

An Efficient Data Indexing Approach on Hadoop Using Java Persistence API

Yang Lai, Shi Zhongzhi

► **To cite this version:**

Yang Lai, Shi Zhongzhi. An Efficient Data Indexing Approach on Hadoop Using Java Persistence API. Zhongzhi Shi; Sunil Vadera; Agnar Aamodt; David Leake. 6th IFIP TC 12 International Conference on Intelligent Information Processing (IIP), Oct 2010, Manchester, United Kingdom. Springer, IFIP Advances in Information and Communication Technology, AICT-340, pp.213-224, 2010, Intelligent Information Processing V. <10.1007/978-3-642-16327-2_27>. <hal-01055056>

HAL Id: hal-01055056

<https://hal.inria.fr/hal-01055056>

Submitted on 11 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



An Efficient Data Indexing Approach on Hadoop using Java Persistence API

Yang Lai^{1,2}, Shi ZhongZhi¹

¹The Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China. ²Graduate University of Chinese Academy of Sciences, Beijing 100039, China. {yanglai, shizz}@ics.ict.ac.cn

Abstract. Data indexing is common in data mining when working with high-dimensional, large-scale data sets. Hadoop, a cloud computing project using the MapReduce framework in Java, has become of significant interest in distributed data mining. To resolve problems of globalization, random-write and duration in Hadoop, a data indexing approach on Hadoop using the Java Persistence API (JPA) is elaborated in the implementation of a KD-tree algorithm on Hadoop. An improved intersection algorithm for distributed data indexing on Hadoop is proposed, it performs $O(M+\log N)$, and is suitable for occasions of multiple intersections. We compare the data indexing algorithm on open dataset and synthetic dataset in a modest cloud environment. The results show the algorithms are feasible in large-scale data mining.

Keywords: Data Indexing, KD-tree, Data Mining, Distributed applications, JPA, ORM, Distributed file systems, Cloud computing

1. Introduction

Many approaches have been proposed for handling high-dimensional and large-scale data, in which query processing is the bottleneck [1]. Business intelligence and data warehouses can hold a Terabyte or more of data. Cloud computing has emerged for the subsequently increasing demands of data mining. MapReduce is a programming framework and an associated implementation for large data sets [2].

A concise indexing Hadoop implementation of MapReduce is presented in McCreadie's work [3]. Ralf proposes a basic program skeleton to underlie MapReduce computations [4]. Moretti presents an abstraction for scalable data mining, in which data and computation are distributed in a computing cloud with minimal user efforts [5]. Gillick uses Hadoop to implement query-based learning [6].

Most data mining algorithms are based on object-oriented programming (OOP), which runs in memory. Researchers elaborate many of these methods [7-10].

However, the following features in the MapReduce framework are unsuitable for data mining. First, in globality, map tasks are irrelevant to each other, as are reducing tasks. Data mining requires that all of the training data be converted into a global model, such as a KD-tree or clustering tree. The tasks in the MapReduce framework only handle its partition of the entire data set and output its results into the Hadoop distributed file system (HDFS). Second, random-write operations are disallowed by the HDFS, thus disabling link-based data models in Hadoop, such as linked-lists, trees, and graphs. Finally, the duration of both map and reduce tasks are based on scanning pro-

cessing, and will end when the partitioning of the training dataset is finished. Data mining requires a persistent model for following testing processing.

A database is an ideal persistent repository for objects generated by data mining using Hadoop tasks. A data mining framework on Hadoop using the Java Persistence API (JPA) and MySQL Cluster is proposed [11]. To mine high-dimensional and large-scale data on Hadoop, we employ Object-Relation Mapping (ORM), which stores objects whose size may surpass memory limits in a relational database. The Java Persistence API (JPA) provides a persistence model for ORM [12]. A distributed database is a suitable solution to ensure robustness in distributed handling. MySQL Cluster is designed to withstand any single point of failure [13], which is consistent with Hadoop.

We performed the same work that McCreddie's performed [3] and now propose a novel indexing Hadoop implementation for continuous values. Using JPA and MySQL Cluster on Hadoop, we propose an efficient data mining framework, which is elaborated by a KD-tree implementation. We also propose an improved intersection algorithm for the distributed data indexing on Hadoop [11], which can be suitable for many situations.

The rest of the paper is organized as follows. In Section 2 we elucidate the related index structures, flowchart and algorithms. Section 3 proposes the KD-tree indexing on Hadoop and the improved intersection algorithm. Section 4 provides descriptions of our experimental setting and results. In Section 5, we offer conclusions and suggest possible directions for future work.

2. Related Work

2.1. Naïve Data Indexing [11]

Data indexing is necessary for querying data in classification or clustering algorithms. Data indexing can dramatically decrease the complexity of querying.

2.1.1. Definition

A $(n \times m)$ dataset is an $(n \times m)$ matrix containing n rows of data, which contain m columns of float numbers. Let $min(i)$, $max(i)$, $sum(i)$, $avg(i)$, $std(i)$, $cnt(i)$ ($i \in [1..m]$) equal the minimum, maximum, sum, average, standard deviation and count of the i -column, respectively—i.e., the essential statistical measures of the i -column of the dataset. Let $sum2(i)$ be the dot product of the i -column, which is used for $std(i)$. From the $ZSCORE(x) = \frac{x - \text{Erreur}!}{\text{Erreur}!}$ [7], the formula $YSCORE(x) = \frac{x - \text{Erreur}!}{\text{Erreur}!}$, is defined, where $iStep$ is an experimental integral parameter.

2.1.2. Simple Design of the Inverted Index

In general, high-dimensional datasets are always stored as tables in a sophisticated DBMS for most data mining tasks. Figure 1 shows the flowchart in which the whole procedure is illustrated for indexing this kind of dataset and similarity querying [11]:

Discretization calculates the essential statistical measures, then performs YSCORE binning on the FloatData.TXT file (CSV format) and outputs the IntData.TXT file in which each line is labeled with a unique number (LineNo) consecutively.

The encoder transforms the text format file IntData.TXT into the Hadoop Sequence File format IntData.SEQ and generates IntData.DAT. To locate a record for a query rapidly, the length of the record has to be equivalent. The IntData.DAT contains an

($n \times m$) matrix, i.e., n rows of records containing m columns of IDs of bins, just as does IntData.SEQ.

With the indexer, the index files Val2NO.DAT and Col2Val.DAT are outputted with MapReduce from the IntData.SEQ file. To obtain the list of LineNo's for a specific bin, it is needed to store the list in a structural file (Val2NO.DAT); the address and the length of the list should be stored in another structural file (Col2Val.DAT). The two structural files are easy to access by offset.

The searcher, given a query—which should be a vector with m float values— performs YSCORE binning with the same statistical measures and yields a vector with m integers for searching; it then outputs a list of the number of records.

In intersection, the positions of each integer in Val2NO.DAT for a particular query will be extracted from Col2Val.DAT, and the lists of LineNo's for each integer can be found. The result will be computed by the intersection of the lists. The algorithm for improved intersection is elucidated in (3.1).

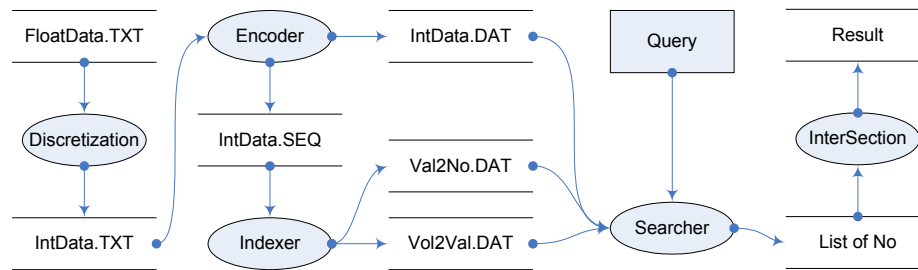


Figure 1 Flowchart for indexing and searching on Hadoop

2.1.3. Statistics Phase

In Hadoop, data sets are scanned once to calculate the four distributive measures, $min(i)$, $max(i)$, $sum(i)$ and $sum2(i)$ as well as $cnt(i)$ for the i -column; it then computes the algebraic measures of average and deviation. Therefore, a $(5 \times m)$ array is used for the keys in the MapReduce process: the first row contains the keys of the minimum of a column; the second row contains the keys for the maximum; the third contains the keys for the sum; the fourth contains the keys for the square of the sum; and the fifth contains the keys for the row count.

The algorithm performs $O(N/mapper)$, where N represents the number of records in the data sets, and $mapper$ is the number of the task of map functions. All partitions will have been calculated into the five distributive measures, the reduce function merges them into global measures. Keys for the map function are the positions of the next line, and values are the next line of text.

2.1.4. YSCORE phase

In Figure 2, the maximum number in the bins reduces to a reasonable level, by increasing the $iStep$. The lowered numbers administer to inverted indexing.

These interval values are put into bins using YSCORE binning [7]. Then, the inverted index will be of the format: the first column holds all fields in the original dataset; the second holds the $\langle bin, RecNoArray \rangle$ tuples.

The YSCORE binning phase does not need a reduce function, thus eliminating the cost of sorting and transferring; the result files are precisely the partitions on which map functions work. Keys in the map function are the positions of the next line, and values are the next line of text. The input file will be FloatDate.TXT, and the output file will be IntData.TXT, as shown in Figure 2. After the statistics' phase and the YSCORE binning phase, Hadoop can handle discrete data as in the commonly-used MapReduce word count example.

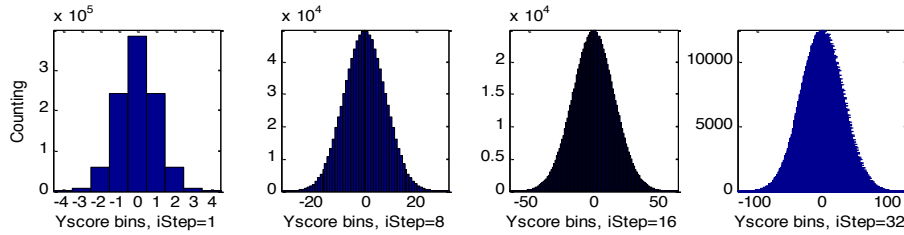


Figure 2 YSCORE binning of 1,000,000 values from a normal distribution with mean 0 and standard deviation 1. These YSCORE binnings use iStep of 1, 8, 16, and 32, respectively.

2.2. Data Mining Framework On Hadoop [11]

To apply common object-oriented data mining algorithms in Hadoop, the three problems concerning globalization, random-write and duration, posed in the introduction, should be handled. For data mining with Hadoop, a database was an ideal persistent repository for handling the three problems mentioned. In the data mining framework, a large-scale dataset is scanned by Hadoop. The necessary information is extracted using classical data mining modules, which will be embedded in the Hadoop map or reduce tasks. The extracted objects, such as lists, trees, and graphs, will be sent automatically into a database—in this case, MySQL Cluster.

3. Method

3.1. Improved Intersection

Searching in the naïve data indexing is based on intersection of multiple index files [11]. In general, the lengths of each index file are not the same order of magnitude. We propose an improved algorithm for this kind of unbalanced intersection.

The length of an intersection is always less than or equal to the two candidates. Figure 3 shows an example of intersection of ten arrays. Let δ be the intersection coefficient; Let x be the length of the first candidate in Figure 3. Therefore the lengths of intersections of multi-arrays are proximately equal to a geometric progression: $x, x\delta, \dots, x\delta^{n-1}$. For best performance, the arrays should be in descending order; the x should be the shortest. If the number of arrays is large enough, the length of the final intersection will be significant small.

The algorithm first checks whether the proportion between the two candidates is beyond a threshold $iTimesLength$. If no then a normal intersection turns out, else the improved intersection will be performed. The proportion determine which array is larger to be applied in binary search. All the elements in the smaller will be scanned, while only a few elements in the larger will be checked. Let m be the length of the

smaller, n for the larger. The complexity is $O(m+\log(n))$. If the lengths of the two candidates are not the same order of magnitude, the algorithm will perform better significantly. In fact, high-dimensional datasets always give rise to unbalanced intersection on Hadoop.

Though the algorithm performs better than the naïve data indexing, a flaw is that the procedure of intersection can not be paralleled to MapReduce model effectively.

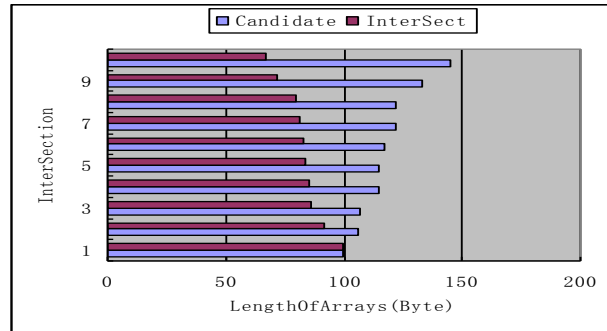


Figure 3 Abstract of improved intersection between multi-Arrays. The blue lower bars are the original arrays in ascend order for optimal complexity. The red upper bars are the results of intersection. The 10th red bar means the final intersection of the ten arrays.

3.2. KD-tree on Hadoop

KD-tree is a traditional data indexing algorithm. Bentley [9] discusses it in detail. The naïve KD-tree algorithm divides high-dimensional dataset on each dimension according to the split value in a defined criterion (**Figure 4 a**).

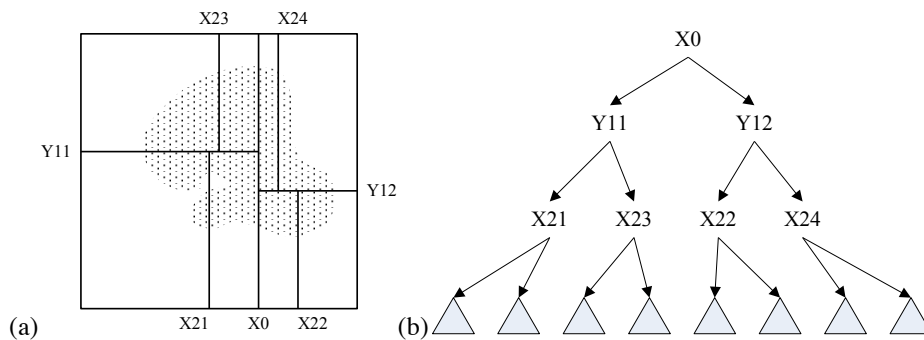


Figure 4 Dataset in 2-space stored in JPA. The dotted area in (a) denotes a dataset. The labels denote the split points. The tree in (b) denotes the 2-d tree in JPA generated from the dataset in (a). The triangles denote the basic blocks of the dataset, which are the directories in HDFS.

3.2.1. Hierarchical Bucket

This algorithm handles large-scale datasets on Hadoop, applies the naïve KD-tree algorithm to small datasets under the limit of memory (**Figure 4**). Let $iMemoryLimit$ be the limit of memory, a KD-tree will be divided into 4 levels, bucket 0, 1, 2, 3. The triangles denote bucket-1, which just below the $iMemoryLimit$. The bucket-0 means the internal nodes beyond the triangles, which denote the sub dataset whose size is too

large to fit in memory. The bucket-2 and bucket-3 are not shown in **Figure 4**, which are exactly the in-memory nodes described by Bentley [14].

Let R be the ratio of dataset to iMemoryLimit, the number of levels for bucket-1 is $\lceil \log_2 R \rceil$, and the number of the internal bucket-0 nodes, i.e. the number of splitting on Hadoop, is $2^{\lceil \log_2 R \rceil}$, approximately R . A node in KD-tree using JPA can be represented by the following MySQL commands:

```
CREATE TABLE `KD-tree` (
  `id` INT(11) PRIMARY KEY, `cnt` INT(11), `mean` INT(11),
  `std` INT(11), `attribute` INT(11), `median` FLOAT,
  `childleft` INT(11), `childright` INT(11),
  `bucket` TINYINT(3), `block` VARCHAR(1000));
```

Where 'id' is the primary key for each node, i.e. the reference for an object in OOP; 'attribute' is to tell which column should be concerned; 'median' is the balanced split-point; 'bucket' is to show what type of the node; 'block' is important for Hadoop by providing the HDFS path of the dataset represented by the node.

3.2.2. Median Selecting

A balanced KD-tree needs the median of each subset. As shown in the textbook [7], median is a holistic measure that is notable in complexity. However, there is a method to approximate it in $O(n)$. Let D be a dataset, D is grouped in bins according to an attribute and the frequency of each bin is known. The approximate median of the attribute is given by

$$\text{Erreur !} \tag{1}$$

Where *median* is the median of a field and L_m is the lower boundary of the bin in which the *median* is; f_m is the frequency of the bin in which the *median* is; f_{less} is the frequency of the bin in which all values are below the *median*; N is the size of D ; *width* is the width of a bin. [7]

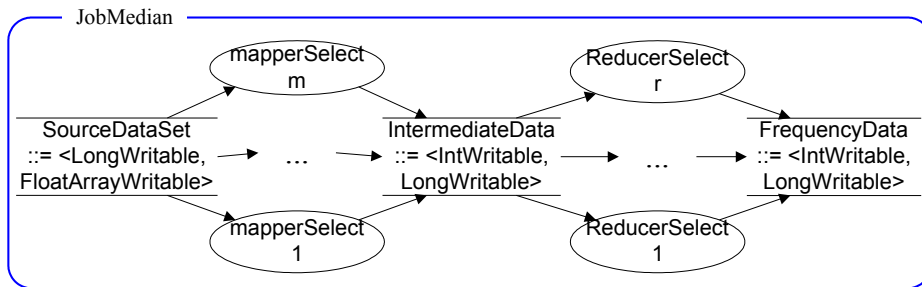


Figure 5 The Data Flow Diagram of Median Selecting in KD-tree on Hadoop

In Figure 5, JobMedian will scan a dataset once and the YSCORE (2.1.4) can be applied in this situation. The $avg(i)$ and $std(i)$ of each subset will be ready for the best attribute, then the subset will be grouped in bins whose width is $std(i)/iStep$ in $mapperSelect(1..m)$. The frequencies of the bins will be similar to Figure 2. Finally, one $reduceSelect$ collects the numbers from each map task, and outputs the median V_0 of

the dataset according to formula (2). The notable thing is the mapperSelects handle the huge dataset once, after then the reduceSelect calculate a few numbers. The sorting overhead is trivial compared with scanning. In JobMedian, only read operations occur with full speed.

3.2.3. Splitting

JobSplit contains only mapperSplit tasks, which do not yield normal HDFS output using the method `output.collect()` (Figure 6); instead, it writes directly to two sequence files in the HDFS to eliminate unwanted sorting I/O overheads in reduce task. The SourceDataSet is D ; the ChildDataLeft is D_1 ; the ChildDataRight is D_2 ; and all of these are, in fact directories in HDFS, which contain the outputs from each mapperSplit task.

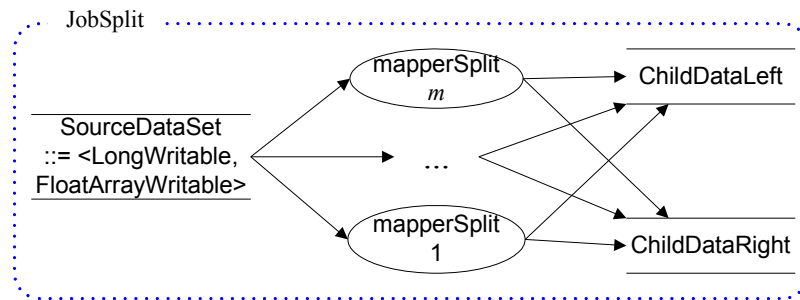


Figure 6 The Data Flow Diagram of Split of KD-tree on Hadoop

Each node in the tree has to contain a member ATTRIBUTE and a member MEDIAN. Given a test record (an array of continuous values), the former tells which field should be checked, and the latter tells which child node should be selected. If the test record is less than or equal to the member VALUE in the member ATTRIBUTE, then the left child node is selected, otherwise the right child node is selected.

As shown in **Figure 4**, the internal nodes in KD-tree will be persisted in database use JPA, and external nodes (denoted by triangles) will be directories in HDFS.

The statistics algorithm runs in JobSplit on both sub dataset. The results $sum(i)$, $sum2(i)$ and $cnt(i)$ of each attributes of each sub dataset on each mapper node will be sent by JPA to calculate $avg(i)$, $std(i)$, the attribute with the largest $std(i)$ will be selected as the best.

Before splitting, a node in JPA will be created corresponding to D , with the best attribute A_0 and best splitting value V_0 . While scanning the dataset D , each mapper yields two separate files to HDFS, read and write operations occur.

3.2.4. Merging

By default, each mapper handles a segment of a file no more than 64MB, the number of split files may increase exponentially after JobSplit. Instead of using normal Hadoop Path as input, the list of Paths will be used as input of Mappers. This makes each mapper to scan multiple files, not a single 64MB block. Meanwhile, the average size of files will be checked, if files are all small enough, JobMerge will merge them into one file on each mapper node. The transfer rate of data will be close to the extreme.

3.2.5. Tree Building

The above processing is just for one node. To build a whole KD-tree, the processing needs to be repeated recursively. A complete KD-tree Hadoop algorithm is shown:

FUNCTION BuildTreeDFS (Path pD)

1. $rootD.Path \leftarrow pD$, $JPA.persist(rootD)$, $Q \leftarrow rootD$;
2. **WHILE** Q is not empty **DO**
 - a. $nodeD \leftarrow Q.removeFirst()$, $JobMerge(nodeD)$;
 - b. **IF** $nodeD.cnt$ less than $iMemoryLimit$ **THEN**
 - 1) $JPA.begin()$, $nodeD.bucket \leftarrow 0$; $JPA.commit()$;
 - c. **ELSE**
 - 1) $JPA.begin()$, $nodeD.bucket \leftarrow 1$, $JPA.commit()$;
 - 2) $JPA.begin()$, $JobMedian(nodeD)$, $JPA.commit()$;
 - 3) $nodeLeft, nodeRight \leftarrow JobSplit(nodeD)$;
 - 4) $JPA.persist(nodeLeft)$, $JPA.persist(nodeRight)$;
 - 5) $Q.addLast(nodeLeft)$, $Q.addLast(nodeRight)$;
 - d. **ENDIF**
3. **ENDWHILE**

Where $JPA.persist()$ function means a new node is persisted by JPA; the assignments enclosed by $JPA.begin()$ and $JPA.commit()$ mean these operations are monitored by JPA; $JobMerge()$, $JobMedian()$, $JobSplit()$ are mapreduce jobs in Hadoop, which will be distributed to many nodes in a cluster.

4. Experimental Setup

4.1. Computing Environment

Experiments were conducted using Linux CentOS 5.0, Hadoop 0.19.1, JDK 1.6, a 1Gbps switch network, and 4 ASUS RS100-X5 servers (CPU: Dual Core 2GHz, Cache: 1024 KB, Memory: 3.4GB, NIC: 1Gbps, Hard Disk: 1TB).

4.2. Performance of Inverted Indexing

We evaluated several synthetic datasets and an open dataset. Two different-sized experiments are shown in Table 1. A (1,000,000 \times 200) synthetic data set was generated at random for test1. A (102,294 \times 117) data set was taken from KDDCUP 2008 [15] for test2. The procedure was followed according to the design (see 2.1.2). Of both tests, the latter takes fewer seconds for searching than the former. The SearcherTime is the average for a single query.

Table 1 Comparison of synthetic and open dataset

Item	Test1	Test2
FloatData(MB)	1,614	204
DiscretizationTime(s)	180	59
EncoderTime(s)	482	42
IndexerTime(s)	8473	397
SearcherTime(s)	1.07	0.68

4.3. Improved Intersection

In Figure 7, the longer candidate has $1E7$ sorted random integers; the x coordinate gives the proportions of the shorter candidate to $1E7$. The red line (normal) shows a constant higher time complexity between unbalanced arrays; the yellow line (improved) shows the time decrease dramatically along with the proportion of the shorter to the longer. The improved algorithm performs $O(M+\log N)$.

We also find the critical proportion is about 200 for the parameter `iTimesLength` in the improved intersection algorithm. Because binary searches arose many cache miss and sequential searches make use of cache hit [16] [17], the algorithm has no advantage in balanced intersections with proportion below 200.

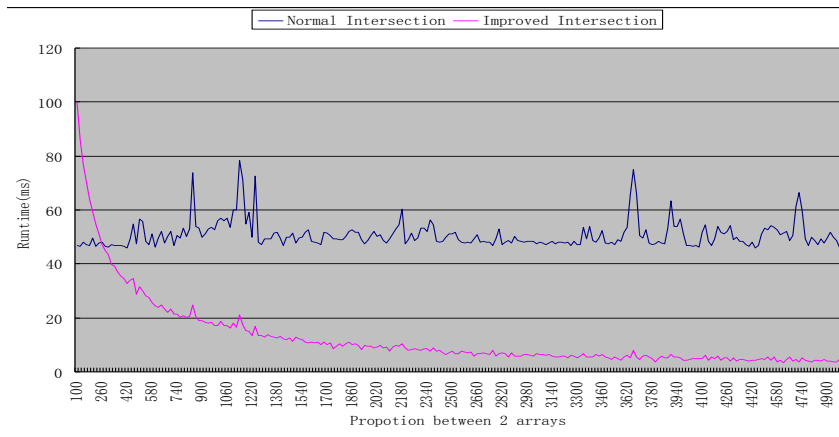


Figure 7. Runtime comparison between normal and improved Intersections. The X coordinate means the proportion between the two arrays, by which the length of one array will become shorter; while the other remains $1E7$ elements.

4.4. Data Indexing

The essential settings in the algorithm are listed in **Table 2**. The setting `dfs.replication` should not be 1, which will ignore the redundancy and the network transfer rate.

Table 2 Main setting. The item ^{a, b, c} are the setting in Hadoop, the item ^{d, e} is in algorithm.

Item	Transfer rate
<code>io.file.buffer.size</code> ^a	65536
<code>dfs.replication</code> ^b	2
<code>mapred.child.java.opts</code> ^c	-Xmx800m
<code>iMemoryLimit</code> ^d	1024*1024*1024
<code>iBucketLimit</code> ^e	1024*64

The data shown in **Table 3** are average from multiple results. The data will be different on the situation of failure of a Hadoop cluster. The size of bucket-1 nodes in the KD-tree is just smaller than `iMemoryLimit`, it ensures that the naïve KD-tree algorithm can split the bucket-1 nodes into the bucket-3 nodes (in-memory nodes). The “time

cost of bucket-1” denotes the cost of JobMedian, JobSplit, and JobMerge on the files and directories on HDFS. The “time cost of bucket-3” denotes the cost on memory.

Table 3 Comparison between datasets with different size in building KD-tree on Hadoop. All the data is from synthesis algorithm, with 117 fields, except KDDCUP.

Item	Time cost of bucket-1	Time cost of bucket-3
Build-KDDCUP	159 seconds	39 second
Build-1GB	313 seconds	51 second
Build-6GB	1963 seconds	243 seconds
Build-32GB	30010 seconds	2703 seconds

The transfer rates (Table 4) describe clearly the performance of data handling in the data mining framework on Hadoop is close to the extreme of the rack of 4 servers.

Table 4 Comparison of transfer rate. The item ^a, ^b has only read operation, the item ^c has R/W operation. The item ^d means file copy between two nodes using the linux command scp; the item ^e means the direct NIC transfer rate by UDP packet.

Item	Transfer rate
JobMerge ^a	59.7MB/sec
JobMedian ^b	64.3MB/sec
JobSplit ^c	24.6MB/sec
Net copy ^d	38.0MB/sec
UDP transfer ^e	120.0MB/sec

Finally, the engine type of MySQL is MYISAM or NDBcluster. The difference between them is trivial in the large-scale dataset, for time costs associate with “time cost of bucket-1” (**Table 3**). A strange error will occur with engine type of InnoDB.

The codes will be open at a cloud-based open source site, JavaForge, the SVN address is <http://svn.javaforge.com/svn/HadoopJPA>.

5. Conclusion

An efficient high-dimension large-scale data mining framework is proposed by the implementation of KD-tree algorithm on Hadoop; it employs the JPA and MySQL Cluster to resolve problems of globalization, random-write and duration in Hadoop. Experimental results show that its performances reach the peak of the transfer rate in our environment and it is technically feasible.

An improved intersection algorithm is proposed to enhance the naive data indexing approach in paper [11]. Experimental results show that the algorithm performs better. The improved intersection is generic for other situations.

We will consider how to implement more tree-based algorithms with JPA, and improve our inverted indexing in future work in an effort to enhance the performance in larger-scale data mining. A convenient tool for the data mining framework will be a focus of future efforts.

Acknowledgements

This work is supported by the National Basic Research Priorities Programme (No. 2007CB311004) and National Science Foundation of China (No.60775035,60903141,

60933004,60970088), National Science and Technology Support Plan (No.2006BAC08B06).

References

1. Bohm, C., et al. Multidimensional index structures in relational databases. in 1st International Conference on Data Warehousing and Knowledge Discovery (DaWak 99). 1999. Florence, Italy.
2. Dean, J., S. Ghemawat, and Usenix. MapReduce: Simplified data processing on large clusters. in 6th Symposium on Operating Systems Design and Implementation (OSDI 04). 2004. San Francisco, CA.
3. McCreddie, R.M.C., C. Macdonald, and I. Ounis, On Single-Pass Indexing with MapReduce. Proceedings 32nd Annual International Acm Sigir Conference on Research and Development in Information Retrieval, ed. M. Sanderson, et al. 2009, New York: Assoc Computing Machinery. 742-743.
4. Lammel, R., Google's MapReduce programming model - Revisited. Science of Computer Programming, 2008. 70(1): p. 1-30.
5. Moretti, C., et al., Scaling Up Classifiers to Cloud Computers, in IEEE International Conference on Data Mining. 2008, <http://icdm08.isti.cnr.it/Paper-Submissions/32/accepted-papers>: Pisa, Italy.
6. Gillick, D., A. Faria, and J. DeNero. MapReduce: Distributed Computing for Machine Learning. 2006; http://www.icsi.berkeley.edu/~arlo/publications/gillick_cs262a_proj.pdf].
7. Han, J. and M. Kamber, Data Mining: Concepts and Techniques, Second Edition. second ed. TheMorgan Kaufmann Series in DataManagement Systems, ed. M.R. Jim Gray. 2006, San Francisco: Morgan Kaufmann.
8. Berchtold, S., D.A. Keim, and H.P. Kriegel, The X-tree: An Index Structure for High-Dimensional Data. Readings in Multimedia Computing and Networking, 2001.
9. Bentley, J.L., Multidimensional binary search trees used for associative searching. Communications of the Acm, 1975. 18(9): p. 509-517.
10. Arya, S., et al. APPROXIMATE NEAREST NEIGHBOR QUERIES IN FIXED DIMENSIONS. in 4th Annual Acm-Siam Symp on Discrete Algorithms. 1993. Austin, Tx: Siam.
11. Yang, L. and Z. Shi, An Efficient Data Mining Framework on Hadoop using Java Persistence API, in The 10th IEEE International Conference on Computer and Information Technology (CIT-2010). 2010: Bradford, UK.
12. Biswas, R. and E. Ort. Java Persistence API - A Simpler Programming Model for Entity Persistence. 2009.
13. Hinz, S., et al. MySQL Cluster. 2009; Available from: <http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-overview.html>.
14. Bentley, J.L. K-d trees for semidynamic point sets. in Proceedings of the sixth annual symposium on Computational geometry. 1990: ACM.
15. USA, S.M.S. kddcup data. 2008; <http://www.kddcup2008.com/KDDsite/Data.htm>].
16. Lam, M.S., E.E. Rothberg, and M.E. Wolf. THE CACHE PERFORMANCE AND OPTIMIZATIONS OF BLOCKED ALGORITHMS. in 4th International Conf on Architectural Support for Programming Languages and Operating Systems. 1991. Santa Clara, Ca: Assoc Computing Machinery.
17. Przybylski, S.A., Cache and Memory Hierarchy Design: A Performance Directed Approach. 1990: Morgan Kaufmann.