2015 International Conference on Computational Science and Computational Intelligence

Optimization of All Pairs Similarity Search

Yuechen Chen
School of Software Engineering
Shanghai Jiao Tong University
Shanghai, China
chenyuechen123@sjtu.edu.cn

Xinhuai Tang
School of Software Engineering
Shanghai Jiao Tong University
Shanghai, China
tang-xh@cs.sjtu.edu.cn

Bing Liu
Shijiangzhuang Institute
Engineering College
Shijiazhuang, China
liubbb@163.com

Delai Chen
Shanghai Telecom
Shanghai, China
dlchen@189.cn

***Abstract*—All pairs similarity search (APSS) is the problem of finding all the similar pairs of items, whose similarity is above a given threshold. APSS algorithm is applied to many data mining fields, such as document matching, collaborative filtering. Due to a large scale of data in real life, some recent work used partitioning, inverted indexing, parallel accumu- lation, and hashing approximation to optimize the APSS algorithm. To optimize the APSS problem, this paper analyzes and compares two parallel approaches. To demonstrate the performance gain of our optimization approaches, we implement our algorithms on Spark and conduct the evaluation on a dataset of one million movies, which gains better performance speedup than other works.**

***Keywords*-Distributed computing, similarity search, parallel optimization, Spark, data mining**

I. INTRODUCTION

All pairs similarity search (APSS) is a problem of finding all the similar pairs of items within a dataset, whose similarity is above a user-defined threshold. APSS is used in many data mining applications. Common examples are as follows.

Document matching [13] is a process of matching similar documents based on the similarity among different documents, which is usually used in duplicated document detection described in [16].

Collaborative Filtering [22] is a process of filtering information or patterns by methods involving collaborative among different user views, which makes recommendations according to similar user taste. As a result, it is strongly dependent on the all pair's similarities among users.

Besides above examples, there are many other data mining applications involving APSS such as search query suggestions, spam detections, clustering [12], and social network [19]. It is obvious that APSS is a general data mining problem. In recent years, big Internet enterprises have always faced problems involving a large scale of data. The naive solution of APSS problem is to join item matrix with itself, and leads to $O(I^2F)$ time consumption, where I represents the item size and F represents the global feature dimensionality. According to the APSS problem, researchers have pursued in two directions to solve the problem. Some previous work tried to optimize the APSS algorithm through distributed computing frameworks, such as MapReduce model, while some recent work used partitioning, inverted indexing, parallel accumulation, and hashing approximation to optimize the APSS algorithm. However, each optimization has its own speedup. As a result, considering the large-scale dataset, we should figure out a distributed solution combining different optimization techniques.

In our work, we combine different optimization techniques and introduces two optimization approaches. One partitions item matrix, and computes each sparse matrix by inverted index, and the other uses an approximate algorithm to compute similarity pair candidates, and then filters the false positive candidates by exact pair similarity computation. To further evaluate the performance and accuracy of our solutions, we implement our optimization algorithms on Spark, and use one real-world million scale of movies in real life as dataset for evaluations.

The structure of this paper is organized as follows. We describe the background about APSS in Section II. We present our algorithms in Section III. We list some evaluations and analyze the result in Section IV, and finally conclude the paper in Section V.

II. BACKGROUND

APSS is a general problem in data mining applications. According to the work in [2], the APSS problem is defined as following: Given a set of item vectors $V = \{v_1, v_2, ..., v_n\}$ of fixed dimensionality of f, and a similarity function about $\text{sim}(x, y)$ in which x, y are item vectors included in set V, and a similarity threshold t, we wish to compute the pair set of all the similarity pairs (x, y), which satisfies $\text{sim}(x, y) \geq t$.

There are a wide range of work on APSS problem, and the naive solution to this problem is to make similarity joins. But this solution performs time-consuming for large scale of data. So there are many work related to the optimization about this problem. One optimization approach is to use inverted index, which is widely used in information retrieval [7], [13], [17]. R. J. Bayardo [2] proposed a simple algorithm based on novel indexing and optimization methods with exact methods, and J. Lin [6] compared the brute force solution with inverted-based solution, showing that inverted-based solution eliminates many unnecessary computations. Many recent work considered approximation techniques [14]. One typical approximation technique is the locality-sensitive hashing algorithm [4], [15], which maps high-dimension-feature items to low-dimension fingers by a few number of hash function projections. Approximation should pay costs for accuracy reduction, and F. Ture [8] compared a brute-force approach with a locality-sensitive hashing approach, and showed that

978-1-4673-9795-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CSCI.2015.16

the approximation approach reduce accuracy of similarity pairs much. As a result, to further improve the performance of approximation approach, V. Satuluri [9] proposed a principled Bayesian algorithm for the subsequent phase of similarity search through LSH, involving candidate pruning by Bayesian algorithm, but this approach further reduces the accuracy by Bayesian approximation pruning.

With increasing scale of dataset, one single machine cannot solve the APSS problem itself, so we need to resort to distributed solutions. Many recent work [3], [10] used the Hadoop MapReduce framework to solve the problem, but the new data processing framework Spark has a better performance than Hadoop MapReduce. One problem which will come with distributed framework, is about how to partition the dataset, in order to reduce the redundant computations [21]. M. Alabdul-jalil [1] proposed a scalable approach called partition-based similarity search, which used a static partitioning algorithm considering the load balance [20] to place dissimilar vectors into different groups, and execute comparison tasks in parallel in each partition. Then according to web search problem, E. Kayaaslan [5] proposed a novel inverted index partitioning model on hypergraph partitioning. Furthermore, in order to fast match similarity pairs, A. Awekar proposed a three-phase framework [11] of data preprocessing, pairs matching, and indexing, effectively reducing the size of the search space through some filtering conditions and heuristic optimizations. To improve the performance of fast matching, A. Awekar then proposed an incremental all pair's similarity search [18] for fast matching problem which developed efficient I/O techniques to manage computation history, and efficiently identified and pruned redundant computations.

In our work, we introduced two scalable APSS approaches by combining common optimizations in recent work, and then implement and optimize them on Spark.

III. ALGORITHMS

In recent work, there are many optimization approaches on APSS problem. Our work mainly involves two APSS optimization approaches. One is a two-stage partition-based approach with inverted index, and the other is a filter-based locality-sensitive hashing approach. The former is an exact approach, which uses partitioning to avoid unnecessary network and I/O costs, and inverted index to eliminate unnecessary computations, and the other is an approximation approach which takes less time to compute the results with much less resources consumption at the cost of accuracy. The two approaches are discussed in the subsection A & B.

A. Partition-based Approach with Inverted Index

When the scale of input dataset is quite large, one single machine cannot solve such a big problem, thus we need to resort to distributed solutions. When it comes to distributed solutions on APSS problem, it is an intuitive thought to partition the dataset into different blocks. The naive APSS solution is to join item matrix with itself, and each partition involves the pairs between one vector and another vector. As a result, in this solution, it needs to transmit all the item vectors into all the partitions, which would make large network and I/O overhead. In our work, we divide the item similarity matrix into blocks through averagely partitioning by row and column, where V is the sparse vectors described in Section II, and b is defined as the partition number by rows or columns. Our average partition algorithm is presented as pseudo-code in Fig. 1. Through this average partitioning strategy, the transmission costs of one vector reduce from b^2 to $2b-1$.

```
1:  procedure AVERAGEPARTITION(V, b)
2:      P_0, P_1, ..., P_{b^2-1} ← ∅
3:      for i = 0 to b − 1 do
4:          for j = 0 to b − 1 do
5:              if i = j then
6:                  for k = 0 to |V|/b − 1 do
7:                      P_{ib+j} = P_{ib+j} ⋃ v_{|V|/b i+k}
8:                  end for
9:              else
10:                 for k = 0 to |V|/b − 1 do
11:                     P_{ib+j} = P_{ib+j} ⋃ v_{|V|/b i+k} ⋃ v_{|V|/b j+k}
12:                 end for
13:             end if
14:         end for
15:     end for
16:     return P_0, P_1, ..., P_{b^2-1}
17: end procedure
```

Fig. 1. Average Partition Strategy

Besides the partition strategy of item vectors, there are still several inefficiencies in this approach: 1) each similarity pair is pairwise so that $\text{sim}(x, y)$ is equals $\text{sim}(y, x)$, and it wastes computation effort to both compute pairwise pairs. 2) considering that the data scale is still quite large in each partitioned block, a naive brute force APSS solution still pull down the performance of our partition-based approach.

For the former issue, we split the blocks by the diagonal line, and the computation scale would be halved. For the latter issue, a better approach is to build an inverted list index of the input vectors.

As a result, we use an inverted-index-based similarity accumulation approach in the second stage based on [2]. As shown in Fig. 2, V is the input set of sparse vectors and t is the similarity threshold described in Section II. The feature lists are represented as $F = \{F_1, F_2, ..., F_n\}$, and the inverted-index lists are represented as $I = \{I_1, I_2, ..., I_n\}$. I_i consists of all the non-zero feature-weight pairs (c, w), where $c[i] = w$, $w \neq 0$. In the inverted-index-based similarity accumulation approach, we scan all the inverted lists to do the similarity accumulation, and it returns all the similarity pairs whose similarity is above threshold t, while we collect all the returned similarity pairs into union set S, and build the inverted-index list dynamically.

Considering that in the dataset there are I items, and totally F different features, and each item has averagely A features.

procedure FINDALLPAIRS(V, t)
  $S \leftarrow \varnothing$
  $I_1, I_2, ..., I_{|F|} \leftarrow \varnothing$
  for each $v_i \in V$ **do**
    $S \leftarrow S \bigcup FindSimilarityPair(v_i, I, t)$
    for each j such that $v_i[j] > 0$ **do**
      $I_i = I_i \bigcup (v_i, v_i[j])$
    end for
  end for
  return S
end procedure
procedure FINDSIMILARITYPAIRS(v, I, t)
  $M \leftarrow Empty\ Map, P \leftarrow \varnothing$
  for each i such that $v[i] > 0$ **do**
    for each $(x, x[i]) \in I_i$ **do**
      $M[x] = M[x] + v[i] * x[i]$
    end for
  end for
  for each $(x, w) \in M$ **do**
    if $w \geq t$ **then**
      $P = P \bigcup (v, x, w)$
    end if
  end for
  return P
end procedure

Fig. 2. Accumulated Inverted-index Approach

The computation cost is shown in (1), in comparison to I^2F computation consumption. Given a certain I and F, the function $Cost(A, I, F)$ is a monotonically increasing function, so that the computation cost increases with the average feature amount.

$$Cost(A, I, F) = (1 - (1 - \frac{A}{F})^A) I^2 F \tag{1}$$

Through the inverted index, we can find all the non-zero matching pairs as if it were an O(1)-time query, and view related items as candidate similar items, which would avoid many unnecessary non-zero matching comparisons.

Our two-stage APSS approach offers locality improvement, avoids unnecessary network and I/O overhead by partition strategy in the first stage, and eliminates non-zero matching computations by inverted index in the second stage.

B. Filter-based Locality-sensitive Hashing Approach

Equation (1) indicates that the computation cost of the inverted-index algorithm is $O(CI^2F)$, where $C = 1 - (1 - \frac{A}{F})^A$. Given a certain A, C can be considered a constant. As a result, with the increasing scale of items and features, the fully exact APSS solution would be still bound to $O(I^2F)$. To meet with this problem, there are intuitively two solutions as following: 1) Increase the compute resources for more partitioning blocks. 2) Without the increasing of computing resources, we should focus on the algorithm itself. Due to the fact that exact APSS should compute all the non-zero similarity pairs, the scale of item is always taken into consideration. Then one intuitive solution is to reduce the large scale of feature. There is a common algorithm named locality-sensitive hashing (LSH) based on the latter solution. It is an approximation algorithm, in which the large scale of feature for each item is reduced to user-defined number k, satisfying $k \ll F$.

procedure FILTERBASEDLSH(b, r, t)
  Generate random numbers from a Gaussian distribution by using Box-Muller transformation [25], in which the average is 0 and STD is 1, and finally generate k numbers of random hashing vectors.
  Compute the Cartesian product of item vector and random hashing vectors, and generate a 0-1 fingerprint for each item by sign of the Cartesian product.
  Divide the fingerprint into b bands, and there are r finger in each band, s.t. $k = b * r$.
  Partition each item into different buckets by fingers in each band.
  Get all item combinations of each bucket, like pair(x, y), which means pair(x, y) is a candidate pair, and then merge it together.
  Filter all the false positive pairs whose similarity is under t, and return all the qualified pairs
end procedure

Fig. 3. Filter-based LSH approach

A common solution for LSH [24] is to map one item vector into a k-length fingerprint by k hash functions. The core principle of LSH is that the fingerprint of similar items would be more likely to be similar. We divide items into buckets by fingerprint, and then we consider those pairs hashed to the same bucket as a candidate group. This hashing approach lead to poor accuracy. A more accuracy way is to use the band strategy, in which we divide the fingerprint matrix into b bands consisting of r rows each. For each band, there is a hash function that takes vectors of r integers and hashes them to different buckets by fingerprint. Through the band strategy, LSH approach gains better precision and recall rate [9].

However, the precision and recall rate for LSH approach is still not very good. As a result, an intuitive approach is to check the candidate pairs by exact APSS approach. In our work, we propose a two-stage approximation approach. In the first stage, we use a locality-sensitive hashing approach to map each item into different fingerprint, and then partition each item into different buckets by its fingerprint. In the second stage, we use an exact APSS solution to filter all the false positive pairs. The complete solution of our two-stage approximation approach on cosine similarity metric appeared as pseudo-code in Fig. 3.

In this two-stage approach, we compute the all item combinations of each bucket, and then filter the false positive pairs. Equation (2) shows the computation cost of this approach, where $|B_{ij}|$ represents the size of the jth bucket in the ith band.

$$Cost(b, r, I, F) = \sum_{i=1}^{b} \sum_{j=1}^{2^r} \frac{|B_{ij}|^2 F}{2} \quad (2)$$

Supposing that items in each band satisfies a Gaussian distribution with mean $\frac{I}{2^r}$ and variance σ, Equation (2) can be transformed to (3).

$$Cost(b, r, I, F, \sigma) = \frac{bF}{2}(\frac{I^2}{2^r} + 2^r\sigma^2) \quad (3)$$

When there is a large scale of data, considering the fact that r is usually set less than 16 and $\sigma \ll I$, the computation cost of two-stage approximation approach is considered $O(\frac{b}{2^r}I^2F)$, and greatly improve the performance in comparison to the exact approach.

IV. EVALUATIONS

In this section, we design and conduct to evaluate different optimization approaches used in our work, and compare our APSS optimization approaches with the previous APSS algorithms. In our experiments, we implement our optimization approaches as Spark applications in Scala, with one real-world dataset of one million scale. All the experiments in the subsections were performed on a Yarn cluster in which there are 736 maximum cores and 2.57 TB maximum memory. The CPU on each machine on the cluster is about 2.00 GHz Intel(R) Xeon(R) CPU E5-2640 class.

A. Datasets

Our dataset consists of one million real-world movies, which are stored on HDFS, and totally includes 1024 blocks of data, with 64M size per block. The format of our input dataset is $<N, (C_1{:}W_{n1}, C_2{:}W_{n2}, ..., C_n{:}W_{nn})>$. Each item is composed of two parts. One is the ID of the item vector, and the other is the sparse vector of this item after the ETL preprocessing. The ID represents real movies id, and the first column of the sparse vector represents the tags of the movies, and the second column of the sparse vector represents the weight of given tag.

B. Evaluation on Partition-based and Inverted-index Approach

In this subsection, we compare our partition-based and inverted-index algorithm separately, and evaluate on the performance of the combination approach.

Firstly, according to the partition-based algorithm, we compare it with the brute-force (BF) algorithm. To implement the brute-force algorithm on Spark, we use Spark SQL. We firstly write our dataset into one NoSQL database, and use a SQL query like 'select sim(x, y) s from V va, V vb where $s > t$' to select all the similar pairs, where V is the vector lists, and t is the user-defined threshold.

Fig. 4 shows the running time of BF approach and partition-based approach for different input block sizes. In this experiment, we used 32 executors, with 20 cores and 32GB memory per executor, and separately tested the two solution

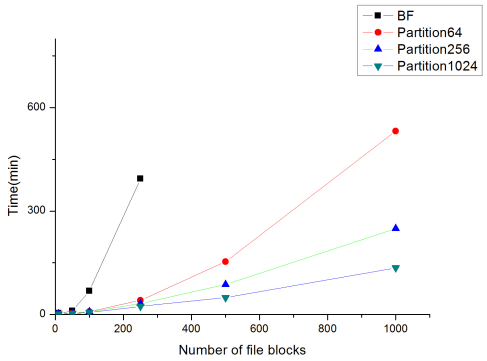

Fig. 4. Partition-based APSS approach

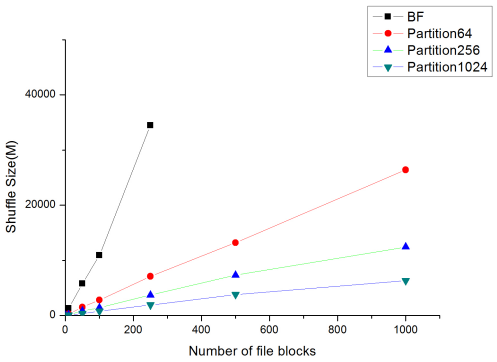

Fig. 5. Shuffle write on GroupByKey transformation

on 1-1024 blocks of dataset. It shows that the partition-based algorithm gains much faster than the BF algorithm. In our analysis in Section III, we drew a conclusion that partition-based algorithm can avoid much necessary network and I/O overhead. In Spark, most network and I/O communications happen in the shuffle process between different stages, which indicates the shuffle scale is determined by the network and I/O cost. As a result, we conduct an experiment on the scale of shuffle write in the GroupByKey transformation, in order to compare the network and I/O cost between the two algorithms. Fig. 5 shows the shuffle size of BF approach and partition-based approach for different input block sizes. The result indicates that the BF algorithm takes quadratic scale of shuffle write than the partition-based method, which is consistent with our analysis in Section III.

Secondly, we compare the inverted-index-based algorithm with BF algorithm. In this experiment, we used 1 core and 1G memory on Yarn cluster, and ran our experiments on a sub-dataset of 1-10 thousands scale of movies, in order to fairly compare the performance of the two algorithms.

The result is presented in Fig. 6, which shows the running time of BF approach and invert-index-based approach for different item sizes. It indicates that the inverted-index approach

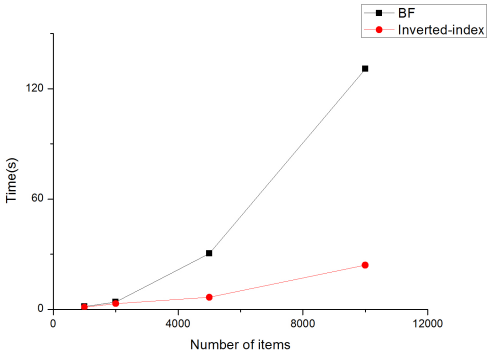

Fig. 6. Inverted-index approach

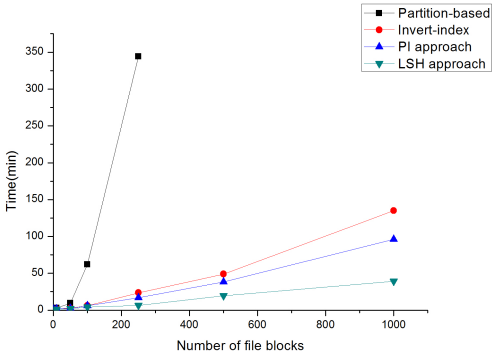

Fig. 7. The performance of APSS approaches

gains nearly 10x faster than the BF algorithm.

After the evaluation on partition-based approach and inverted-index-based approach separately, we conducted an evaluation on the performance of the combination approach (PI approach), in which we used 32 executors, with 20 cores and 32GB memory per executor, and separately tested the approaches on 1-1024 blocks of dataset, while we used 16 executors, with 8 cores and 24G memory per executor on the evaluation of the LSH approach discussed in next subsection. Fig. 7 shows the running time of different approaches for different input block sizes, and indicates that the combination approach performs better than the separate approach.

C. Evaluation on LSH Approach

In this subsection, we evaluate the performance and accuracy of LSH approach. In this experiment, we implemented our LSH approach on cosine similarity metrics, and set the default b (number of bands) to 8 and default r (number of fingers per band) to 8, and used 16 executors, with 8 cores and 24G memory per executor. As shown in Fig. 7, we can draw a conclusion that the LSH approach gains better performance with much less resources consumption.

In the next experiment, we tested our LSH approach on different similarity metrics with different numbers of b and

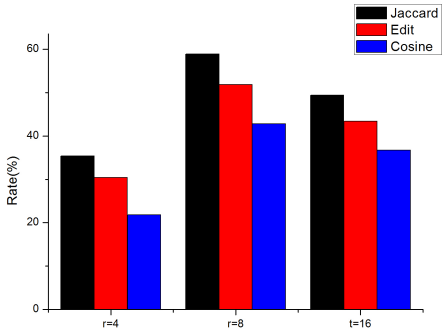

Fig. 8. The precision rate of LSH approach on different similarity metrics

r with the restriction that $b * r$ = 64, although our algorithm explicitly exploits the cosine similarity metrics tested in Fig. 7. In this experiment, we measure the accuracy of our approach by precision and recall rate. The computation of precision and recall rate is defined as (4) & (5), where $PredSet$ represents the set we predicted, and $RefSet$ represents the set of pairs whose similarity is above the user-defined threshold.

$$Precision = \frac{|PredSet \bigcap RefSet|}{|PredSet|} \tag{4}$$

$$Recall = \frac{|PredSet \bigcap RefSet|}{|RefSet|} \tag{5}$$

Fig. 8 shows the evaluation result of LSH approach with different numbers of b and r on different similarity metrics. It indicates that 1) the LSH approach perform better for large scale of data, due to the cost of accuracy. 2) LSH approach on Jaccard distance metric has better performance of precision and recall rate. But on cosine distance, the average precision and recall rate are not very well, as Anand Rajaraman discussed in [24].

After the evaluation of the performance of LSH approach, we find the precision and recall rate is not very well on typical LSH approach, especially on cosine similarity metric. As a result, in our work, we implement a filter-based LSH approach, which filters the false positive candidates, to improve the precision of our approach at less cost of recall rate, and then compare the accuracy with a Bayesian LSH approach based on [9].

Fig. 9 shows the F1 rate of different LSH approaches for different item sizes. The F1 rate is computed by precision and recall rate with (6).

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{6}$$

In this experiment, we tested filter-based LSH approach on cosine similarity metric, with the parameters that b=8 and r=8. The result indicates that the filter-based LSH approach observably improve the accuracy of LSH approach in comparison to Cosine LSH and Bayesian LSH [9].

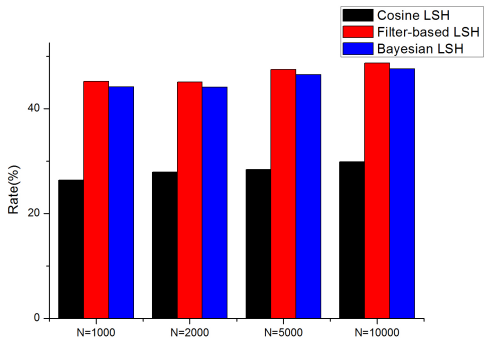

Fig. 9. The optimization of LSH approach

V. CONCLUSIONS

The contribution of this paper is the design and implementation of two effective parallel algorithms on distributed data processing framework Spark. From the evaluations, we find that 1) the partition-based solution runs 10x faster than the brute-force solution, due to the fact that it avoids more unnecessary network and I/O overhead among different partitions, 2) the combining approach based on average-partition and inverted-index gains better performance speedup than each separate technique, 3) the approach used locality-sensitive hashing works faster and takes much less resources, due to the dimension reduction from large scale to limited scale of hash functions, 4) with the false positive filtering, the result of approximation solution are made much more accurate.

ACKNOWLEDGMENT

We thank China Telecom for its assistance with the usage of Yarn cluster, and Shijiangzhuang Institute Engineering College for their assistance with our research.

REFERENCES

[1] M. Alabduljalil, X. Tang, and T. Yang. Optimizing parallel algorithms for all pair's similarity search. In ACM Inter. Conf. on Web Search and Data Mining, pages 203–212, 2013.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In Proc. of Inter. Conf. on World Wide Web, WWW '2007, pages 131–140. ACM.

[3] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with mapreduce. In ACL '2008, pages 265–268.

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

[5] E. Kayaaslan, S. Jonassen, and C. Aykanat. A Term-Based Inverted Index Partitioning Model. ACM Transactions on the Web (TWEB), 7(3):1–23, 2013.

[6] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In Proc. of ACM SIGIR'2009, pages 155-162.

[7] G. D. F. Morales, C. Lucchese, and R. Baraglia. Scaling out all pairs similarity search with MapReduce. In 8th Workshop on Large-Scale Distri. Syst. for Information Retrieval, 2010.

[8] F. Ture, T. Elsayed, and J. Lin. No free lunch: brute force vs. locality-sensitive hashing for cross-lingual pairwise similarity. In Proc. of SIGIR'2011, pages 943–952.

[9] V. Satuluri and S. Parthasarathy. Bayesian Locality Sensitive Hashing for Fast Similarity Search. PVLDB, 5(5):430–441, 2011.

[10] R. Vernica, M. Carey, and C. Li. Efficient Parallel Set-Similarity Joins Using MapReduce. In SIGMOD Conference, pages 495–506, 2010.

[11] Amit Awekar and Nagiza F. Samatova. Fast matching for all pairs similarity search. In International Conference on Web Intelligence and Intelligent Agent Technology Workshop, 2009.

[12] Shanzhong Zhu, Alexandra Potapova, Maha Alabduljalil, Xin Liu, and Tao Yang. Clustering and load balancing optimization for redundant content removal. In Proc. of WWW 2012 (21th international conference on World Wide Web), Industry Track, 2012.

[13] M. Persin, J. Zobel, R. Sacks-Davis (1994). Fast document ranking for large scale information retrieval. In Proc. of the First Int'l Conf. on Applications of Databases, Lecture Notes in Computer Science v819, 253–266.

[14] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications of the ACM, 51:117–122, 2008.

[15] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In SOCG, pages 253–262. ACM, 2004.

[16] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near duplicate detection. ACM Transactions on Database systems, 2011.

[17] S. Sarawagi and A. Kirpal, 'Efficient set joins on similarity predicates,' in ACM SIGMOD '04, Pages 743–754.

[18] A. Awekar, N. F. Samatova, and P. Breimyer. Incremental all pairs similarity search for varying similarity thresholds. In SNA-KDD, 2009.

[19] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. In KDD '05.

[20] M. Alabduljalil, X. Tang, and T. Yang. Load Balancing for Partition-based Similarity Search . In ACM Conference on Research and Development in Information Retrieval (SIGIR), pages 193–202, 2014.

[21] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. In Proceedings of KDD, 2013

[22] Y. Shi, M. Larson, and A. Hanjalic. Collaborative Filtering beyond the User-Item Matrix: A Survey of the State of the Art and Future Challenges. ACM Comp. Surv., 47(1), 2014.

[23] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012. 1

[24] Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman, Mining of Massive Datasets, Copyright c 2010, 2011, 2012, 2013

[25] Deepak Ravichandran, Patrick Pantel, Eduard Hovy, Randomized algorithms and NLP: using locality sensitive hash function for high speed noun clustering, ACL '05, pages 622–629

[26] Spark: Cluster Computing with Working Sets. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. HotCloud 2010. June 2010.

[27] Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. HotCloud 2012. June 2012.

[28] Marc Moreno Maza, Optimizing Algorithms and Code for Data Locality and Parallelism, University of Western Ontario, Canada, SHARCNET, 2013