XML Structural Similarity Search Using MapReduce

Peisen Yuan[1,2], Chaofeng Sha[1,2], Xiaoling Wang[3],
Bin Yang[1,2], Aoying Zhou[2,3], and Su Yang[1,2]

[1] School of Computer Science, Fudan University, P.R.C
[2] Shanghai Key Laboratory of Intelligent Information Processing, P.R.C
[3] Shanghai Key Laboratory of Trustworthy Computing, Software Engineering Institute,
East China Normal University, P.R.C
{peiseny,cfsha,byang,suyang}@fudan.edu.cn
{xlwang,ayzhou}@sei.ecnu.edu.cn

Abstract. XML is a de-facto standard for web data exchange and information representation. Efficient management of these large volumes of XML data brings challenges to conventional technique. To cope with large scale data, MapReduce computing framework as an efficient solution has attracted more and more attention in the database community recently. In this paper, an efficient and scalable framework is proposed for XML structural similarity search on large cluster with MapReduce. First, sub-structures of XML structure are extracted from large XML corpus located on a large cluster in parallel. Then Min-Hashing and locality sensitive hashing techniques are developed on the distributed and parallel computing framework for efficient structural similarity search processing. An empirical study on the cluster with real large datasets demonstrates the effectiveness and efficiency of our approach.

1 Introduction

XML has become a standard for data exchange in web application development. With the development of web and the wide usage of XML, search or query in the centralized solution cannot get satisfied result timely. To efficient manage these huge of XML data, distributed processing techniques for storing and managing the XML data in P2P or traditional distributed database techniques are developed [1,2].

Nowadays, cluster computing is broadly applied to the data explosion problem, which brings up an efficient solution by dividing the big data into small parts and processing in parallel on thousands of computers. The basic computing framework of cluster computing is MapReduce, which is a distributed programming model and software framework for rapidly writing applications that process vast amounts of data in parallel on thousands of cheap commodity computers. It has attracted much attention from many fields for processing and analyzing huge volumes of data.

Structural similarity search of XML data is an important problem in XML data management, which is related to data integration, XML classification and clustering, data cleaning etc. Tree edit distance is used for measuring the structure difference of tree-structured data, however, its complexity is rather expensive even for ordered tree [3]. Recently, Augsten et al.[4] proposes the *pq*-gram concept for tree structure comparing

L. Chen et al. (Eds.): WAIM 2010, LNCS 6184, pp. 169–181, 2010.
© Springer-Verlag Berlin Heidelberg 2010

for the ordered tree. However, these methods are running on single machine, which cannot scale well to large cluster for big corpus.

In this paper, we propose a *filter-and-refine* framework on large cluster with the Min-Hashing and locality sensitive hashing techniques for efficient structural similarity search on large XML corpus.

First, *pq*-gram proposed in [4] is adopted for extracting XML tree-grams, which is flexible for tree data structural similarity comparing. Then Hadoop is used as a platform for storing and processing large scale of XML data in parallel. Furthermore Min-Hashing and locality sensitive hashing techniques are implemented on the cluster for efficient searching. Extensive experiments indicate that our framework is efficient and effective for XML structural similarity searching on large cluster and scales well in term of the size of corpus.

To summarize, the main contributions of this paper are outlined as follows:

- We propose an efficient framework for structural similarity searching of XML data on cluster computing platform.
- Min-Hashing and locality sensitive hashing are implemented on cluster for efficient searching.
- Extensive experiments are conducted to demonstrate the effectiveness and efficiency of our approach.

The rest of paper is organized as follows. In Section 2, the preliminaries of problem definition, XML model and MapReduce are introduced. Min-Hashing and locality sensitive hashing are presented in Section 3. Architecture and main algorithms are introduced in Section 4, and experiment evaluations are presented in Section 5. The related work is surveyed in Section 6. Conclusion and future work are presented in Section 7.

2 Preliminaries

2.1 Problem Definition

The problem of structural similarity search is to retrieve the top-k most similar documents. Given an XML document corpus $\mathcal{D}$, a query document $\mathcal{Q}$ and a similarity function *sim*, it retrieves the the document set $\widetilde{D_k} = \{d_i | d_i \in \widetilde{D_k}, \widetilde{D_k} \subseteq \mathcal{D}, \forall d \in \mathcal{D} \backslash \widetilde{D_k}, sim(d_i, \mathcal{Q}) \geq sim(d, \mathcal{Q}), i = 1, \cdots, k\}$.

Jaccard similarity is a set based similarity, which is a simple and effective similarity measure and defined as $sim_{jacc} = \frac{|A \cap B|}{|A \cup B|}$, where A and B are two sets. The intuitive meaning of Jaccard similarity is that the more overlapping of A and B, the higher similarity they are. In this paper, Jaccard similarity is adopted as the similarity measure for the XML structural similarity.

2.2 XML Model and Tree Gram

In this paper, an XML document is modeled as a rooted ordered labeled tree. According to this model, element nodes and attributes are considered as *structural* or *internal* nodes, and the text value and attribute value nodes are treated as *text* nodes or $leaf$

node. Only *structural* nodes are taken into consideration as the structure of XML tree, and the $element\ tag$ and the $attribute$ node name are considered as the tree node $label$.

After modeling, the *pq*-gram proposed in [4] is adopted for extracting $tree\text{-}gram$ from XML tree, which is a flexible way to extract the structural information from tree structured data for approximate structural similarity comparing. The pq-gram is a sub-tree of the extended tree for two given integer parameters p and q.

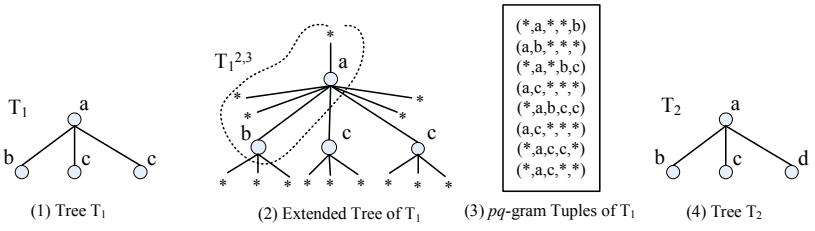


Fig. 1. Example Trees and pq-gram Tuples

Example of pq-gram of a tree T_1 is shown in Fig. 1. According to [4], for two integers p and q, an extended tree is first built as shown in Fig.1(2), the '$*$' nodes represent the extended empty nodes. Parameters of p and q are set to 2 and 3 respectively in the example tree T_1. The sub-tree enclosed by the dotted line is the first *2,3*-gram of T_1.

Let g be a pq-gram with the nodes $N(g)$= $\{n_1, \cdots, n_p, n_{p+1}, \cdots, n_{p+q}\}$, where n_i is the i-th node in pre-order traversal of g. The $(p+q)$-tuple $L^*(g) = < l(n_1), \cdots, l(n_p), l(n_{p+1}), \cdots, l(n_{p+q}) >$ is called *pq*-gram tuple of g, where $l(n_i)$ is the node label of the tree node n_i [5].

Example 1. The first *2,3*-gram of tree T_1 is the subtree circled by the dotted line in the Fig. 1(2). After pre-order traversing of the subtree, nodes '*', 'a', '*', '*' and 'b' are visited orderly. The *2,3*-gram tuple is denoted as (*,a,*,*,b).

An example of pq-gram tuples of T_1 is shown in the Fig. 1(3). The *pq*-gram tuples of XML Tree T are denoted as $\mathbf{G}_T$. The pq-gram tuple universal of the XML corpus is noted as $\mathbf{G}_\mathcal{U}$, $\mathbf{G}_\mathcal{U} = \bigcup_{i=1} \mathbf{G}_{Ti}$. The pq-gram tuple is also called $tree\text{-}gram$ or *pq*-gram if not confused in the following paper. Assume the universe of *pq*-gram tuples of the corpus are sorted lexicographically. Consequently, the structure of XML tree can be represented by a binary vector with dimension $|\mathbf{G}_\mathcal{U}|$.

Definition 1. *pq-Gram Binary Vector*

The binary vector $\boldsymbol{v}_T^{p,q} = (c_1, c_2, \cdots, c_{|\boldsymbol{G}_\mathcal{U}|})$ *is the pq-gram binary vector of the tree* T, $c_i = 1$ *iff* g_i *occurs in* $\boldsymbol{G}_T$, *otherwise* $c_i = 0$.

Example 2. In the example trees T_1 and T_2 of Fig. 1, vectors (1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0), (1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1) are *2,3*-gram binary vectors respective to T_1 and T_2.

2.3 Hadoop and MapReduce

Hadoop is the core of Apache Hadoop project [6]. It is an open-source implementation of frameworks for cluster computing, which consists of the Hadoop Distributed File

System (HDFS) and MapReduce. HDFS is a scalable and reliable data storage system for storing the file in the distributed file system, and MapReduce is a data processing framework for distributed parallel computing.

The MapReduce programming framework is designed for fast developing applications which run on large master-slaves shared-nothing cluster in parallel. It is the basic infrastructure for cluster computing for data-intensive problems. It includes two primitives: map and $reduce$ [7]. The map primitive processes $key/value$ pairs and generates intermediate $key/values$. The $reduce$ primitive merges all the intermediate $key/values$ of the map output results with the same key. Finally the results are output to the HDFS.

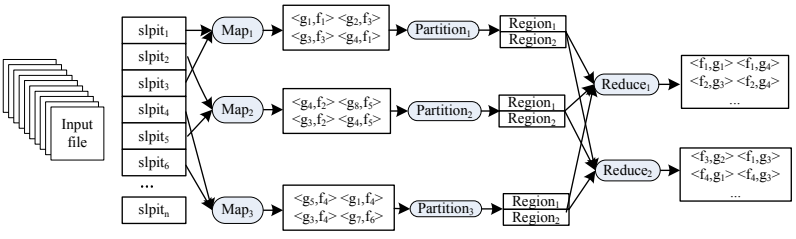


Fig. 2. MapReduce Processing Architecture

The processing architecture of MapReduce is outlined in Fig. 2. First, files to be processed are divided into equal splits. Each split is assigned to a map task. The $inputformat$ of the application parses the data in each split into $key/value$ pairs as the map's input. The intermediate key value lists are obtained and sorted on each node after map task. Then hash-based $partition$ procedure divides the intermediate key value list into R groups according to the key on each data node. Next, the $reduce$ task is scheduled for merging value with the same key. Finally, the results are output into HDFS.

The advantage of MapReduce is its simple programming interface, which can be run on shared-nothing cluster without worrying about the details of parallel distributed tasks, such as task coordinating, load balancing and data storage problems etc.

3 Min-Hashing and Locality Sensitive Hashing

The structural similarity measure between XML documents is evaluated by *tree-gram* tuple set and Jaccard similarity, which is defined as: $sim_{jacc}(\mathbf{G}_{T1}, \mathbf{G}_{T2}) = \frac{|\mathbf{G}_{T1} \cap \mathbf{G}_{T2}|}{|\mathbf{G}_{T1} \cup \mathbf{G}_{T2}|}$.

Min-Hashing. (MH) is introduced in [8] and widely used in duplicate detection [9] etc, which creates compact signatures for sparse binary vectors such that the similarity can be effectively measured by their signatures. Min-Hashing is an easy way to implement *min-wise independent permutation*[10], which has the property that the probability of

two sets have the same value of Min-Hashing is equal to their Jaccard similarity. The formal definition is given as follows.

Given a random hash function $h : x \rightarrow I$, where x is an integer, and $I \subset \mathbb{N}$, the Min-Hashing function is defined as $m_h(\boldsymbol{v}) = argmin\{h(\boldsymbol{v}[i]) \mid \boldsymbol{v} \in \boldsymbol{V}_b, \boldsymbol{V}_b$ is binary vector set, $\boldsymbol{v}[i]$ is the i-th component of $\boldsymbol{v}$ and $\boldsymbol{v}[i] = 1, 0 \leq i \leq |\boldsymbol{v}| - 1\}$. According to the property of Min-Hashing, given two XML documents d_1, d_2, the structure of which is represented by the *tree-gram* sets $\mathbf{G}_{T_1}$ and $\mathbf{G}_{T_2}$ and their $tree\text{-}gram$ binary vector are $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ respectively, their structural similarity can be obtained by Eq. 1.

$$\mathbf{Pr}[m_h(\boldsymbol{v}_1) = m_h(\boldsymbol{v}_2)] = sim_{jacc}(\mathbf{G}_{T_1}, \mathbf{G}_{T_2}) = \frac{|\mathbf{G}_{T_1} \cap \mathbf{G}_{T_2}|}{|\mathbf{G}_{T_1} \cup \mathbf{G}_{T_2}|}. \tag{1}$$

Min-Hashing is an approximate procedure for evaluating the set similarity. Thus a random hash family $\mathcal{H} : \boldsymbol{V}_b \rightarrow I$ is used for improving the efficiency of retrieval. Given a hash family $\mathcal{H}$, n Min-Hashing signatures are computed for each document. Thus the binary vector $d_{\boldsymbol{v}}$ of the document d is transformed into Eq. 2

$$g(d_{\boldsymbol{v}}) = \{m_{h_1}(d_{\boldsymbol{v}}), m_{h_2}(d_{\boldsymbol{v}}), \cdots, m_{h_n}(d_{\boldsymbol{v}})\}, m_{h_i} \in \mathcal{H}, i = 1, \cdots, n. \tag{2}$$

After generating n Min-Hashing value for all the documents, a signature matrix is obtained, which is a compact representation of the original binary document-gram vectors. Then pairwise similarity can be evaluated by the *signature* matrix. According to the property of Min-Hashing, Eq. 3 can be obtained.

$$\mathbf{Pr}[g(d_{\boldsymbol{v}_1}) = g(d_{\boldsymbol{v}_2})] = sim_{jacc}(d_1, d_2)^n. \tag{3}$$

Locality Sensitive Hashing. (LSH) is introduced in [11] and widely used in nearest neighbor search for high dimensional data, approximate KNN query etc, which focus on pairs of signature of the underlying similar sets. The scheme of locality sensitive hashing is a distribution on a hash function family $\mathcal{H}$, which operate on a collection of objects, such that for two objects o_1 and o_2, $\mathbf{Pr}_{h \in \mathcal{H}}[h(o_1) = h(o_2)] = sim(o_1, o_2)$ [12]. The basic idea of LSH is to hash the objects using $\mathcal{H}$, and ensures that for $\forall h \in \mathcal{H}$, the more similarity of the objects, the higher probability they are hashed to the same bucket. The LSH is a filter-and-refine framework for retrieval. For the retrieval precessing, the candidates of similar objects can be retrieved in the collision buckets, and the non-similarity objects are filtered out. The real similarity objects can be validated in the candidates set, without the need of sequential examining.

In order to improve the searching efficiency, the signature matrix is divided into b bands with r Min-Hashing signatures each band. Signatures in each band are hashed into B collision buckets, where B is a large number. Suppose the similarity of two objects is s, the probability that their LSH signatures agree in all r Min-Hashing signatures of in at least one of b bands is $1 - (1 - s^r)^b$, i.e. the probability that they can be retrieved is $1 - (1 - s^r)^b$.

In our system, r Min-Hashing values are concatenated for the hash key. The band number and the bucket number make up the key jointly of the MapReduce job. The similarity list in the bucket of each band is considered as the $value$.

4 System Architecture and Algorithms

4.1 System Architecture

The architecture of the system is shown in Fig. 3, which consists of three parts. The first part is the $parsing\ phase$. The structure of XML corpus stored in the HDFS are parsed and extracted into *tree-gram* tuples using the algorithm in [4] with one MapReduce job. The key of the result is file name and the $value$ is the *tree-gram* set. The $key/value$ pairs are output to HDFS by the $reduce$ task finally. The processing of parsing is shown in the Fig. 2, where g_i is the *tree-gram*, f_i is the file name.

The second part is the $vector\ building\ phase$. The $tree\text{-}gram$ universal $\mathbf{G}_{\mathcal{U}}$ of the corpus is obtained by the $Universal\ Generator$ module in one MapReduce job firstly. Then $\mathbf{G}_{\mathcal{U}}$ is distributed by $DistrbutedCache$ function of Hadoop to each data node. According to $\mathbf{G}_{\mathcal{U}}$ and the pq-gram tuple set of the XML documents, the binary document-gram vectors are built in one MapReduce job subsequently.

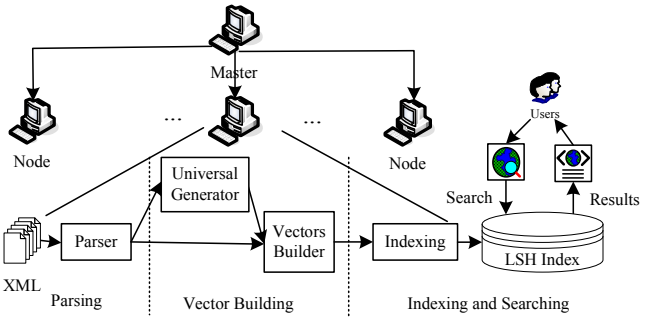


Fig. 3. Architecture of the System

The third part is the $indexing$ and $searching\ phase$. The index is built in one MapReduce job. Given a binary document-gram vector $\boldsymbol{v}$, r Min-Hashing values of each vector are computed for each band. Then these r Min-Hashing values are concatenated to make up the hash key jointly and the keys of all the vectors are hashed into B buckets of each band. The band number and the collision bucket number make up the key of the index, and the candidate file list is used as the $value$. The first three part are computed in parallel on each data node.

Given a query $\mathcal{Q}$, the collision bucket numbers of each band for the query are evaluated. Next the candidate set is obtained with the LSH index. Then the similarity distances of $\mathcal{Q}$ with candidate sets are computed. Finally the results are sorted according to their similarity and returned to the user.

After the introduction of the architecture of the system, main algorithms used in the system are presented in the following section.

4.2 Algorithms

Vector Building. The *tree-grams* are extracted from the XML documents residing on each data node in one MapReduce job using the algorithm from [4] in parallel. The *tree-*

Algorithm 1. map Function for Building Binary Vectors

Input: Key: *file-name*. Value: *tree-gram list*. *tree-gram universal* $\mathbf{G}_{\mathcal{U}}$.
Output: Key: *file-name*. Value: *doc-gram vector*.

```
1  Vector v = ∅;
2  G_U is distributed to each data node by DistributedCache;
3  foreach g ∈ G_T do
4      if g ∈ G_U then
5          v.add(1);
6      else
7          v.add(0);
8  output_key = file-name;
9  output_value = v;
10 Emit(output_key, output_value);
```

gram sets of the XML corpus are output into HDFS. After that, the *tree-gram* universal $\mathbf{G}_{\mathcal{U}}$ is generated in another MapReduce job.

The map function of the binary document-gram vector building is introduced in Algorithm 1. First $\mathbf{G}_{\mathcal{U}}$ is distributed to each data node by $DisributedCache$ (line 2). Then the binary vector is built for the XML document (line 3-7). Next the vector is emitted with the file name as the key and the vector as the $value$ in the map. Finally the $reduce$ algorithm outputs the binary vectors into HDFS. The $reduce$ algorithm is omitted here due to its simplicity.

LSH Indexing. LSH Indexing of map function is presented in Algorithm 2. For each vector $\boldsymbol{v}$, the $computeMinHash()$ function compute the r Min-Hashing values for each band. These r Min-Hashing values are concatenated to make up the hash key for each band(line 4). Then the band number and the collision bucket number make up the key of the MapReduce, and the file name as the $value$ are emitted to $reduce$ (line 7).

Algorithm 2. map Function for LSH Indexing

Input: Key: *file-name*. Value: *binary doc-gram vectors*.
Output: Key: *collision bucket number*. Value: *file list in the bucket.*

```
1  MinHashClass minHash = new MinHashClass();
2  fn = getFileName(file-name); /*get the file name*/;
3  docVector = getVector(fn); /*get the file vector by file name*/;
4  Band:BucketNo = minHash.computeMinHash(docVector);
5  output_key = Band:BucketNo; /*the band number and the bucket number are concatenated
   as the key*/
6  output_value = output_value ∪ file-name;
7  Emit(output_key, output_value);
```

Subsequently the $reduce$ task of the LSH indexing merges the file list with the same key together, i.e. the same band and the same bucket number.

Searching Algorithm. The map function for structural similarity search is shown in Algorithm 3. Given a query $\mathcal{Q}$, the $minHashSimilarityQuery()$ function searches

Algorithm 3. map Function for Structural Similarity Searching

Input: Query Q.
Output: Key: *file name pair*. Value: *similarity distance*.

```
1 dist-list = minHashSimilarityQuery(Q, k); /*call algorithm 4*/
2 foreach dist ∈ dist-list do
3     output_key = file-name;
4     output_value = dist;
5     Emit(output_key, output_value);
```

the structural similarity XML documents and computes their similarity distances (line 1). Finally the similarity distance list is emitted to the $reduce$ task (line 5).

The $minHashSimilarityQuery()$ is introduced in Algorithm 4. Given a query $\mathcal{Q}$, the $getVector()$ accesses the binary vector firstly (line 4). Subsequently the collision bucket numbers for each band is computed by $computeMinHash()$ for the query vector (line 5). Then for each collision number, the candidate file list is obtained (line 6-8). The Algorithm 4 can be divided into two phrases. The first phrase is the filtering phrase, which filters out the most of the non-similarity objects (line 2-8), and the second phrase is the refining phrase, which affirms the true valid similarity objects in the candidate set. The similarity distances of $\mathcal{Q}$ with each file in the candidate set are evaluated (line 9-12). Finally, the result list is sorted and returned (line 13-14).

Algorithm 4. Min-Hashing Structural Similarity Search on Hadoop

Input: *LSH index*, Query Q.
Output: *Top-k result list*.

```
1  Result = Φ;
2  C = Φ; /* list of candidate files */
3  MinHashClass minHash = new MinHashClass();
4  q'_v = getVector(Q);
5  queryCollisionBucket = minHash.computeMinHash(q'_v);
6  foreach collisionNo ∈ queryCollisionBucket do
7      c = getVectorByNo(bucketNo);
8      C = C ∪{c};
9  foreach f ∈ C do
10     CandidateVector CV = getVector(f);
11     simi = computeSetSimilarity(q'_v, CV);
12     Result = Result ∪{simi};
13 sort(Result);
14 return top-k result list;
```

5 Experiments

5.1 Experiment Setup

All the algorithms are implemented in Java SDK1.6 and run on Hadoop 0.19.1 and Hbase 0.19.1. The cluster is configured with 1 master node and 6 salve nodes. Each

node has a duo core intel 2.33GHz processor, 2G main memory and 180GB disk, which runs on Ubuntu 9.04, and with 1G memory allocated to JVM. The cluster is organized in the LAN with 100.0 Mbps. The XML documents are stored on HDFS. Hadoop is configured with the default and the replication of the data is set to 1. Each data node is configured with 2 maps and 1 reduce. The default parameters of b and r are set to 20 and 3 respectively.

For XML datasets, data from [13,14] are used. Big XML documents such as psd7003, SwissProt, dblp, nasa and treebank in the datasets are split with the XML Twig [15] to smaller ones with size of 100KB, 200KB, 500KB and 1MB. Finally, the size of the XML corpus is 3.4G and consists of 18015 XML documents. The parameters of p and q of the pq-gram are set to 2 and 3 respectively.

The datasets are divided into 4 groups with different size which randomly choose from the 18015 XML documents. The first group D_1 is 667 XML documents with the size about 100MB. The second one D_2 is 4863 XML documents with the size about 900MB. The third group D_3 is 9382 XML documents about 1.9 GB and the forth group D_4 is 18015 XML documents with 3.4 GB size.

5.2 Time Performance

In this section, the performance of the system is studied. We mainly studied (1) Parsing of the XML corpus to generate the pq-gram profile of the XML corpus, (2) Vector building for each XML corpus, (3) LSH index building. For the comparing searching performance, 2 searching schemes are studied: (1) Sequential scan from HDFS, which evaluates the similarity by sequential scan from HDFS in one MapReduce job, (2) Using LSH index.

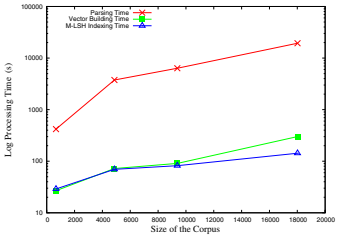


Fig. 4. Processing Time

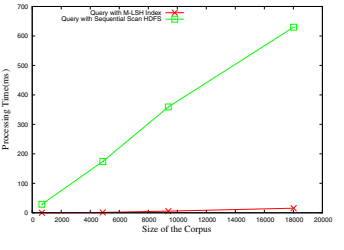


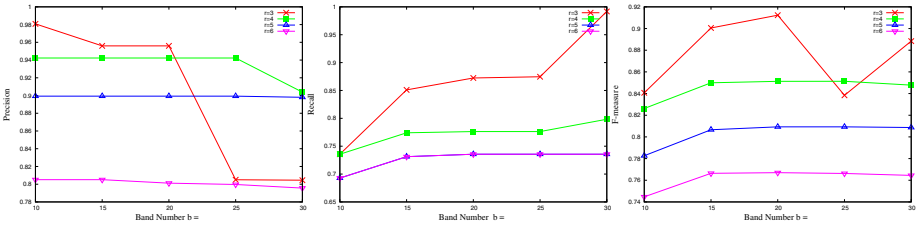
Fig. 5. Query Performance Comparison

Fig. 4 demonstrates the processing time. The time axis is logarithm time and the time unit is second. Fig. 4 indicates that (1) with the increasing of the corpus size, the parsing time increases almost linearly with the corpus size; (2) the XML parsing takes most of the time and LSH indexing takes about 150s for 18015 XML documents.

The query performance comparison is shown in Fig. 5. The time unit is millisecond. For query performance test, the number of bucket B of each band is configured to 7997. Fig. 5 indicates that the query performance with LSH index is far lower than sequential scan from HDFS, which takes time less than 0.2% of scanning from HDFS.

5.3 Precision

For precision experiment, the first group XML corpus dataset D_1 is used. The default bucket number B is set to 7997, and the *precision*, *recall* and *F-measure* are tested. The *F-measure* is defined as $2 \times \frac{precison \times recall}{precison + recall}$. 14 queries are issued in total and the average result is used as the final result.



(a) Precision (b) Recall (c) F-measure

Fig. 6. Precision, Recall and F-measure

Figures in Fig. 6 show the *precision*, *recall* and the *F-measure* with different parameters b and r, $r = 3, \cdots, 6$ and $b = 10, 15, 20, 25, 30$. Fig. 6(a) shows the *precision* is decreasing with the increasing of b and r. Fig. 6(b) indicates that the *recall* is increasing the the increasing of b, however, it decreases with the increasing of r. This figure proves the principle of the LSH that the more bucket, the more relevant are retrieved. Fig. 6(c) demonstrates that (1) the *F-measure* increases with the band number b before 15, it decreases after 25; (2) it increases with the r. When $r = 3$ and $b = 20$, the *F-measure* reaches peak. From Fig. 6(a) to 6(c), when $r = 3$ and b between 15 and 20, the measures of *precision*, *recall* and *F-measure* can get the best search quality. However, when $r = 4$, the search quality is stable with different b, but is much lower than $r = 3$. Thus, conclusions can be drawn that $r = 3$ and b between 15 and 20 is a good choice for Min-Hashing LSH.

5.4 Scalability

For scalability performance test, the search time are experimented with 4 group XML corpus and different number of similarity candidates.

Fig. 7 shows the search processing time with different buckets of per band and different size of corpus. The time scale of is logarithm time and unit is millisecond. From Fig. 7, conclusion can be drawn that (1) with the size of corpus increasing, the processing time increases, (2) with the number increasing of bucket of each band, the search time decreases. However, with the increasing of the buckets, the searching time decreases tiny after a sharp decreasing. Thus a tradeoff should be made between space cost and performance.

Fig. 8 demonstrates the searching time with different corpus size and candidate set, and the time unit is millisecond. 4 queries are issued for this experiment. $\mathcal{Q}_1$: OrdinaryIssue.xml, $\mathcal{Q}_2$: Proceedings.xml, $\mathcal{Q}_3$: customer.xml, $\mathcal{Q}_4$: reed.xml. There are 99,

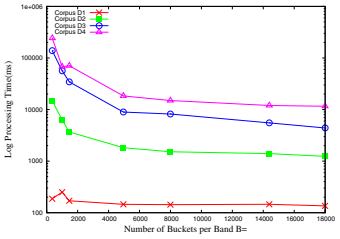


Fig. 7. Processing Time with Buckets

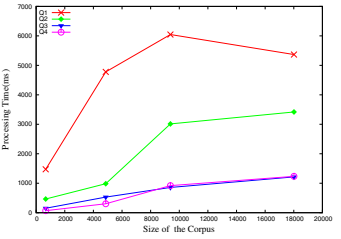


Fig. 8. Query Performance with different Candidates and Corpus Size

40, 7, 7 similarity documents respectively in each corpus. The bucket number B is set to 14389. Fig. 8 indicates that with the increasing of the size of corpus, the searching time is increasing. The reason is that the larger of the corpus size, the more candidates need to be probed. This experiment also shows that, the more similar documents in the corpus, the more time it costs for searching. However, the query time $\mathcal{Q}_1$ decreases in D_4 than D_3, the reason is that the data may be distributed unevenly in the cluster. Fig. 5 and Fig. 8 indicate that our system scales well with the size of the corpus.

6 Related Work

In this section, the literatures on approximate structural similarity computing of XML data, cluster computing and the application of locality sensitive hashing are reviewed.

Approximate tree structural comparison is extensively studied which can give satisfied result. Thus researchers have proposed several approximate methods for XML structural comparison. Joshi et al. [16] proposed bag of $xpath$ based structural similarity. Recently, *tree-gram* based methods are proposed to capture the structural similarity for ordered tree [4,17], which is becoming an efficient and flexible way for tree data similarity evaluation, They also give the bound against the tree edit distance. However, they do not take large scale data collection comparison into consideration.

In term of cluster computing, extensive researches have been conducted on large scale distributed storage and computing nowadays. Such as GFS [18], MapReduce [7], Dryad [19], Hbase [20] etc. These techniques have applied in many fields and have attracted lots of attention. The performance and efficiency of these system have been shown by empirical research for big data problem in these fields. The MapReduce as the basic computing framework provides easy operations for processing massive data on thousands of machine for distributed computing.

In the last few years, the locality sensitive hashing and Min-Hashing have been well studied and applied in many fields, such as duplicate detection, large rare association rule mining, clustering, information retrieval etc. LSH is widely used for efficient nearest neighbor searching in high dimension objects etc. Google news [21] employs the Min-Hashing and locality sensitive hashing for personal news distribution. Manku et al. [22] make use of Min-Hashing for detecting the duplicate web page in the web crawler.

7 Conclusion and Future Work

We propose a *filter-and-refine* framework for searching structural similarity XML documents on the cluster with MapReduce, which is efficient and effective with high search quality. Min-Hashing and locality sensitive hashing techniques are implemented with MapReduce in the framework. Our design gives an efficient solution for large scale XML documents management in the web age. Extensive experiments on real datasets show that our framework is efficient and scales well in term of the size of the corpus for structural similarity searching for XML data. In the web age, data update is very common, so how to manage these huge amount of data incrementally is rather important. In the future, optimization of searching algorithm and incremental update functionality will be added to the system. Furthermore, data mining and analyzing functions will be incorporated into the framework for large scale XML data management problems.

Acknowledgments. This work is supported by NSFC grants (No. 60773075, No.60925008 and No. 60903014), National Hi-Tech 863 program under grant 2009A A01Z149, 973 program (No. 2010CB328106), Shanghai International Cooperation Fund Project (Project No.09530708400) and Shanghai Leading Academic Discipline Project (No. B412).

References

1. Özsu, M.T.: Distributed XML Processing. In: Li, Q., Feng, L., Pei, J., Wang, S.X., Zhou, X., Zhu, Q.-M. (eds.) APWeb/WAIM 2009. LNCS, vol. 5446, p. 1. Springer, Heidelberg (2009)
2. Abiteboul, S., Benjelloun, O., Milo, T.: The Active XML project: an overview. The VLDB Journal 17(5), 1019–1040 (2008)
3. Jiang, T., Wang, L., Zhang, K.: Alignment of Trees-An Alternative to Tree Edit. In: Crochemore, M., Gusfield, D. (eds.) CPM 1994. LNCS, vol. 807, pp. 75–86. Springer, Heidelberg (1994)
4. Augsten, N., Böhlen, M., Gamper, J.: Approximate matching of hierarchical data using pq-grams. In: VLDB, pp. 301–312 (2005)
5. Yuan, P., Wang, X., Sha, C., Gao, M., Zhou, A.: Grams^3: An efficient framework for xml structural similarity search. In: DASFAA workshop: UDM (to appear, 2010)
6. Apache Hadoop Project (2009), `http://hadoop.apache.org/`
7. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI, pp. 1–13 (2004)
8. Broder, A.Z.: On the resemblance and containment of documents. In: Proceedings of Compression and Complexity of Sequences 1997, pp. 21–29 (1997)
9. Chum, O., Philbin, J., Isard, M., Zisserman, A.: Scalable near identical image and shot detection. In: ICVR, pp. 549–556. ACM, New York (2007)
10. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. JCSS 60(3), 630–659 (2000)
11. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: ASTC, pp. 604–613. ACM, New York (1998)
12. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: STOC, pp. 380–388. ACM, New York (2002)
13. XML Data Repository (2009), `http://www.cs.washington.edu/research/xmldatasets/`

14. Sigmod Record (2009), http://www.sigmod.org/publications/sigmod-record/xml-edition
15. XML Twig (2009), http://xmltwig.com/xmltwig/
16. Joshi, S., Agrawal, N., Krishnapuram, R., Negi, S.: A bag of paths model for measuring structural similarity in Web documents. In: SIGKDD, pp. 577–582. ACM, New York (2003)
17. Yang, R., Kalnis, P., Tung, A.K.H.: Similarity evaluation on tree-structured data. In: SIGMOD, pp. 754–765 (2005)
18. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS 37(5), 43 (2003)
19. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. In: SIGOPS/EuroSys, pp. 59–72. ACM, New York (2007)
20. Hbase Project (2009), http://hadoop.apache.org/hbase/
21. Das, A.S., Datar, M., Garg, A., Rajaram, S.: Google news personalization: scalable online collaborative filtering. In: WWW, pp. 271–280. ACM, New York (2007)
22. Manku, G.S., Jain, A., Das Sarma, A.: Detecting near-duplicates for web crawling. In: WWW, pp. 141–150. ACM, New York (2007)