International Journal of Database Theory and Application
Vol.8, No.3 (2015), pp.57-68
http://dx.doi.org/10.14257/ijdta.2015.8.3.06

An Efficient Parallel Top-k Similarity Join for Massive Multidimensional Data Using Spark

Dehua Chen[1], Changgan Shen[2], Jieying Feng[3] and Jiajin Le

Computer Science and Technology Academy, Donghua University, Shanghai, China
chendehua@dhu.edu.cn[1], dhshenchanggan@163.com[2]

Abstract

Top-k similarity join has been used in a wide range of applications that require calculating the most top-k similar pairs of data records in a given database. However, the time performance will be a challenging problem, as an increasing trend of applications that need to process massive data. Obviously, finding the top-k pairs in such vast amounts of data with traditional methods is awkward.

In this paper, we propose the RDD-based algorithm to perform the top-k similarity join for massive multidimensional data over a large cluster built with commodity machines using Spark. The RDD-based algorithm consists of four steps, which loads a set of multidimensional records stored in HDFS and finally output an ordered list of top-k closest pairs into HDFS. Firstly, we develop an efficient distance function based on LSH(Locality Sensitive Hashing) to improve the efficiency in pairwise similarity comparison. Secondly, to minimize the amount of data during the RDD running-time, we split conceptually all pairs of LSH signatures into partitions. Moreover, we exploit a serial computation strategy to calculate all top-k closest pairs in parallel. Finally, all the local top-k pairs sorted by their Hamming distances will contribute to the global top-k pairs. In this paper, the performance evaluation between Spark and Hadoop confirms the effectiveness and scalability of our RDD-based algorithm.

Keywords: *Massive multidimensional data; top-k similarity join; Spark; Resilient Distributed Datasets; Hadoop*

1. Introduction

Top-k similarity join, such query plays an important role in a wide range of applications including time series analysis, CAD, similarity search, social networks and link prediction, etc. In these domains, numerous real-world information can be presented as multidimensional data which contain a huge amount of useful and valuable information. For instance, assume that an IT company plans to recruit one project manager and one product manager. However, how to choose the best two is a hard work. Recently, a breakthrough pointis to analyzing applicants' professional social networks, like LinkeIn[1], a good and trendy way for employers to seek candidates. In paper [2], we know that there is an interesting observation in sociology: the more similar two individuals are, the greater the trust between them is. Thus, we expect that the project manager and the product manager have good trusting relationships. Plato observed in Phaedrus that "similarity begets friendship" [3]. Therefore, analyzing all the candidates' social network, we could find the top-1 pair of managers who are similar to each other if they have many common friends.

Finding top-k similar pairs in large multidimensional databases is a formidable problem as the vast amounts of multidimensional data usually do not fit in the main memory of one machine. Recently, many applications of big data analysis leverage cloud computing technologies in order to efficiently deal with this amount of data. Spark[4], a super-fast, open source large-scale data processing and advanced analytics engine in use at

ISSN: 2005-4270 IJDTA

Alibaba, Cloudera, Databricks, IBM, Intel, and Yahoo, among others, is a top-level project of the Apache Software Foundation now. And its programming model, Resilient Distributed Datasets (RDDs)[5], a distributed memory abstraction that lets programmers performin-memory computations on large clusters in a fault-tolerant manner, provides a restricted form of shared memory based on coarse-grained transformations. RDDprovides numerous abstractions for accessing a cluster's computational resources and efficiently leverage distributed memory. It can efficiently reuse intermediate results across multiple computations and even 100x faster than Hadoop[6]MapReduce[7] in memory, or 10x faster on disk[4].In this paper, we use RDD as the parallel data-processing framework to calculate top-k similar pairs of multidimensional data records in large databases.

Motivated by these, in this paper, we develop a scalable parallelized algorithm for the top-k similarity join over massive multidimensional data with LSH-based distance. However, when the multidimensional data quantity becomes more and more, computing the top-k similarity join on them immediately can be a big challenge. Our proposed algorithm leverages the idea of parallel computation in RDD to complete two main tasks before computing top-k similar pairs. The first task is to leverage an RDD to compute LSH signature for each point record in a parallel way. The second one is to split conceptually all pairs of LSH signatures into partitions such that every pair appears in a single partition only. After all pair partitioning, we can correctly find the top-k closest pairs by computing the top-k closest pairs in each partition separately. To improve the efficiency in finding the local top-k closest pairs in each partition, we propose the divide-and-conquer algorithm in traditional settings which will be used in each partition later when using the RDD framework. Based on all the top-k closest pairs from all partitions, we can select the final top-k closest ones. Even though we still compute overall Θ(n2) distance computations, the execution times of the top-k closest pair algorithms using RDD will be actually improved since Spark's good architecture and outstanding performance.

2. Background

2.1. Locality-sensitive Hashing

Locality-sensitive Hashing(LSH) is a method of performing probabilistic dimension reduction of high-dimensional data. The basic idea is that if two points are close together, their projection remains close. So the similar points hashed are mapped to the same buckets with high probability. Comparing with conventional hash functions, such as those used in cryptography, LSH aims to maximize probability of "collision" of similar points rather than avoid collisions.

Definition 1:ALSHfamily[8] f is defined for a metric space$M$with a threshold$r > 0$andanapproximationfactor$c > 1$.Thefamily$f$is a family of functions$h: M \to S$which m appoints from Mtoabuckets $\in S$. The LSH family satisfies the following conditions for any two points $p, q \in M$, using a function$h \in f$which is chosen uniformly at random:

If $d(p,q) \le r$, then $Pr_f[h(q) = h(p)] \ge P_1$ (1)

If $d(p,q) \ge cr$, then $Pr_f[h(q) = h(p)] \le P_2$ (2)

A family is interesting when it satisfies $P_1 > P_2$. We call such a family $f$as$f(c, r, P_1, P_2)$-sensitive.

2.2. Hamming Distance

Named after Richard Hamming, paper[9] introduced the fundamental of the Hamming distance. The hamming distance measures the number of positions at which the corresponding symbols are different between two strings of equal length. It has been used in some disciplines like information theory, cryptography, coding theory, and etc.

Definition 2:Let $H = \{0,1\}$. The d-dimensional Hamming Space H^d consists of bit strings of length d. Each point $p \in H^d$ is a string $p = (p_1, p_2, \dots, p_d)$ of zero's and one's. Given two points $p, q \in H^d$, the Hamming distance $d_H(p, q)$between them is the number of positions at which the corresponding strings differ, i.e.,$d_H(p, q) = |\{i: p_i \neq q_i\}|$.

2.3. Spark

Spark, a MapReduce-like cluster computing engine, extends the MapReduce model to better support two common classes of analytics apps: (1) iterative algorithms (machine learning, graphs); (2)interactive datamining. Unlike tranditional Map Reduce engines[10], Spark has several different features:

- Unlike the two-stage MapReduce topology,it has an advanced DAG execution engine that supports cyclic data flow.
- It provides an in-memory storage abstraction called ResilientDistributed Datasets (RDDs) that offers over 80 high-level operators that make it easy to build parallel apps, and automatically recover from failures.
- It optimizes the engine for low latency. Spark can efficientlymanage tasks on clusters ofthousands of cores in sub-second, while engines like Hadoop incur a latencyof 5 to 10 seconds to launch each task with the heart-beat mechanism.
- It also provides powerful integration with Hadoop ecosystem. Spark is easy to run standalone or on EC2or Mesos[11], and can read from HDFS, HBase, Cassandra, and any Hadoop data source.

In addition to those features above, Spark provides other difference features that contribute to its superior performance.

2.4. Resilient Distributed Datasets (RDDs)

Resilient distributed datasets (RDDs), the main abstraction of Spark, represents immutable, partitionedcollectionsthatcanbecreatedthroughvariousdata-parallelloperators (e.g., map, group-by, join).Each RDD is a read-only, partitioned collection of records and can only be created through deterministic operations on either (1) data in stable storage or (2)other RDDs.

RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset. All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. Table 1 lists several RDD transformations and actions available in Spark.

3. Problem Definition

Let $M: \{p_1, p_2, \dots, p_n\}$ be the d-dimensional dataset in which each point p_i is represented as $< p_{i(1)}, p_{i(2)}, \dots, p_{i(d)} >$, and $f: \{h_1, h_2, \dots, h_d\}$ be the LSH function family with the function $h_i(p) = \lfloor (a_i + b_i)/\omega \rfloor$, where a_i is a d-dimensional vector whose coordinates are

Table 1. Several Transformations and Actions Available on RDDs in Spark

Transformations	map(f : T)U) :	RDD[T])RDD[U]
	filter(f : T)Bool) :	RDD[T])RDD[T]
	Flat Map(f : T)Seq[U]) :	RDD[T])RDD[U]
	groupByKey() :	RDD[(K, V)])RDD[(K, Seq[V])]
	Reduce ByKey(f : (V;V))V) :	RDD[(K, V)])RDD[(K, V)]
	union() :	(RDD[T];RDD[T]))RDD[T]
	Partition By(p: Partitioner[K]) :	RDD[(K, V)])RDD[(K, V)]
	sort(c : Comparator[K]) :	RDD[(K, V)])RDD[(K, V)]
Actions	count() :	RDD[T])Long
	reduce(f : (T;T))T) :	RDD[T])T
	collect() :	RDD[T])Seq[T]

picked uniformly at random from a normal distribution, and b_i is a random variable uniformly distributed in the range $[0, \omega]$. Suposed$_H(p, q)$ is the Hamming distance.

We first define the LSH-based distance:

Definition 3: Given two points$p_1$andp_2, their LSH-based distance:

$D(p_1, p_2) = 1 - [Pr_f(h(p_1) = h(p_2))]$. (3)

Then, we give the definition of multidimensional signature:

Definition 4: For a given d-dimensional point p, its signature$S(p): < h_1(p), h_2(p), \dots, h_d(p) >$ is the concatenation of all the hash values of $h_1(p), h_2(p), \dots, h_d(p)$.

Next is the fefinition of Hamming distance of d-signatures:

Definition 5: Given the d-signatures$S(p_1)$ and$S(p_2)$ of two points p_1 andp_2, the Hamming distance is

$DH(S(p_1), S(p_2)) = \sum_{i=1}^{d} d_H(h_i(p_1), h_i(p_2))/d$, (4)

where $d_H(h_i(p_1), h_i(p_2)) = \begin{cases} 1, h_i(p_1) \neq h_i(p_2) \\ 0, h_i(p_1) = h_i(p_2) \end{cases}$.

Proposition 1: For two points$p_1$andp_2, their LSH-based distance $D(p_1, p_2)$is equal to the Hamming distance $DH(S(p_1), S(p_2))$of two signatures $S(p_1)$ and$S(p_2)$.

In summary, the computation of the LSH-based distance of multidimensional points can be transformed into the computation of Hamming distance of multidimensional points' signatures. And computing the top-k similarity join with Hamming distance over the multidimensional dataset, RTOP(k): $\{(p_{i(1)}, p_{j(1)}), (p_{i(2)}, p_{j(2)}), \dots (p_{i(k)}, p_{j(k)})\}$, needs to satisfy the conditions as follows:

$DH(S(p_{i(1)}), S(p_{j(1)})) \leq DH(S(p_{i(2)}), S(p_{j(2)})) \leq \dots \leq DH(S(p_{i(k)}), S(p_{j(k)}))$ holds.

For each pair $(p_{i(l)}, p_{j(l)}) \in$ RTOP(k), we have $i(l) < j(l)$and$i(l), j(l) \in R$.

For each pair $(i(l), j(l))$with$i(l) < j(l)$, $(i(l), j(l)) \notin$ RTOP(k)and $(i(l), j(l)) \in R$, we have $DH(S(p_{i(k)}), S(p_{j(k)})) \leq DH(S(p_{i(l)}), S(p_{j(l)}))$.

4. RDD-based Algorithm

4.1. Algorithm Overview

As mentioned previously, computing the LSH-based distance of two multidimensional points can actually be transformed into computing the Hamming distance of theird-signatures. Considering that when the multidimensional data is massive, computing directly the top-k similarity join on them will be a big challenge, we divide our RDD-based algorithm into four steps, each of which owns its RDD transformations or actions. First, we compute each point's signature $S(p)$. Second, we use BKDRhash[12] function to compute the hash-values of each point's signature $S(p)$ by which all points

will be distributed to different buckets. Third, we do the groups of buckets by merging two different buckets to one. Finally, we calculate the local top-k similar pairs in each group by using a parallel divided and conquer algorithm TopK-DC, and then do the globle top-k pairs.

4.2. Computing Each point's Signature $S(p)$

The original data are organized in a relational table which consists two columns: the ID of multidimensional point and its data record. And all the records here form thedataset $M:\{p_1, p_2, \dots, p_n\}$. The first step is to launch a RDD transformation for computing each point's signature $S(p)$. Figure. 1 shows the dataflow of this step.

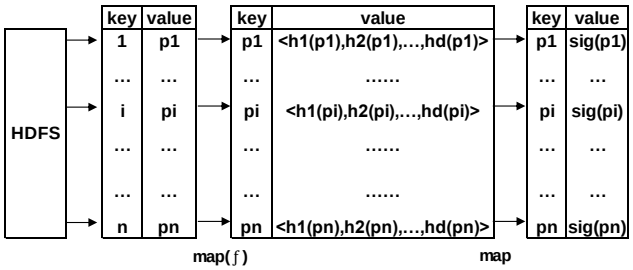

Figure1. The Dataflow of Signature Computation

As showed in Figure. 1, a RDD will be created by multidimensional data files stored in the Hadoop distributed file system (HDFS). The RDD sequentially reads each point from the input split, and then uses the LSH function families mentioned in the section 2.1 to compute each point's signature $S(p)$. For each point, the RDD executes a map transformation to output key-value pairs $< i, h_1(p), h_2(p), \dots, h_d(p) >$, then another map transformation will perform the concatenation of d hash values to signature $< i, S(p_i) >$.

4.3. Data Division

The step is to divide the multidimensional data into a number of buckets by hashing their signatures. The dataflow of this step is shown in Figure. 2.

First, we need to define a hash function which maps multidimensional records into different buckets in the RDD map transformation. In this paper, we use the BKDRhash[12] function $BH()$ as the hash method because of its higher speed and less collision. With the hash function, the map(BH) transformation maps the key-value pairs $< i, S(p_i) >$ into key-value pairs $< BH(S(p_i)), p_i >$by performing hash calculation over each signature $S(p_i)$ for each point p_i. Then a groupByKey transformation will be executed to group the values into a hashing bucket Bi with $1 \le i \le m$ by the same key. Finally, this transformation outputs key-value pairs $< B_i, list[B_i] >$, where list[Bi] is the list of data records in the same bucket Bi.

In conclusion, we formalize the map tasks and reduce tasks in the step below:

map(BH): <key1=ri,value1=sig(pi)>
→<key2=BH(sig(pi)),value2=pi>

groupByKey: <key2=BH(sig(pi)) ,value2=pi> →<key3=Bi ,value3=list[Bi] >

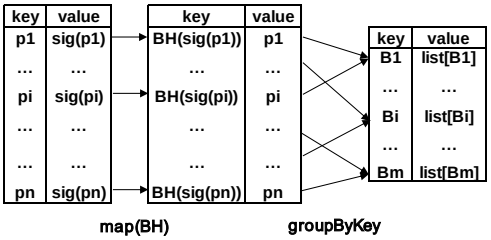

Figure 2. The Dataflow of Data Division

4.4. Bucket Group

Based on the results of the previous step, this step is to generate all pairs of data records by combining two buckets Bi and Bj into one pair$< B_i, B_j >$, where i<j. Figure. 3 shows the dataflow of bucket group.

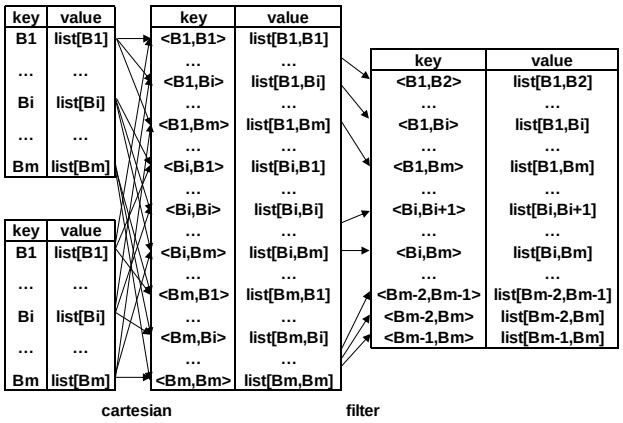

Figure 3. The Dataflow of Bucket Group

Specially, a transformation named cartesian, creating a Cartesian production of two RDDs, can contribute the combination of two buckets. In doing so, we obtain the buckets combinations, but with the by-product of duplication. Fortunately, the filter transformation is really available to remove the duplicates. Finally, the transformation outputs key-value pair <(Bi,Bj),list[Bi]> where i≤j≤m.

In conclusion, we formalize the cartesian and filter transformation in the step below:

cartesian: <key1=Bi,value1=list[Bi]> →<key2=(Bi,Bj),value2=list[Bi,Bj]>(1≤i,j≤m)

filter: <key2=(Bi,Bj),value2= list[Bi,Bj]> →<key3=(Bi,Bj),value3=list[Bi,Bj]>(1≤i≤j≤m)

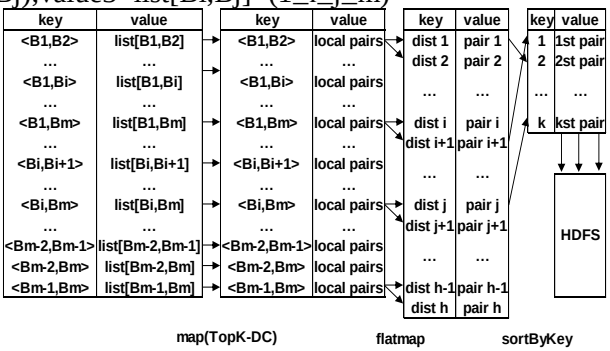

Figure 4. The Dataflow of Calculating Top-k Pairs

4.5. Calculating Top-k Similar Pairs

The final step is to calculate the top-k similar pairs of multidimensional records by using all bucket group results. We first use a RDD transformation with TopK-DC function to calculate the local top-k closest pairs in each bucket group. And then the flat Map transformation outputs the global closest pairs to the sort ByKey transformation to sort by their Hamming distance. Subsequently, the top-k pairs are our goals and will be saved into HDFS. The dataflow of this step is shown in Figure. 4.

To improve the efficiency in finding local top-k closest pairs, we propose a divide-and-conquer algorithm TopK-DC to do calculation in parallel. Thus, each multidimensional record becomes one point in a d-dimensional space. In such d-dimensional space, the TopK-DC algorithm divides the dataset M into two sides by a hyperplane and to find the top-k closest pairs on each side recursively. To compute the distances of the pairs crossing the hyperplane, the TopK-DC algorithm invokes recursions with next splitting dimensions of signatures until there remains only a single dimension not split yet so that we can consider only constantnumber of distance computations for each multidimensional record when wecompute the distances of the pairs crossing the hyper plane. The pseudo code of TopK-DC is presented in Figure. 5. The TopK-DC takes the dimension lof signatures S(p)sfor dividing M,k, and (d-l+1) arrays, Xl, ..., Xdas its input values. All arrays Xiwith l≤i≤d contain the same and are sorted by thei-thdimension of signatures S(p)s.

Function TopK-DC (l, k, X_l, … , X_d)

(1) Check the boundarycondition whether $|X_l| \leqslant 2$. If the condition is true, it justreturns the pairs.
(2) Divide the input multidimensional record into two partitions{XR_l, … ,XR_d} and {XL_l, …, XL_d}.
(3) Invoke TopK-DC recursively foreach partition with the same size,H_L = TopK-DC (l, k, XR_l, … ,XR_d) and H_R = TopK-DC (l, k, XL_l, … , XL_d).
(4) Set $H = H_L \cup H_R$.
(5) Find the k-th similar pair $H(k)$ in H and its distance $\delta = DH(H(k))$ using Eq. (4).
(6) Calculate all the pairs whose distances is at most the distance of the k-th pair from recursive calls TopK-DC(l+1, k, B_{l+1}, … , B_d).

Figure 5. The Topk-DC Algorithm

The TopK-DC algorithm is implemented in the map(TopK-DC) transformation in order to find the local top-k join pairs efficiently in each bucket group. Then the flatMap transformation flattens all the local pairs to global closest pairs. Given all the local top-k join pairs, the sort ByKey transformation finally sorts all of them by their Hamming distances so that we can find the final top-k closet pairs.

5. Experiments

5.1. Methodology and Cluster Setup

The experiments run on an8-node cluster with 4 cores, 4GB of RAM running Ubuntu 12.10 operating systems. All algorithms were implemented using Javac Compiler of version 1.7. The new version Spark-0.91 and scala-2.11.0 are chose in this paper. In order to study the performance of our algorithms using Spark, we also do another implementation in Hadoop-1.1.2 with MapReduce. We measure the performance in term of execution time as well as speedup and scaleup[13].

We test several multi-dimensional datasets with varied dataset size in our experiments. Specially, we generate several synthetic datasets with varying the number of data records from 10,000 to 200,000.

5.2. Performance Evaluation

In our experiments, we evaluate the effects of the dataset size n, the result number k, and the number of machines s using Spark. The default setting for parameters is: n =100,000, k = 20 and s = 6.

5.2.1. The Effect of k

We now evaluate the effect of k on the performance of our proposedtechniques. Figure.6 presents the running time by varying k from 10 to 80.As the graph confirms, theperformances of our algorithm do not degrade that much as k increases. And Spark with persistence in memory only spends one half of the time less than Hadoop does, and Spark with non-persistence also has a better performance than Hadoop does.

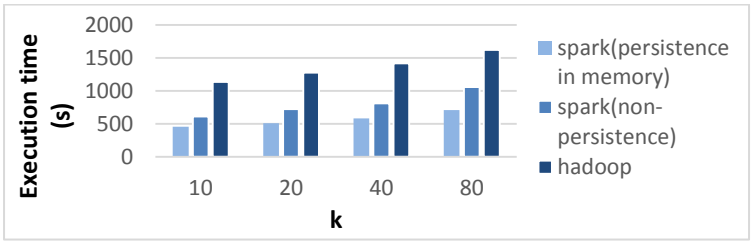

Figure 6. Varying k

5.2.2. The Effect of n

We now study the scalability of RDD-based algorithm by changing the number of multidimensional recordsn from 10,000to 200,000. The execution times are shown in Figure.7.Obviously, we can see that the overall execution time of our algorithm increases quadratically when we enlargethe data size. This is determined by the fact that the number ofmultidimensionalrecord pairs increase quadratically with the data size.Even though both of the time overhead in Hadoop and Spark increasequadratically, Spark consumes less time because of its better framework.

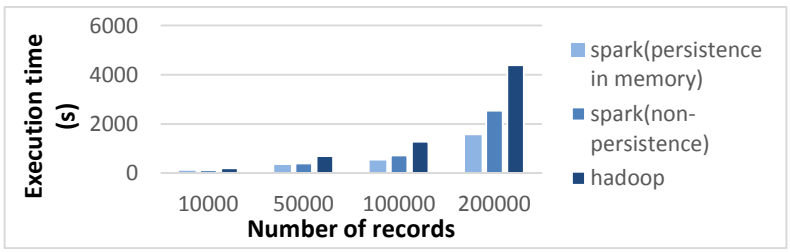

Figure 7.Varying n

5.2.3. Effect of s

We varied the number of machines s from 2 to 8 in our experiments.Figure.9 shows us that as the numberof machines increases, the performances of our algorithmsare also improved. Specially, Spark has a remarkable performance than Hadoop really does.

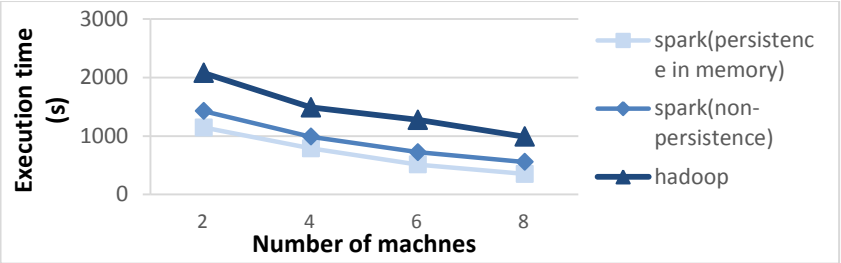

Figure 8. Varying s

6. Related Work

Performing similarity joins of multidimensional data in the traditional setup has been extensively studied in the literature [14, 15, 16]. Nevertheless, these previous works focus on a single machine and performs similarity joins on spatial indexes like R-trees, hash functions, *etc*. The work in [15] first built an R-tree on each database, then traversed R-trees in depth-firstapproach, and finally got the similar pairs in leaf nodes. However, the time complexities of these similarity joins algorithms grow exponentially. In general, given a threshold, a similarity join algorithm can use inverted indexes for pruning, such as [17,18,19]. For similarity join queries, [20] proposed several similarity measures like cosine distance, and Jaccard coefficient. Unfortunately, these algorithms have no scalability for large data since they mainly assume that all data can be loaded into main memory.

Nowadays, studies about similarity join of multidimensional data over MapReduce, a popular model for large-scale data processing, develop increasingly. [21] proposed novel algorithms to execute kNN joins efficiently on large data stored in a MapReduce cluster. In [26], they studied the problem of the top-k closest pair problem with Euclidean distance using MapReduce and presented scalable MapReduce algorithms. However, restricted by the blemish of Hadoop framework, such as high-latency, no control of data co-partitioning, lack of optimization based on data statistics, much overhead of task scheduling and launch, similarity join queries can't achieve a higher efficiency.

7. Conclusion

This work studies parallel top-k similarity join queries over large multidimensional data using Spark. We propose a four-step approach and explore several solutions to improve the efficiency of computation. We first introduce a LSH-based distance function for efficient multidimensional similarity computation. We next adopt all pair partitioning method to divide the data into different partition and we also leverage serial computation strategy for answering top-k closest pairs by only checking point pairs in parallel within each partition. By experiment results, this paper shows the better effectiveness and scalability of our RDD-based algorithms than Hadoop's.

References

[1] Z.Abbassi, V.S. Mirrokni,"A recommender system based on local random walks and spectral methods", Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis. ACM, **(2007)**, pp.102-108.
[2] M.McPherson, L.Smith-Lovin, "Cook J M. Birds of a feather: Homophily in social networks", Annual review of sociology, **(2001)**, pp.415-444.
[3] Plato, "Plato: In Twelve Volumes", Harvard University Press, **(1982)**.
[4] Apache, Spark, http://spark.apache.org, **(2014)**.
[5] M.Zaharia, M.Chowdhury, T.Das, A.Dave, J.Ma, M.McCauley, M.Franklin, S.Shenker, and I. Stoica,"Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, **(2012)**.

[6] Apache, Hadoop, http://hadoop.apache.org, **(2013)**.
[7] J.Dean, S. Ghemawat,"MapReduce: simplified data processing on large clusters", Communications of the ACM,**(2008)**, vol.51, no 1, pp.107-113.
[8] A. Gionis, P. Indyk, R.Motwani,"Similarity search in high dimensions via hashing", VLDB(1999), vol.99,pp.518-529.
[9] R. W. Hamming,"Error detecting and error correcting codes", Bell System technical journal, **(1950)**, vol.29, no. 2, pp.147-160.
[10] R. S. Xin, J. Rosen, M. Zaharia, "Shark: SQL and rich analytics at scale", Proceedings of the 2013 international conference on Management of data. ACM, **(2013)**, pp. 13-24.
[11] B. Hindman, A. Konwinski, M. Zaharia, "Mesos: A platform for fine-grained resource sharing in the data center", Proceedings of the 8th USENIX conference on Networked systems design and implementation, **(2011)**, vol. 11, pp. 22-22.
[12] C. Henke, C. Schmoll, T.Zseby,"Empirical evaluation of hash functions for multipoint measurements", ACM SIGCOMM Computer Communication Review, vol. 38, no. 3, **(2008)**.
[13] D. DeWitt, J.Gray,"Parallel database systems: the future of high performance database systems",Communications of the ACM, vol. 35,no. 6, **(1992)**.
[14] T. Brinkhoff, H. P. Kriegel, B. Seeger,"Efficient processing of spatial joins using R-trees", **(1993)**.
[15] Y. W. Huang, N. Jing, E. A. Rundensteiner,"Spatial joins using R-trees: Breadth-first traversal with global optimizations", VLDB, **(1997)**,vol. 97, pp. 25-29.
[16] M. L. Lo, C. V.Ravishankar,"Spatial hash-joins", ACM SIGMOD Record, **(1996)**,vol. 25, no. 2, pp. 247-258.
[17] R. J. Bayardo, Y. Ma, R.Srikant,"Scaling up all pairs similarity search", Proceedings of the 16th international conference on World Wide Web, ACM, **(2007)**, pp. 131-140.
[18] S. Chaudhuri, V. Ganti, R.Kaushik,"A primitive operator for similarity joins in data cleaning", Data Engineering, 2006. ICDE'06, Proceedings of the 22nd International Conference on. IEEE, **(2006)**, pp. 5-5.
[19] S. Sarawagi, A. Kirpal,"Efficient set joins on similarity predicates", Proceedings of the 2004 ACM SIGMOD international conference on Management of data, ACM, **(2004)**,pp. 743-754.
[20] C. Xiao, W. Wang, X. Lin,"Efficient similarity joins for near-duplicate detection", ACM Transactions on Database Systems (TODS),vol. 36, no. 3,**(2011)**, p. 15.
[21] C.Zhang, F. Li, J.Jestes,"Efficient parallel kNN joins for large data in MapReduce", Proceedings of the 15th International Conference on Extending Database Technology, ACM, **(2012)**.
[22] Y. Kim, K.Shim,"Parallel top-k similarity join algorithms using MapReduce", Data Engineering (ICDE), 2012 IEEE 28th International Conference on. IEEE, **(2012)**.

Authors

Chen Dehua, he is currently an associate professor in School of Computer Science and Technology, Donghua University, Shanghai, China. His research interests include big data management, high dimensional data query processing and information diffusion in social network. He was a visiting scholar at University of Stavanger, Norway.

Shen Changgan, he is currently working toward the master degree in School of Computer Science and Technology, Donghua University, Shanghai, China. His research interests include big data query processing and time series data management.

Feng Jieying, he is currently working towards the master degree in School of Computer Science and Technology, Donghua University, Shanghai, China. Her research interests include big data query processing, natural language processing, and data mining.

Le Jiajin, he is currently a professor of computer science and software engineering at Donghua University, Shanghai, China. He serves as the research director of the Center for Medical Wisdom at Donghua University. His main research interests include database systems, data warehousing, big data systems and medical wisdom.