International Journal of Database Theory and Application
Vol.8, No.1 (2015), pp.111-128
http://dx.doi.org/10.14257/ijdta.2015.8.1.13

Improving the Performance of Precise Query Processing on Large-scale Nested Data with UniHash Index

Li Wang[1], Dunlu Peng[1,*] and Ping Jiang[2]

[1]School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai, 200093, China
[2]Shanghai BroadText Iswind Software Co. Ltd, Shanghai, 200072, China
dlpeng@usst.edu.cn

Abstract

Querying nested data has become one of the most pivotal issues for seeking desired information on the Web. Unlike the traditional information retrieval, to effectively manage nested data, we generally need store the data and its structures separately, which significantly reduces the performance of data retrieving, especially when the dataset is in a large scale. More seriously, it brings a big challenge on ensuring the efficiency of processing precise queries that need to locate the exact positions of some certain values in a nested dataset. Combining the techniques of column-strip storage and inverted index, this paper defines an expression to represent the data objects' unique location in nested records— UPath, and based on which we present a new index structure— UniHash to support precise query processing on nested datasets. In addition, this work develops the related algorithms for building UPath, establishing UniHash, performing precise queries on UniHash with MapReduce platform and maintaining UniHash as well. Compared with some existing approaches, such as XPath-based and Dremel, UniHash index is capable of supporting the execution of precise queries over nested dataset with better performance. We give the results of a group of experiments, which were conducted on different real datasets, to demonstrate the efficiency of the approach.

Keywords: *nested data, MapReduce, precise query, column-storage, UniHash, data indexing*

1. Introduction

The rapid development of Internet, popularization of social networks, mobile communications and various applications have made data generated tremendously in recent years, undoubtedly, humanity has entered into a big data era. Most of the data used in programming languages, messages exchanged in distributed applications and publishing data on the Web by the major web companies is expressed in nested representation, such as XML, JSON, etc. According to the report from IDC, by the end of 2012, the proportion of nest data occupied 75% of the entire data [1]. The vast amount and complex types of nested data take companies lots of time and efforts to analyze in order to discover useful information. Seeking desired information efficiently from nested data becomes one of the hottest issues which have gained much attention from both academic and industrial areas.

Precise query on nested data at Web scale is usually rather costly. In contrast to the fuzzy queries, which contain fuzzy words or vague relationship [2], a precise query refers to a query that involves user-specified keywords or conditions, such as exactly querying one's position information on Google Maps, precisely extracting a series of signals on a Web page, quickly finding error signals among a large column of regular signals, etc. However, some factors, such as the explosive growth of column and storing structure separately from the content of nested data, bring great challenges to efficiently conducting

Copyright ⓒ 2015 SERSC

queries on such datasets, especially precise queries. In order to retrieve the desired information from massive nested datasets through a precise query in an efficient and accurate way, one crucial issue is to develop an effective structure, which indexes the massive nested data structure and content efficiently, to support queries of nested data with high performance.

XML (eXtensible Markup Language) is a popular representative structure for expressing nested data. As a standard format for data transmission and exchange on the Web [3], XML has been widely supported and used in many applications [4, 5]. Due to the flexibility of XML structure, it is hard to store its structure and content separately which makes record as the most common way of storing XML data [6]. However, the efficiency of querying data stored in records cannot be guaranteed because it needs to read a large number of irrelevant records on the disk repeatedly during the period of processing query. To solve this problem, literature [7] defines the repetition and definition levels to represent the structure of nested records, and uses columnar storage to store data. Columnar storage ensures the efficiency of aggregation queries between records or within records. Nevertheless, because of not being able to precisely locate the data, it cannot ensure the performance of precise queries. Currently, an effective method to pinpoint the position of a certain data object in a nested dataset is to use path expressions-XPath 2.0 [8]. Since W3C determined XPath as the recommended standard for querying XML data in 2007, it has been broadly used and implemented in many systems [9, 10].

XPath is a kind of structured query of nested data. Structured query uses a pattern tree to represent the query expressions, and then matches the pattern tree with the original records to get the query results [11-13]. One of the advantages of XPath is that it can express the query request accurately and precisely find the location of a certain part in the nested records. But when using one statement to query multiple records, or when the data model is very complex, XPath is not applicable in these situations. Another common used query is keyword-based query which is also known as full-text query. With full-text query, users can retrieve information even without knowing the structure of nested records [14, 15]. But due to the unclear expression of query purpose, full-text query cannot support precise query efficiently.

As an attractive paradigm for processing data in a distributed environment, MapReduce [16] provides a parallel solution on retrieving data with better performance than the traditional DBMS. Applications based on MapReduce implement the business logic independently with parallel coding and then are carried out over shared-nothing clusters composed of commodity machines that are connected via high-speed networks. The applications can concurrently cope with data in PB magnitude with a reliable fault-tolerant manner. Nonetheless, the literature introduces little research work on how to employ MapReduce to process nested data, particularly precise query processing over nested data. In this work, we try to deal with query on massive nested datasets with MapReduce.

Based on the prior discussion, this work focuses on improving the efficiency of precise query processing over nested data using MapReduce platform. By taking advantages of XPath and full-text query, we propose a hash table based structure called UniHash to index a value in the nested datasets uniquely. UniHash supports the precise queries of nested data very well. When querying, after finding the value’s path in UniHash table, the system matches the pattern with the original record to locate the value’s position in the original records. Additionally, during the query processing, multi-core parallel technology is employed to handle the concurrence of query requests. With these measures, the overall performance of query processing has been improved. The results of experiments demonstrate the efficiency and effectiveness of the proposed approach.

The rest of this paper is organized as follows: Section 2 introduces the related work to our study. Section 3 describes fundamental knowledge for establishing UniHash index. The algorithms for generating UPath and building UniHash index are proposed in Section

4. Section 5 describes how to maintain UniHash when the original dataset changes. The effectiveness of UniHash index is verified by experiments in Section 7. Finally, we draw the conclusion in Section 8.

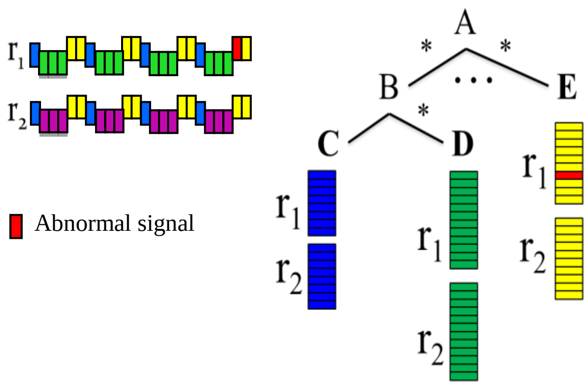


Figure 1. Nested Records and their Columnar Storage

2. Related Work

As one of the the main forms for expressing unstructured data on the Web, nested data has been widely used and took up the vast majority of entire data in the big data era. In such a scenario, performing queries efficiently over nested data becomes increasingly important and pressing in many applications. As we aforementioned, the storage of nested data is relative complicated because the structure and data itself are often separated and need to be stored respectively. Generally, from the perspective storage, there are two aspects affecting the efficiency of querying nested data most: storage mode and indexing structure.

In [7], a system called Dremel was proposed to store nested data in columnar storage which supports aggregate queries over trillion-row tables in seconds. However, for precise queries, Dremel cannot perform the same good because it is not capable of locating a value accurately in the dataset. Figure 1 is derived for the example records used in [7] by adding an abnormal signal labeled read into record r1. Suppose an engineer wants to correct the abnormal signal, he first needs to find exactly its position. Due to the location and structure of the abnormal signal are exactly the same as that of some other normal signals in r1, in the worst case, in order to locate the abnormal signal, Dremel has to traverse all the signals that can be accessed through the path. It is obvious that most of the accessed signals are not in the query results at all. Hence, it is very time-consuming for running precise queries over a vast amount of nested records through Dremel.

As known to us, a proper index structure can largely improve the efficiency of data query processing. In literature [12], the authors employed covering index for branching path of nested records. Through the index, the performance of branching path query is significantly improved. V. Hristidis *et al.*, addressed the problem of identifying the most specific context elements (*i.e.*, LCAs) that contain all the keywords, along with a compact description of their witness (*i.e.*, GDMCTs) which can efficiently execute keyword proximity queries on labeled trees. Literature [15] gives the suggestion for the typographical errors in keyword query. It is stated that the suggestion could improve users' search experience by avoiding returning empty or poor quality results. But when we try to utilize the structured indexing or keyword indexing over nested data, there will arise many issues such as excessively performing disk I/Os and low efficiency for precise queries because it could not locate the data accurately.

The technology of full-text retrieval is widely used in search of massive documents, and it enables users to get correct results in relative short period of time. For this reason, many researchers have applied inverted index, which is a widely exploited indexing structure for full-text retrieval, in nested data queries in recent years [17, 18]. The index records established for XML documents in [17] include some position information for the data, such as the serial number and the path expression of a node. The indexes are stored in RDBMS, and the queries are also executed on the RDBMS. D. Florescu and his group extended the inverted index to 2-INDEX that supports two kinds of indexes: T-index for indexing text words and E-index for indexing the elements [18]. T-index is formed of the numbers of words, the words' location and their nested depth in the document. E-index is constituted by the document numbers of the elements, their starting and ending positions and nested depth in the document. They support queries of the relationship between contents and elements and that between elements and attributes efficiently. However, they cannot guarantee the efficiency of processing the precise queries executed on them.

As an efficient solution for processing large-scale of dataset, MapReduce showcases many advantages over traditional databases, i.e., good fault tolerance, high availability of nodes, high operability of heterogeneous environments, etc. Many applications and systems for processing big data are implemented based on MapReduce, such as Apache Hadoop [19], MicroSoft Dryad [20], Facebook Hive [21], etc. Therefore, researchers have already paid some attention to process nested data with MapReduce. Currently, some related achievements have been reported in the literature, such as Xadoop1, ChuQL [22] and MRQL [23]. On the basis of Hadoop, Xadoop uses XQuery syntax to write MapReduce procedures and executes the queries with existing XQuery engines. The obvious drawback of this mode is that it takes record storage as the unit for partitioning data, which causes low query efficiency. ChuQL implements XML query with a hybrid approach which specifies XQuery statements being executed in Map or Reduce phase, however, it is hard to optimize the query mode. MRQL provides a SQLlike language to conduct operations over XML data. It performs XQuery with a XML query compiler which is rebuilt on MapReduce. But it is time-consuming to re-implement the XQuery engine, even it offers good extensibility.

Generally, there have multiple values along a common path in nested datasets. Unfortunately it is unable to locate accurately a certain value with the prior discussed structures, especially when the dataset is in vast amount. In this work, we utilize ideas from the technologies of full-text indexing and XPath to build the UniHash index for the data in nested records. UniHash not only includes the number of a document (DocId), but also the Unique Path (or UPath) of the data which marks a certain value uniquely. With UPath, a value can be located precisely in the dataset. For the sake of efficiency, we use the computing model of MapReduce to establish the UniHash index and store the entries with columnar storage. It is proved that columnar storage is more suitable for data retrieving in MapReduce environment than record storage that is universally employed in tradition databases. Through these measures, the query requests can be responded in a fraction of time.

3. Preliminary

In this section, we mention the fundamental techniques employed in our work, including path expression, MapReduce, full-text retrieval and the data model.

3.1. Path Expressions

Currently, one of the most commonly used query languages for nested data is XPath which uses regular path expressions to accurately locate the nodes in the dataset. By

[1] http://www.xadoop.org/

taking advantage of its tree structure, nested data can be expressed as a tree. The elements are considered the labeled nodes of the tree, while the values are described as the tree leaves. The definition of path expressions is formally defined as follows.

Definition 3.1: A path expression for a node is composed by a sequence of labels that are labels of the nodes from the root to the given node in the tree. The intermediate nodes are selected through the paths or steps, and the adjacent nodes are separated with "/" or ".".

For conciseness, we use 'path' to stand for 'path expression' for a given node in what follows. For example, the path of the yellow signals in Figure 1 is A/E or A.E. In the figure, there are multiple values located on each leaf, which means even a leaf node is located successfully through its path, in order to get the exact value we want, the system has to access and filter a lot of irrelevant values, which is indeed a very inefficient operation. To solve this problem, this paper proposes a structure to uniquely describe the location of each value and implement the structure on the platform of MapReduce.

3.2. Full-text Retrieval

Among the most popular text indexing models, such as the inverted index, bitmap, suffix tree, signature files, *etc.*, the inverted index overcomes other models with better comprehensive performance and is widely used in the applications of full-text retrieval [24].

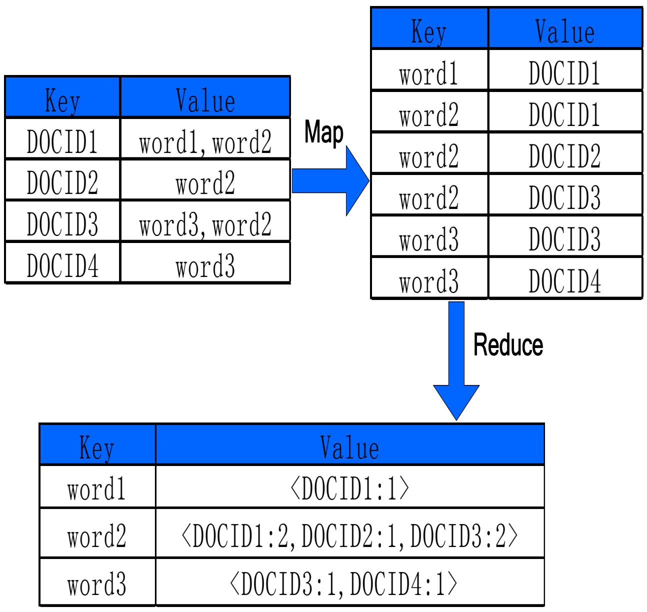
It is convenient to adopt inverted index to answer queries of records on the basis of their attributes. In search engines, an inverted index is often composed of the terms and their posting lists that make it fit for fast retrieving of certain words. The primitive form of the inverted index is as follows:

(term, posting<DocId , payload>)

where term is the searching keyword, posting denotes the term's occurring information containing two parts: document ID DocId and information about occurrences of the term payload. During the process of retrieving, search engines traverse the inverted list to get the information of the related documents that contain the searching keywords and then return the documents to users.

When document sets is massive (*e.g.*, at Web scale), full-text retrieval is extremely time-consuming. Reportedly, as a programming model which can parallel process large-scale datasets, MapReduce improves the efficiency of full text retrieval to a great degree. In the applications of text retrieval, the Map phase is often used to accomplish word segmentation datasets, MapReduce improves the efficiency of full text retrieval to a great degree. In the applications of text retrieval, the Map phase is often used to accomplish word segmentation and statistics, and the Reduce phase is to classify and merge the results produced in Map stage, and finally form the inverted index. The process of building an inverted index is illustrated in Figure 2.

On the foundation of inverted index, this work will establish UniHash index for the nested data based on MapReduce framework. UniHash is benefit for precisely retrieving data value in the nested dataset through keywords. Before discussing how to build UniHash with MapReduce in detail, we propose the data model that is employed to describe the nested data in our work.



Key	Value
DOCID1	word1, word2
DOCID2	word2
DOCID3	word3, word2
DOCID4	word3

Key	Value
word1	DOCID1
word2	DOCID1
word2	DOCID2
word2	DOCID3
word3	DOCID3
word3	DOCID4

Key	Value
word1	⟨DOCID1:1⟩
word2	⟨DOCID1:2, DOCID2:1, DOCID3:2⟩
word3	⟨DOCID3:1, DOCID4:1⟩

Figure 2. Process of Building Inverted Index with MapReduce

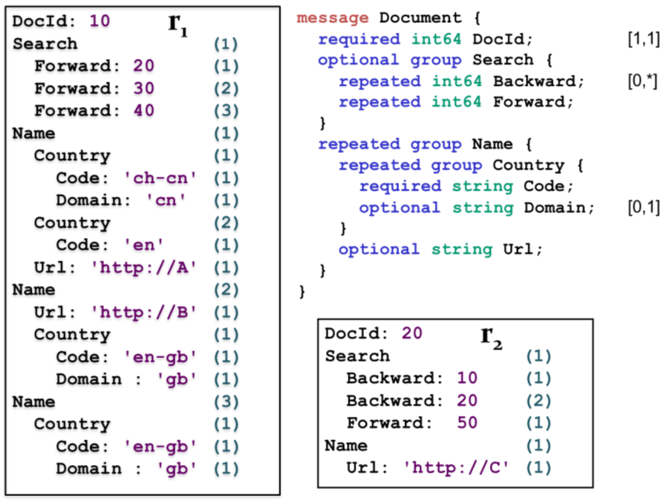
3.3. Date Model

A data model is a structure that uses some regular labels to represent nested records. For example, the data model in the open source distributed systems Protocol Buffers[2] is described with following syntax:

$$\tau = \mathrm{atom}|\langle A_1{:}\tau[*|?],\ldots,A_n{:}\tau[*|?]\rangle$$

Where τ is an atomic type or a record type. An atomic type denoted as atom in the expression comprises integers, strings, *etc.*, and $\langle A_1{:}\tau[*|?],\ldots,A_n{:}\tau[*|?]\rangle$ represents the record type which contains at least one field. Each field has a name Ai and an optional label "∗" or "?" for its multiplicity, where "∗" indicates the field must appear at least once, "?" denotes the field is optional and can appear any times. If a field's multiplicity is explicitly specified, the occurrence should conform to it. For instance, the fields labeled as required [1,1] in the upper right of Figure 3 are fields that must appear exactly once.

Let us take the data model shown in Figure 3 as an example. Field **DocId** is the



```
DocId: 10              r1
Search                 (1)
  Forward: 20          (1)
  Forward: 30          (2)
  Forward: 40          (3)
Name                   (1)
  Country              (1)
    Code: 'ch-cn'      (1)
    Domain: 'cn'       (1)
  Country              (2)
    Code: 'en'         (1)
  Url: 'http://A'      (1)
Name                   (2)
  Url: 'http://B'      (1)
  Country              (1)
    Code: 'en-gb'      (1)
    Domain : 'gb'      (1)
Name                   (3)
  Country              (1)
    Code: 'en-gb'      (1)
    Domain : 'gb'      (1)
```

```
message Document {
  required int64 DocId;              [1,1]
  optional group Search {
    repeated int64 Backward;         [0,*]
    repeated int64 Forward;
  }
  repeated group Name {
    repeated group Country {
      required string Code;
      optional string Domain;        [0,1]
    }
    optional string Url;
  }
}
```

```
DocId: 20          r2
Search             (1)
  Backward: 10     (1)
  Backward: 20     (2)
  Forward:  50     (1)
Name               (1)
  Url: 'http://C'  (1)
```

Figure 3. Two Sample Nested Records and their Data Model

[2] http://code.google.com/p/protobuf/

Copyright ⓒ 2015 SERSC

document's ID number and is required to appear exactly once, and **Name** is a repeated field which is nested by another repeated field **Country**. **Country** is composed of two sub-fields, **Code** and (optional) **Domain**. The relationship among these fields can be expressed with a tree model in which the full path of a nested field is expressed with a path expression, *e.g.*, **Name.Language.Code** for field **Code**.

It is clear Figure 3 describes nested dataset in a proper way. As we can see, this model brings benefits for storing and processing the data with columnar storage. Hence, in this paper, we use this model to express the nested records. Next, we will focus on the establishment of UniHash index and the implementation of its columnar storage.

4. Establishment of UniHash Index

The purpose of this paper is to build an effective index structure supporting precise queries for nested records. UniHash, the structure we propose in this paper, saves the position of the values' in the records, and specially it stores the values can be accessed through same paths on the contiguous blocks to improve the query efficiency. In what follows, we first give the definition of UPath and the approach to generate it, and then discuss how to establish UniHash index with MapReduce and build UniHash index tables to store the content and structure of nested records with columnar storage.

4.1. UPath and its Generation

The tree structure can be used to describe the relationships among data objects in nested documents. To get the values, it needs to traverse the trees by starting at the root node along different levels till reaching the leaf node. Probably, there are multiple values can be accessed through a same path. Moreover, some of values themselves are the same. For example, in Figure 3, in Path **Name.Country.Domain** we can get three values: one "cn" and two "gb".

In some applications, it is necessary to quickly locate a certain value (*e.g.*, "cn") or distinguish the same values (*e.g.*, the two "gb"). We call these query demands as precise queries. On the basis of our prior discussion, the path expressions proposed by previous work cannot answer the precise queries in an efficient way, because they do not differentiate the values (especially the same values) stored in the same path. To address this issue, we propose a new path expression in this work and name it as Unique Path (or UPath). Besides the data contained in the corresponding XPath expression, a UPath also records the number of occurrence for the repeated nodes. With this extension, UPath can be used to locate a value uniquely in the nested records, namely, to answer the precise queries. Before discussing how to fulfil the query with UPath, we give the definitions for the times of repetition for a given node and that of UPath.

Table 1. Terms and their Meanings used in Description of Algorithms

Name	*Meaning*
Element object	*Attributes:{path, index, value, UPath}*
Posting<DocId,UPath>	*UniHash record of a node*
Postinglist	*a list of postings with same term*
UniHash{key,posting}	*an entry of UniHash index, where key is the value, posting is the key's Posting*
UniHashList{path,UniHash set,use}	*a list of UniHash, UniHash set is the set of UniHash in path path, use is a boolean value for identifying the status of the UniHahList(true for valid,false for invalid)*

Definition 4.1: The times of repetition for a node refer the duplicate occurrences of the node in the same path.

For example, in Figure 3, two paths **Name.Country.Domain** whose values are “gb”, the times of repetition for the repeated field “**Name**” is 2 and 3 respectively, and for the value “en” in path **Name.Country.Code**, the times of repetition for repeated field “**Country**” is 2.

Definition 4.2: The UPath for a given value is a path expression that is formed by the connection of the repetition times of each node along the path to accessing the value. Normally, the times are separated by “.”.

For record r1 and r2 in Figure 3, the parentheses in the right of each row represent the current repetition times of each node. For example, even the paths of the two “gb” in r1 are both **Name.Country.Domain**, their UPaths are 2.1.1 and 3.1.1 separately. The former indicates that the “gb” value is in the path formed with the second **Name** node, the first **Country** branch and the first **Domain** node. The latter demonstrates that to get that “gb”, we need to traverse the nodes of the third **Name**, the first **Country** and the first **Domain**. Similarly, for the value of “en”, its path is **Name.Country.Code** and its UPath is 1.2.1.

Clearly, it is an effective way of using UPath to identify a certain value even there are some nodes with same value. However, computing all the UPaths for nested documents might be time-consuming, especially when the dataset is very large. As reported in the literature, there are lots of advantages of MapReduce on processing massive datasets. Therefore, in this paper, we look to establish UniHash index by using MapReduce. The terms, as well as their meanings, are listed in Table 1 to describe the algorithms developed this works.

Algorithm 1 presents the description of computing UPath. It contains three steps: the first step is to read the records stored in the common file path in batches, the second step

```
Algorithm 1: Generating information for records
Input: the path of the data file filePath
Output: Information of all element objects in the data file
Element loadData (string filePath) {  // create data model
    objs.index ← 1; // the level of root is 1
    objs.path ← DocId ; // root name named as DocId
    objs.value← 10 ; // the first Docid is 10
    objs.UPath← 1 ; // record the Unique Path
    foreach node n in the root's child nodes of the document filePath do
        getUPath (n,objs);  // fill the repeated time of each node in the record
    end-for
    return objs;
 }
```

Figure 4. Generating Information of Records

```
Algorithm 2: Function getUPath
Input: node obj and its parent
Output: obj’s UPath
void getUPath (Element node,Element parent) {
    obj.index++ ; // record repeated time of this node
    obj.UPath←parent.UPath+obj.index; // get the UPath
    obj.path ← parent.path+obj.name; // get Path
    if obj has child nodes then foreach node n in obj's childnodes do
            getUPath (n,obj) ; // producing the child nodes UPath
       end-for
    end-if
}
```

Figure 5. Algorithm of Computing UPath

is to create the data model which includes the path of the current node, the times of repetition for the nodes at each level, the values of the current leaf node and the UPath, and the third step is to extract information from the nested records to form the UPaths by calling function getUPath iteratively. The main codes of getUPath are shown in Figure 5. getUPath is the core algorithm for producing the UPaths. It recursively calculates the times of repetition for each node and produces the UPaths and paths from the root to the leaf nodes. After all the UPaths are generated, they are passed to the Map function to build UniHash index for each leaf node.

4.2. UniHash and its Establishment

Linklist is a usually used data structure in which each node stores a pointer to the next one. Specially, hash linklist stores the records with a same key in a common linear linklist. This approach not only supports dynamically allocate memory, but also accelerates the query processing. In this work, we employ hash linklist as the basic structure for building the index for nested datasets. The index is called UniHash, each of whose entry can position a unique value in the dataset.

Definition 4.3. An UniHash index is a hash linklist whose keys are the values of nested records and values are the postings with the same term. The values for a common key are listed as a postinglist and stored in a same linear list. Formally, a UniHash index entry is expressed as U<key, postinglist>. Correspondingly, a UniHash table consists of multiple UniHash index entries.

Now let us illustrate the UniHash index structure with examples. Figure 6 displays the

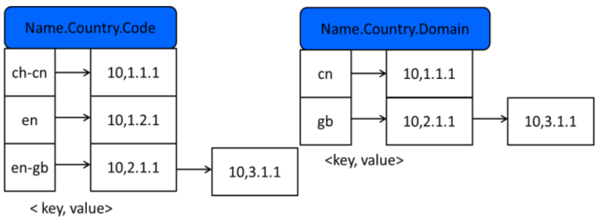


Figure 6. An Example of UniHash Table

UniHash tables for the nested data shown in Figure 3. The keys of the left UniHash table are the values of the nest data that can be accessed through path **Name.Country.Code**, and its postinglist contains the postings corresponding to the values of the nested data; similarly, the right UniHash index takes the values that can be accessed by path **Name.Country.Domain** as its keys and the corresponding postings as its values.

Evidently, the structure of UniHash index is very suitable for answering precise queries. For example, <10,3.1.1>, which is the posting of the UniHash table whose key is “en-gb”, guides us clearly to precisely locate the value of “en-gb” through accessing the third **Name**, the first **Country** and the first **Domain**. Another example is answer aggregative queries, such as to count the number of the elements satisfying a certain filtering condition. For instance, to get the number of elements whose **Domain** is “gb” in path **Name.Country.Domain**, we only need to calculate the length of the hash linklist in the UniHash table whose key is “gb”. These examples show that using UniHash index we can directly find the exact location of a certain value with the UPath in the original nested document rather than traversing the entire document.

Generating UniHash index is very costly for large-scale nested data because it needs to access every node in the nested documents for computing the node’s order. In order to improve the efficiency of generating UniHash index for large nested datasets, we implement the computation on the basis of MapReduce computing model. In the Map stage, a record is decomposed into the key-value pairs as <key, value>, where key is set of keywords and value is the keyword posting. A posting is a binary set and expressed as posting(DocId,UPath), in which DocId is the document number and UPath refers the keywords' UPaths in the document. Thus, the basic expression of the key-value pair is as follows:

Figure 7(Algorithm 3) depicts the Map function to create a UniHash index for each value on the leaf nodes of a nested document. The algorithm stores the values with the same path in the same UniHash table which makes it convenient to apply the columnar

$$\langle \text{term, Posting (DocId,UPath)} \rangle$$

storage. Two kinds of UniHash indexes are generated during the computation. The first

```
Algorithm 3: the Map function
Input: Element obj
Output: Set of UniHash tables defined as UniHashList
UniHashList Map(Element obj) {
          n ← obj.value ; // record the id of each record
   if obj is a leaf node then
      UniHash (obj.value, posting(n, obj.UPath));
            // create an UniHash index entry
       if obj.Path is equal to one of UniHashList.paths then
         UniHashList.add (UniHashList.path, UniHash);
              // add this UniHash entry to UniHash set
       else
         new UniHashList uhl; // Create a new UniHash set object
         uhl.path=obj.path;
         uhl.add (uhl.path, UniHash); // add leaf nodes to this new sets
       end-if
   else
      foreach node n in obj's child nodes do
         UniHashList.add (Map(Element n)) ;
         // repeat execute until leaf node
      end-for
   end-if
   return UniHashList;
}
```

Figure 7. Map for Constructing UniHash Index

kind is for the values, that is, the values form the keys and the document id's combined with its UPath construct the value of the index, *i.e.*, a postinglist. The second kind takes the path as the key and the set of UniHash index indexing the same path as the value. The output of the algorithm is a set of UniHash tables in the form of UniHashList which is passed to the Reduce phase as the input.

4.3. Generation of UniHashLists

The UniHash tables computed with the Map function contain the entries with same keys. Thus, it is necessary to merge the entries before establishing the final index. Figure 8 (Algorithm 4) describes the Reduce function.

We give a short description about the main idea of the function. After receiving the UniHashList passed from Algorithm 3, Algorithm 4 traverses the UniHashList in order to merge the UniHash index entries having the same keys. Specifically, it appends the postings of all the entries with a same key into that of a single one. This job is accomplished by function Add. In this way, the algorithm guarantees all the entries in the merged UniHashList have a unique key. The returned UniHashLists are stored in HDFS, and each UniHashList stores in one block in the columnar form according to its unique path.

```
Algorithm 4: Reduce function for merging the UniHash entries with the same key
Input: the original UniHashList A
Output: the UniHashList B by merging the UniHash entries with the same keys in A
 UniHashList Reduce(UniHashList A){
    UniHashList B;
    foreach entry ka of UniHash a in A do
       foreach entry kb of UniHash b in A do
          if ka.key==kb.key then
             ka.posting.Add (kb.posting); // append b's Posting to a's Posting
          end-if
          B.Add (a.path,ka) ; // append a to the set of UniHash
       end-for
    end-for
    return B;
}
```

Figure 8. Algorithm of Reduce to Build UniHashList

5. Maintenance of UniHash Index

In some circumstance, such as on the Web, usually the dataset varies dramatically, thus, there must be an efficient way of maintaining UniHash index as the dataset changes. In essence, the main task for maintaining UniHash index is to modify index key-value pairs correspondingly to the changes of dataset. There are three situations need to be processed: adding (or deleting) nested documents into (or from) the dataset, adding (or deleting) nodes into (or from) the data models and adding (or deleting) the values to (or from) the nested documents. As traditional data processing does, modifying nodes in data models or documents is regarded as first deleting and then adding, hence, we do not discuss this kind of operations in detail in this work.

For the increment of the documents, it is intuitive to employ Algorithm 1-3 to establish the UniHashLists for the new documents. Then, use the Reduce algorithm described in Algorithm 4 to merge with the new produced UniHash tables with those stored in the HDFS, and finally store them back to the disk. As we know, UniHash index supports precise queries well, so finding the entries with the same keys as the new produced UnHash entries is very efficient through the index. When adding a new node into the data model, the UPaths contain the being added node need be changed as well. It is necessary to rebuild the UniHash index as described above. For the addition of a new value into the documents, first it needs to compute its posting using Algorithm 1-3, and then searches the index stored in the HDFS to see if any entry has the same key as the new value. If there is, the posting of the new value will be merged with the existing postinglist and save it back on the disk. Otherwise, a new entry will be created for the values. If a deletion happened on a value of a document, the system only needs to search the UniHash index to find the entry whose key is the same as the deleted value, and then remove the corresponding posting from the entry's postinglist. If the postinglist becomes empty after the remove, delete the entire entry from the index. When deleting a node from the data model with large-scale nested datasets, changes may happen on the paths of many data objects as well as their UniHash. For nested records, the values should be removed before the removing of the node if the values' paths contain the removed node. Then, find and remove the UniHash entries whose paths contain the deleted node.

Compared to the above deletion operations, deleting nested documents from the dataset is more complicated. When a deletion happened on a a nested document, the UniHash tables will be firstly created from the document before the deletion. Then, search the UnHash index to find the entries whose keys are the same as the those of the UniHash tables, and delete the postings from the postinglist of the found entries. After deleting the postings, if the entries' posting is empty, remove the entire entry from the UniHash index.

Otherwise, delete the related postings with the same DocId. Figure 10 demonstrates when a document (DocId=20) is removed from the dataset, the change of the UniHash index for the records shown in Figure 3. The computation of modifying UniHash index for the deletion of a document is presented in Algorithm 5 (Figure 9).

```
Algorithm 5: Modifying the UniHash index for the deletion of a nested document
Input: the UniHashList A for the original UniHash Index, the UniHashList B for the deleted
document
Output: the new UniHashList A after deleting B
UniHashList deleteDocument(UniHashList A, UniHashList B){
    if B==null then
            return;
    end if
    if A.path==B.path then
       for all UniHash a in A, UniHash b in B do
          if b.key ==a.key and a.posting.DoicId==b.posting.DocId
          then
             if a.postingList.length ==1 then
                     A.remove (a);
             end-if
                  a.postingList.remove (b.posting);
          end-if
       end-for
    end-if
    return A;
}
```

Figure 9. Algorithm of Deleting Documents

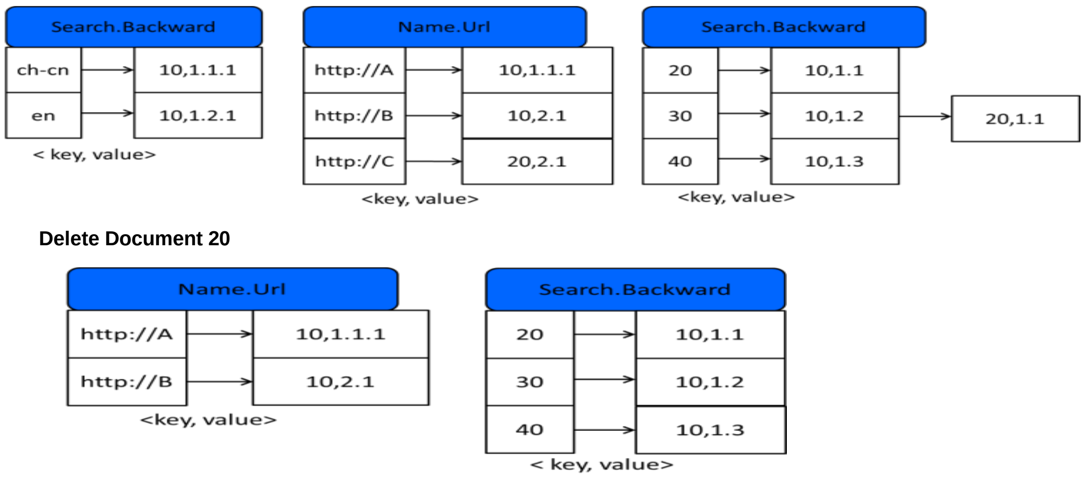


Figure 10. Change of UniHash Index for the Deletion of Document 20

6. Query Language

To illustrate our system is applicable, we employ a SQL-like language to describe the requests of precise query on nested data. The process of evaluating queries using the UniHash index is also presented in this section. In the context of distributed environment, there are always multi-users concurrently querying the data.

As in literature [7], a SQL-like language is designed to describe the queries executed in Dremel. Formally defining the language is out of scope of this paper; instead, we will showcase it with examples. Each query statement takes one or more nested tables and their schemas as input and produces a nested table and its schema as output. Let table t =

{r1, r2} contain the two records shown in Figure 3, consider the following query: search table t to find the total number of **Code** following path **Name.Country** in the documents whose **DocId** is less than 20 and the values of path **Name.Country.Domain** is “gb”.

```
SELECT Docid as Id,
Count(Name.Country.Code) with name as Cnt From t
Where Name.Country.Domain =’gb’ and Docid<20
```

Obviously, to answer this query, the system needs to locate the node precisely. The query can be described with a SQL-like statement:

In the statement, the fields are referenced with path expressions. Even there is no record constructor presented in the query, it will still produce a nested document as the result. If a nested record is viewed as a labeled tree, each label corresponds to a field name. To accomplish this query, we need to access the column whose path is **Name.Country.Domain** and the entries whose key is “gb” using the UniHash index, and determine if their **DocIds** are less than 20. If the conditions are met, return the **DocIds** and the UPaths of “gb”, i.e, 2.1.1 and 3.1.1 in our example. Then, continue to count the nodes **Code** following the common prefix **Name.Country** of **Name.Country.Domain** and **Name.Country.Code**, namely, to get the total number of the values of **Name.County.Code** whose UPaths start with 2.1 and 3.1. The result is 1 in the example records.

7. Experiments

In this section, we present the results of experiments conducted to verify the performance of UniHash. We first give some details on our experiment settings, including the datsets on which our experiments were carried and the computing environment.

7.1. Experimental Settings

The datasets used in the experiments are real and were downloaded at some recruitment websites (such as 58.com, ChinaHR, *etc.*,) by Web crawlers. We established the UniHash indexes for them with the algorithm proposed in the previous sections. Our datasets include three UniHash tables, Company, Search and Link. Each of them has three backups in HDFS. Table 2 displays the detailed information of the three tables.

The experiments were conducted over a cluster with 8 virtual machines installed on 4 physical machines. Each physical machine was configured with a quad-core Intel I5 3470 processor, 4GB RAM, and 1TB hard disk using Windows 7 Ultimate in 64-bit mode. Two virtual machines running CentOS 5.5 equipped with Drill, Hadoop-1.0.4 and HBase-0.90.4 were installed on each physical machine.

Table 2. Datasets used in Experiments

Table name	Number of records	Size(compresed)	Number of fields
Company	2+ million	95M	12
Search	1 million	59M	15
Link	1.5+ million	53M	9

7.2. Experimental Results

We designed five categories of experiments to examine the performance of processing precise queries with UniHash and the strategies applied in the implementation of the index.

Experiment I: Comparing the performance of UniHash with Drill on running queries Drill is an open source implementation of Dremel that uses repetition definition level to represent the structure of nested records. We implemented three queries with Drill and UniHash in order to verify their performance of executing precise queries and

```
Q1 : SELECT Addr. FROM Company
     Where Info..Name='Google'
Q2: SELECT COUNT(Name) FROM Company
     GROUP BY Country
Q3 : SELECT COUNT(Name) FROM Company
     WHERE Info..Name=' Coca-Cola'
```

aggregative queries on nested records. The three queries marked with Q1, Q2 and Q3 were executed on table Company. Among them, Q1 was an exactly query, Q2 was a general aggregative with group-by and Q3 was an aggregate query on a precise condition. Each query traversed the records in the table once. Table Company contains two million nest records, and each record has two repeated fields **Country** and **Info** which has two subfields **Name** and **Address**. This experiment was performed with Drill and UniHash index on 8 virtual machines.

Figure 11 depicts the time for processing the three queries using UniHash index and Drill. It is observed that even Drill outperforms UniHash index at aggregative queries (Q2 and Q3), it takes more time to answer the precise query (Q1). The reason behind this observation is that Drill expresses the location of the values in nested records with repetition and definition levels that could be the same for different values. This feature makes some of aggregative queries quite efficient. However, for answering precise queries, it has to costly investigate the keys separately in order to get the exact ones. On the contrary, an entry in the UniHash index preserves the UPath that describes precisely the location information for each value in the nested data, which results in locating the exact keys for precise queries directly in UniHash index rather than evaluating the keys one by one. That is why UniHash is better at performing precise queries.

Experiment II: Comparing the efficiency of precise queries on nested data through UniHash index and XPath According to our previous discussion, both XPath and UniHash can support precise queries on nested data. So it is meaningful to testify which one is better at answering precise queries with experiments. This group of experiments includes three queries, Q4, Q5 and Q6. All of these queries were designed to seek the data satisfying some exact conditions. Q4 and Q5 were evaluated on table Search. Table Search is a nested data source with 1.5 million records indexed by UniHash index. Each of these records has two repeated fields, **Date** and **Record**. Field Record consists of

```
Q4 : SELECT Amount FROM Search
     WHERE Record.UserId='413645'
Q5 : SELECT Record FROM Search
    WHERE UserId='413645' AND date BETWEEN
    '2013-8-10' AND DATEADD(day,1, '2013-8-10')
Q6 : SELECT Forward.Count(visited) FROM Link
     WHERE Backward.Url ='http://blog.sina.com'
```

two subfields, **UserId** and **Amount**. Q6 was executed on table Link. The records that generated table Link also have two repeated fields, **Backward** and **Forward**. Both of the two fields consist of two subfields **Url** and **visited**.The experiments measured the time for carrying out the queries with XPath on the original data and the SQL-like statements on the UniHash tables.

The result of the experiments is illustrated in Figure 12. It displays that compared to XPath, UniHash index improves the efficiency of processing precise queries (Q4 and Q5) about 80 times on table Search over XPath. Similar result also appears on the queries executed on other datasets. For instance, Q6 is a precise query conducted on table Link. Still, as we can see, UniHash spends much less time on responding the query than XPath, even table Link has 9 fields and 1.5 times more records larger than table Search. The experimental result implies that UniHash index offers advantages and good stability for precise queries.

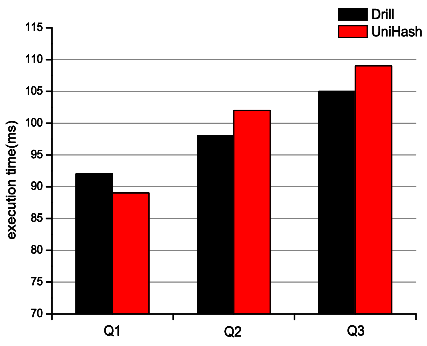


Figure 11. Comparison between UniHash and Drill

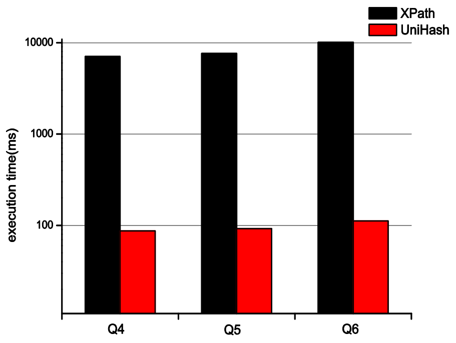
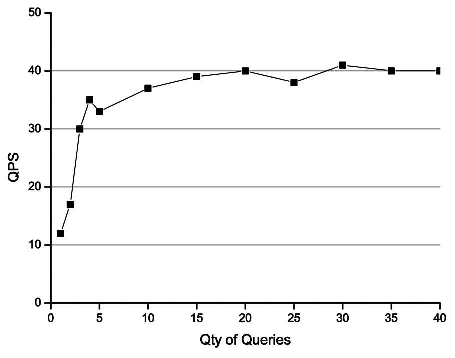


Figure 12. Comparison between UniHash and XPath

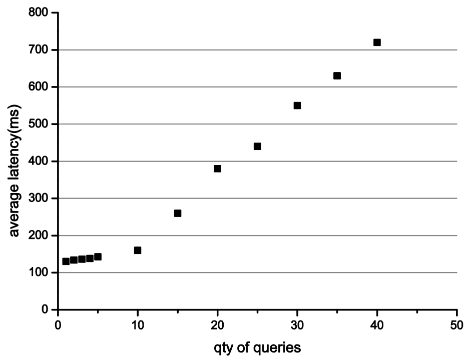
Experiment III: Investigating the performance of UniHash index with columnar storage To measure the advantage of managing UniHash with columnar storage, we tried to investigate the cost of executing precise queries of nested data on a single server. The metrics we employed to measure the cost are the throughout and the average response time for query requests when using a single-thread to process concurrent queries on the server. Suppose there are multiple users performing multiple queries on table Company at the same time, and the server's throughput is measured with PQPS (Processed Queries Per Second), that is, QPS= $\sum_{i=0}^{i=n} m_i / \sum_{i=0}^{i=n} \sum_{j=1}^{j=m_i} t_{i,j}$, where n is the number of processes, m_i is the number of queries tackled by process i in the period of observation and $t_{i,j}$ is time cost for query j processed by process i. In our experiment, the number of processes was 1 because we only used one process to execute the queries on each node. We measured the time for responding each query during the experiment.

According to Figure 13(a), PQPS of the system increases with the growth of concurrent queries before it reaches 40. Finally, the PQPS keeps constantly even the number of concurrent queries continues to go up. Of course, we can improve the PQPS in some degree by using multi-process with multi-threads. However, when we tried to perform the queries on the datasets with same size stored in records with the same configuration, the tasks did not finish in 2 hours. Compared to record storage, columnar storage is more suitable to store vast amount of data.

Figure 13(b) depicts the average response time for the queries varying with the number



(a) Effects of query concurrency on throughput



(b) Effects of query concurrency on average response time

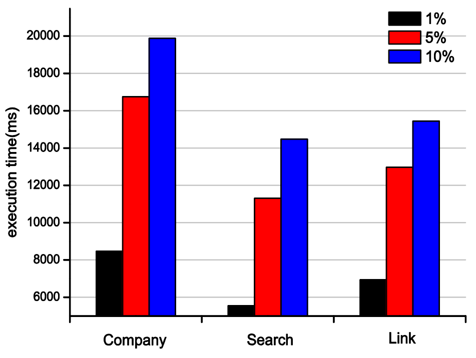
Figure 13. Effects of Storage Mode on the Query Efficiency

of concurrency. The figure illustrates that the average response time also increases as the concurrency rises. Concretely, it goes from 130ms and 850ms as the number of concurrent queries increases from 10 to 40. Obviously, it still takes much less time than the record storage often used RDBMS. UniHash performs over RDBMS because it employs columnar storage to store the entries, that helps it locate data values precisely in higher speed. With columnar storage, when executing queries, only the relevant entries are accessed, rather than scanning irrelevant records as the record storage does. Evidently, the observation of this experiment could also be extended to the situation of processing queries on clusters.

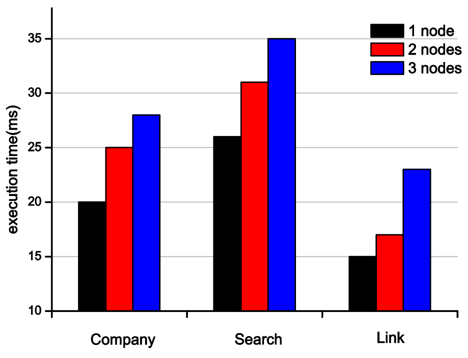
Experiment IV: Testifying the performance of maintaining UniHash index In the system of Dremel, the nested records are assumed to be read-only which means there is no change happened on the dataset. However, in our system, we consider using the UniHash index in some open circumstance such as on theWeb, where there is inevitably lots of modification on the data. Therefore, it is necessary to develop some strategies to efficiently support the maintenance of the UniHash as the indexed data changes. This group of experiments took deleting some nested records and some nodes (or fields) as the examples to examine the efficiency of maintaining the UniHash index with our proposed approach. Exactly, the experiments evaluated the time needed to modify the UniHash index when 1%, 5% and 10% records were removed from the original dataset, and 1, 2 and 3 nodes (fields) were deleted from the nested records.

In our strategy of deleting documents, we first built the postings for the documents being deleted and then dropped the corresponding postings from the UniHash index. This results in the time for modifying UniHash linear to the number of deleted documents. This trend is also reflected in Figure 14(a), that is, the time increases as the modified proportion of the datasets changes from 1% to 10%.

Even the time for deleting nodes in a data model goes up as the growth of deleted nodes, it is still kept in a low scale (less than 35ms). According to our strategy, all the entries in a common UniHashList are stored on the same HDFS blocks and marked by the same path, which makes it efficiently seek the paths of the deleted nodes. The performance of the algorithm is proven by the experimental results shown in Figure 14(b).



(a) Performance of Deleting a Set of Documents



(b) Performance of Deleting a Set of Nodes from a Data Model

Figure 14. Performance of Maintaining UniHash Index

8. Conclusion

Querying precisely on nested datasets has been widely applied in many fields. Building effective index structures for nested datasets is one of the most pivotal measures to ensure the performance of query processing. This paper proposes an index structure-UniHash, which efficiently supports the precise queries of nested data. An entry in a UniHash index is formed by the value and its postings that contain the DocId and the UPath of the value. The UPath expresses the location of a value in the indexed documents uniquely, which enables it to exactly locate a certain value in the document in an efficient way. To improve the performance of generating UniHash for large-scale nested data, in addition to implementing generation in MapReduce, some other strategies such as the columnar storage is also employed when processing queries using UniHash index. We demonstrate its performance with experiments that were conducted on tables of millions real records, and show that UniHash performs over some existing approaches on precise queries of nested data.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61202376 and No.61003031, and Shanghai Pudong New Area Science and Technology Development Foundation Innovative Funding for Scientific and Technological Cooperation Project under Grant No.PKJ2013-Z14.

References

[1] J. Gantz and D. Reinsel, 2011 digital universe study: “Extracting value from chaos”, http://china.emc.com/leadership/programs/digital-universe.html, **(2011)**.
[2] A. Campi, S. Guinea and P. Spoletini, “Fuzzy Querying of Semi-Structured Data”, Proceedings of IADIS International Conference Applied Computing, IADIS International Conference Applied Computing, San Sebastian, Spain, **(2006)**, pp. 241-248.
[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler and F. Yergeau, “Extensible markup language (xml) 1.0 (fifth edition)”, http://www.w3.org/TR/2008/REC-xml-20081126/, **(2008)**.
[4] T. Milo, S. Abitebour, B. Amann, O. Benjelloun and Fred Dang Ngoc, “Exchanging intensional xml data”, ACM Transactions on Database Systems, vol. 30, no. 1, **(2005)**, pp. 1-40.
[5] T. Bohme and E. Rahm, “Supporting efficient streaming and insertion of xml data in rdbms”, Third International Workshop on Data Integration over the Web, Riga, Latvia, **(2004)**, pp. 70-81.
[6] P. O’Neil, E. O’Neil, S. Pal, I. Cseri, G. Schaller and N. Westbury, “ORDPATHs: insert-friendly xml node labels”, Proceedings of the 2004 ACM SIGMOD 34 international conference on Management of Data, Paris, France, **(2004)**, pp. 903-908.

[7] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton and T. Vassilakis, “Dremel: interactive analysis of web-scale datasets”, Proceedings of the VLDB Endowment , vol. 3, no. 1, **(2010)**, pp. 330-339.

[8] A. Berglund, S. Boag, D. Chamberlin, M. F. Fern´andez, M. Kay, J. Robie and J. Sim´eon, “Xml path language (xpath) 2.0”, http://www.w3.org/TR/2007/REC-xpath20-20070123/, **(2007)**.

[9] N. Hoeller, C. Reinke, J. Neumann, S. Groppe, C. Werner and V. Linnemann, “Efficient xml data and query integration in the wireless sensor network engineering process”, International Journal of Web Information Systems, vol. 6, issue 4, **(2010)**, pp. 319-358.

[10] L. Libkin, W. Martens and D. Vrgoc, “Querying graph databases with xpath”, Proceedings of the 16th International Conference on Database Theory, Genoa, Italy, **(2013)**, pp. 129-140.

[11] J. Song, T. Kim and W. Kim, “Pattern-based extensible index technique for xml documents”, Proceedings of the 4th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering Data Bases, Tenerife, Canary Islands, Spain, **(2005)**, pp. 1-7.

[12] R. Kaushik, P. Bohannon, J. F. Naughton and H. F. Korth, “Covering indexes for branching path queries”, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, **(2002)**, pp. 133-144.

[13] N. Bruno, N. Koudas and D. Srivastava, “Holistic twig joins: optimal xml pattern matching”, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, **(2002)**, pp. 310-321.

[14] J. Li, C. Liu, R. Zhou, W. Wang, “Top-k keyword search over probabilistic xml data”, Proceedings of the 27th International Conference on Data Engineering, Hannover, Germany, **(2011)**, pp. 673-684.

[15] Y. Lu, W. Wang, J. Li and C. Liu, “Xclean: Providing valid spelling suggestions for xml keyword queries”, Proceedings of the 27th International Conference on Data Engineering, Hannover, Germany, Hannover, Germany, **(2011)**, pp. 661-672.

[16] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters”, Proceedings of 6th Symposium on Operating System Design and Implementation, San Francisco, CA, USA, **(2004)**, pp. 10-10.

[17] C. Seo, S. Lee and H. Kim, “An efficient inverted index technique for xml documents using RDBMS”, Information and Software Technology, vol. 45, no. 1, **(2003)**, pp. 11-22.

[18] D. Florescu and I. Manolescu, “Integrating keyword search into xml query processing”, Proceedings of the Ninth International World Wide Web Conference, Amsterdam, Netherlands, **(2000)**, pp. 119-135.

[19] T. White, “Hadoop: The Definitive Guide”, O ´Reilly Media, Inc., **(2009)**.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks”, Proceedings of the EuroSys Conference, Lisbon, Portugal, **(2007)**, pp. 59-72.

[21] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff and R. Murthy, “Hive: a warehousing solution over a map-reduce framework”, Proceedings of the VLDB Endowment, vol. 2, no. 2, **(2009)**, pp. 1626-1629.

[22] S. Khatchadourian, M. P. Consens and J. Sim´eon, “ChuQL: processing XML with XQuery using Hadoop”, Center for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, **(2011)**, pp. 74-83.

[23] L. Fegaras, C. Li, U. Gupta and J. Philip, “Xml query optimization in MapReduce”, Proceedings of the 14th International Workshop on the Web and Databases, Athens, Greece, **(2011)**.

[24] J. Lin and C. Dyer, “Data-Intensive Text Processing with MapReduce”, Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Boulder, Colorado, USA, **(2009)**, p. 1-2.