobility: User movement in location-based social networks. International Conference on Knowledge Discovery and Data Mining, 2011.

[7] MovieLens data sets, grouplens ressearch. http://www.grouplens.org/node/73 Aug. 2011.

[8] B. Yang, J. Myung, S. goo Lee, and D. Lee. A mapreduce-based

ltering algorithm for vector similarity join. In *ICUIMC(IMCOM) '13, January 2013.*

[9] M.-M. Deza and E. Deza. Dictionary of Distances. Elsevier Science, Amsterdam, The Netherlands, Oct. 2006.

[10] Lowe, G. David, istinctive Image Features from Scale-Invariant Keypoints. Int. J. Comput. Vision, Hingham, MA, USA, Nov. 2004.

요약

PLF-Join: 벡터 유사 조인을 위한 효율적인 맵리듀스 알고리즘

김현준
전기컴퓨터공학부
서울대학교 대학원

벡터 유사 조인은 주어진 벡터들 중에서 모든쌍의 벡터 유사치가 특정 한계치를 초과하는 벡터들을 찾는 문제이다. 벡터 유사 조인은 중복 제거나, 추천, 소셜 데이터 마이닝 등 많은 곳에서 사용된다. 그렇지만 n이 벡터의 개수일때 기본적으로 $O(n^2)$의 시간 복잡도가 필요하다. 이런 비현실적인 시간 복잡도가 벡터 유사 조인이 실제적인 문제들에 활용되는데 어려움을 주고 있다. 그렇기 때문에, 많은 하둡 맵리듀스 기반의 알고리즘들이 제시되었다. 현재 가장 빠른 알고리즘은 Prefix 필터링과 Length 필터링을 고려한 방법으로 벡터 유사 조인 시간을 줄이고 있다. 이보다 더 소요되는 시간을 줄이기 위해서, 이 논문에서는 네트워크 입출력을 줄인 변형 알고리즘을 제시한다. 이외에도 벡터 유사 조인을 촉진 시키는 효율적인 전처리 방법 또한 제시한다.

주요어: 벡터 유사 조인, Prefix 필터링, Length 필터링, 모든쌍 유사 검색, 표준편차

학번: 2012-23207