J Inf Process Syst, Vol.17, No.1, pp.37~50, February 2021
https://doi.org/10.3745/JIPS.04.0205

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Query Optimization on Large Scale Nested Data with Service Tree and Frequent Trajectory

Li Wang* and Guodong Wang**

Abstract

Query applications based on nested data, the most commonly used form of data representation on the web, especially precise query, is becoming more extensively used. MapReduce, a distributed architecture with parallel computing power, provides a good solution for big data processing. However, in practical application, query requests are usually concurrent, which causes bottlenecks in server processing. To solve this problem, this paper first combines a column storage structure and an inverted index to build index for nested data on MapReduce. On this basis, this paper puts forward an optimization strategy which combines query execution service tree and frequent sub-query trajectory to reduce the response time of frequent queries and further improve the efficiency of multi-user concurrent queries on large scale nested data. Experiments show that this method greatly improves the efficiency of nested data query.

Keywords

Caching, Frequent Sub-query Trajectory, Nested Data, Query Optimization, Service Tree

1. Introduction

The investigation reported by the Internet Data Center (IDC) predicted that by 2025, total data in the world will reach 175 ZB [1], of which over 75% will be unstructured. Query applications based on nested data (such as XML and HTML), one of the main forms of unstructured data, are becoming increasingly extensive. For example, one can search for academic articles on Google, or locate someone's position on a map. The characteristics of nested data which are big, non-standardized, and unstructured make it more complex and difficult for enterprises to analyze it and effectively extract information from it. The existing data retrieval technologies and tools no longer meet query needs, yet it is complicated and time-consuming to develop new query languages to support heterogeneous data retrieval. Therefore, effectively improving the efficiency of nested data query has become a hot issue for scholars.

Nested data storage models are complex. It is not just data values being stored, but also their corresponding structures. Melnik et al. [2] proposed that Dremel realizes nested data column storage, which can complete aggregate query on trillions of tables in a few seconds. In view of Dremel's failure in accurate positioning, the UniHash index technology is proposed in [3] to improve the precise query efficiency of nested big data. UniHash achieves good efficiency by building an index and using column

Manuscript received September 15, 2020; first revision November 16, 2020; accepted December 4, 2020.

Corresponding Author: Guodong Wang (guodong@shnu.edu.cn)
* Basic Courses Department, Shanghai Institute of Tourism, Shanghai, China (liwang@shnu.edu.cn)
** Teaching Affairs Office, Shanghai Institute of Tourism, Shanghai, China (guodong@shnu.edu.cn)

storage to improve retrieval speed, but it does not consider optimization from the perspectives of user query processing and historical query records.

As shown in Fig. 1, the nested data query processing workflow includes parsing, optimization, and execution. Parsing refers to transforming query statements into internal machine-recognizable expressions according to certain rules in order to execute nested data query. This step only transforms query statements into internal expressions, without optimizing them. The nested algebraic query expression determines the amount of space required for the query, so it is also necessary to optimize the nested algebraic query expression to speed up retrieval. Physical optimization is one of the methods to optimize query expression. Physical optimization calculates the cost of the query expression according to a cost model, and then optimizes the query expression and selects the optimal query [4]. In this process, it is necessary to decide the execution order and algorithm of each operation. The last step of query execution is the process of searching the nested source data and producing results according to the execution sequence generated in the physical optimization stage and related algorithms. With larger query data and more aggregate queries, it becomes highly necessary to optimize query execution.

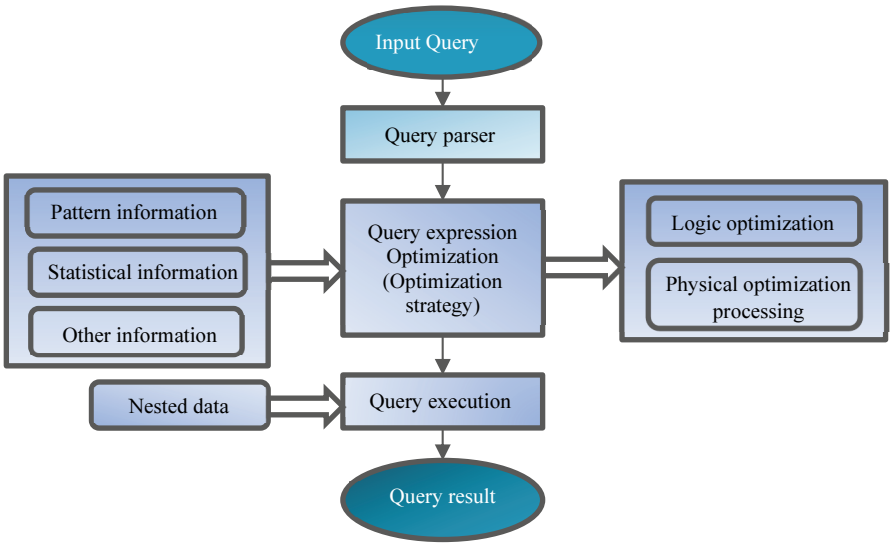


Fig. 1. Nested data query processing optimization system.

Because query requests are executed concurrently and many requests are similar, the effect of indexing is better for a single query request, but lower for repeated requests. Mining frequent subqueries through multi-user aggregate queries is valuable and meaningful, especially in large-scale data environments. Frequent pattern discovery, one of the most important parts of data mining, is widely used in shopping data analysis, clustering and classification, frequency graphing, and tree mining, etc. Meanwhile, it is common to apply caching in research on XML query optimization, but there are few studies on mining frequent subqueries based on caching for large-scale nested data. Therefore, based on the MapReduce platform, this paper uses the large-scale nested data indexing method in [3] to build an index, and based on it, proposes a three-tier query execution service tree to improve query throughput. At the same time, a caching technology is combined with frequent historical path mining to process frequent queries and reduce the response time of frequent queries.

2. Research Status

Many studies exist on improving query efficiency through indexes. In [5], the query performance of branch paths is improved by establishing covering indexes for branch paths of nested records. The study of Lu et al. [6] gives suggestions on misspelling of keyword queries and [7] puts forward XML keyword retrieval based on keyword density. Wei and Luo [8] put forward interval code reservation for XML keyword queries.

Query optimization is one of the key parts of data query, and querying interesting node-to-node relationships is the key point affecting query efficiency during processing. These query processing methods include the representative twig pattern query [9], match queries based on structural connection [10], B+ trees [11], l-indices [12], CB+-tree indexing [13], and optimal query optimization query expressions [4]. Caching is another technology to improve retrieval efficiency and is widely used in [14,15]. Commonly used distributed caching strategies, including full replication, partition, and near caching each have their own advantages and disadvantages in different scenarios [16]. Two important points in caching are cache update strategy [17] and cache access [18], and different strategies apply to different scenarios.

Frequent pattern discovery is one of the research hotspots for complex semi-structured data, such as for single frequent pattern tree mining [19], fast frequent subtree discovery of projection branches [20], induced frequent subtree discovery based on mined content [21], bottom-up subtree mining based on disordered trees [22], and association rules mining for XML data [23]. Therefore, this paper adopts the model in [3] and builds an index in the form of <key, value> pairs for nested data, and uses column-based storage in the following structure:

$$<term\ t, Posting(DocId, UPath)>$$

where term is the keyword in the nested document, Posting is a binary <DocId, UPath>, DocId represents the unique identifier of the document, and UPath represents the unique path of the keyword in the document. Then, the SQL-like language supporting nested document retrieval introduced in [2] is used for retrieval. It regards nested records as tag trees, each node as a field name. SQL input statements are nested tables and models, and the output query results are also nested tables and models. On this basis, a frequently nested data query optimization strategy combining service tree structure and caching is proposed, taking the query and the decomposed sub-query of each server as the key, the result returned after the query as the value. Frequent user subqueries are mined in the form of <SQL-like query, query result> in the cache on the query execution service tree, further improving the efficiency of data retrieval. This study helps improve the efficiency of nested data bug retrieval, and optimizes the following points:

- It proposes a query execution service tree to optimize execution of complex queries. Combining the web search method with advantages similar to MapReduce, it uses the query execution service tree to divide large queries into small simple ones, runs on multiple nodes concurrently, and finds the hierarchical structure of the optimal tree to further optimize the performance of complex queries.
- For multi-user repeated queries, it proposes to mine frequent historical query paths based on caching to further improve query throughput.

3. Query Optimization Based on Service Tree and Frequent Query Trajectory

3.1 Query Execution Service Tree

In practical applications, query requests are usually concurrent, which may cause bottlenecks in server processing. In addition to using multi-core technology to allocate multiple threads to servers to solve this problem, a query execution service tree can also be used to improve query throughput.

In the query execution service tree, after the query statement reaches the root server, the root server sends the request to the second-level service tree, which is composed of multiple servers, decomposes the request and sends it to the next level, and so on. Finally, the query is decomposed into query units and sent to the leaf server, which starts a corresponding number of threads according to the query units to retrieve the data in HDFS (Hadoop Distributed File System). In HDFS, each column is backed up in three blocks to ensure that the required data can be read smoothly. Given that it takes a lot of time and bandwidth to transmit data between servers, it can significantly improve the efficiency of aggregate query. However, it is not suitable for precise query without the need of multi-step decomposition. Experiments found that three-tier execution service tree query has the best efficiency for a certain amount of data. Fig. 2 is an example diagram of a three-tier query execution service tree.

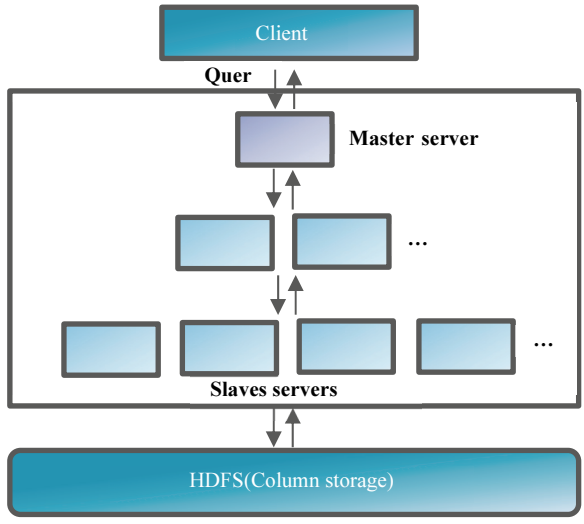


Fig. 2. Three-tier query execution service tree.

3.2 Frequent Query Mining Algorithm

This paper uses SQL-like statements to query data in column-based storage, which is similar to the retrieval of relational data, for the following aggregate queries:

```
DEFINE TABLE t as Name.Country. *
Select count(*) FROM t
```

when the user inputs the query in the root query execution service tree, the program first searches the cache of the root server to see if any key matches it. If so, it directly returns the result, and increases the

number of visits by 1; if not, the algorithm in the root server initializes a <key, value> pair, where the key is a string. That is, the whole request is a SQL-like query statement (recorded as query, similar sql); the value is the result returned by the query (recorded as result, initialized as null), and then the query is decomposed the next query execute. The second layer service tree first also matches the data in the cache, and so on until the result is returned to the root service tree, which receives the result, returns its value to the user, and assigns it to the result. Fig. 3 shows the frequent subquery discovery algorithm, and its process is described in detail below.

Algorithm 1. Frequent subquery mining algorithm

Input: Recently SQLs and results which compose the Lists D; the minsup; freList

Output: The Frequent lists, named freLists

```
1.  List FreSubSql(D, minsup, freList) {
2.      cacheList ← { }                          //defines an empty frequent set
3.      for each query q in D.keys {             //traverses set D, for each query q in D.
4.          if query.key equals q.keys           //key in cacheList is equal to query q.
5.              query.key.times++;
6.          else {                               //if it is not equal
7.              query.key = q.key;
8.              query.key.time = 1;
9.              query.key.value = q.key.value;
10.         }        //endif
11.     }  //endfor
12.     for each query in cacheList.keys {    //query for cacheList
13.         if query.time >= minsup {
14.             if query.key equals q in freList.keys
15.                 //puts the query whose statistics are greater than the support degree into freList
16.                 q.key.flag = current_time;
17.             else{
18.                 query.key.flag = current_time;
19.                 freList.add(query);
20.             }         //endelse
21.         } //endif
22.     } //endfor
23.     return freList;
24. }
```

Fig. 3. Three-tier query execution service tree.

The input of the algorithm is historical queries and their result set D (the old one is stored on the disk) for each period (for example, updated once a day), and each record of this set is a key-value pair formed by the query statement query and the corresponding return result. The output of the algorithm is the frequent query set freList obtained by mining at regular intervals. The algorithm counts the frequencies of different historical queries in this period—that is, the support degree—and saves them in the cacheList set. Then, the queries in the cacheList that are greater than the support degree are put into freList, timeFlag is created to facilitate subsequent cache update, and finally the freList frequent result set is returned. In

this way, frequent historical queries that satisfy minsup and their cache times are preserved in freList, in which each record is also a key-value pair. The type key is a string, that is, the whole query statement (recorded as a query), and value contains two parts: the cache time of the query (recorded as timeFlag) and the result returned by the query (recorded as result). Its structure is as follows:

(*query,Posting<timeFlag,result>*)

Frequent historical queries are stored in the root server cache after they are found in the root server; Similarly, the historical queries of the server on the next tier are mined for frequency, and stored in the cache of the second-tier server to improve efficiency. Because D stores the data of the latest period, it is necessary to re-run Algorithm 1 every once in a while, and call the algorithm shown in Fig. 4 to update the frequent query results in the cache regularly.

Algorithm 2. Cache frequent query update algorithm

```
Input:  freList
Output:  freList
1.  List updateFre(freList, period) {
2.     for each query in freList.keys  {
3.        //traverses freList set for each query in freList
4.        if query.key.flag > period  //deletes the recently unused query
5.           freList.remove(query);
6.     }         //endfor;
7.     return freList;
8.  }
```

Fig. 4. Cache replacement algorithm.

3.3 Nested Data Query Optimization Based on Service Tree and Frequent Trajectory

3.3.1 Frequent query architecture

This paper adopts the service tree structure for query execution. So different levels of service trees have their own caches. Fig. 5 shows the overall nested data query architecture, which is composed of several main parts: the cache querier, which saves the returned query results in the cache system; the frequent query cache, which stores frequent queries or subqueries and their results; the frequent historical query miner, which finds frequent query patterns according to the user's historical query path; the query matcher, which finds the query in the cache that completely matches the new query; the index manager, which is used to the maintain nested data indexes to the retrieve nested data sets; and the universal query builder, which searches the resulting index structure.

When the cache query builder receives the nested data query, it first checks the cache to determine whether the same query already exists. If it is completely the same, it directly returns the result. If it is partly the same, it decomposes the query, obtains the partial matched results from the cache, and then transmits the unmatched parts to the lower-tier server. If there is no match, it decomposes the query to

the second-tier query server (a simple query does not need to be decomposed). When the second-tier cache querier receives the subquery, it also compares it with queries in its cache. If it finds a match, it returns the results; if it does not, it decomposes it again and transmits it to the third-tier server, and so on. If it reaches the end without finding the same query, it directly queries the general query builder to obtain the result. In this process, each layer uses a frequent historical query miner to dynamically mine frequent nested query patterns, which exists in the frequent query cache and uses a certain cache update strategy.

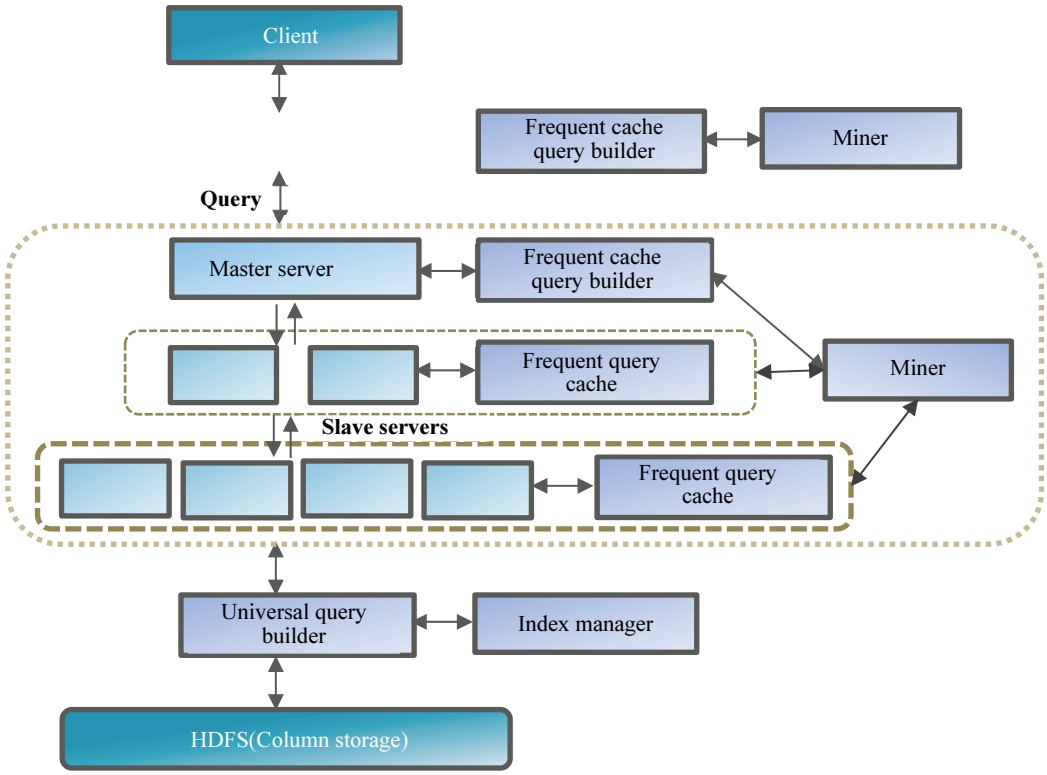


Fig. 5. Query architecture based on service tree and caching.

3.3.2 Cache replacement strategy in nested query architecture

When the query system runs and the time interval reaches the preset system threshold, the frequent historical query miner starts to automatically mine new frequent queries, which are stored in the frequent query cache. When the cache is full, it is updated with a certain replacement strategy. While the miner is mining, the minimum support is automatically and continuously adjusted in order to make full use of the cache; that is, with a longer interval, the query volume may be larger, and the minimum support value increase, otherwise it decreases. The changes follow the following rules:

$$minsup = minsup * (1 \pm change\_percent) \quad (1)$$

Since the replacement of frequent queries in the cache is related to the latest time of entering the buffer,

the time of its access, and the number of visits. The most important metric parameter is last access time. The most commonly used replacement algorithms include the Least Recently Used algorithm (LRU), the Least Frequently Used algorithm (LFU), and the Most Recently Used algorithm (MRU). However, the time factor of these three replacement algorithms are too unitary. And the efficiency of these three replacement algorithms is insufficient enough high. Hence the time factor is divided into several layers in this research, the lowest of which is queries that have not been accessed, which will be the first object to be replaced. However, the most recently accessed queries are on higher layers and the replacement decisions are decided according to the time of cache access. Therefore, when the cache is full, the first query to be removed is the queries which have not been accessed recently and have been stored in the cache for a long time. In the experimental part, we will compare three replacement algorithms on nested data query based on service tree and cache query architecture.

4. Experimental Evaluation

4.1 Experimental Setup

The experimental nested dataset was obtained by crawling recruitment websites and establishing a UniHash index. The UniHash index table is shown in Table 1. “Company” comes from a dataset of about 1.9 GB with more than 2 million records, with 12 fields, and the records are about 95 M when they are not copied following compression. “Search” comes from a dataset of about 1.2 GB with 15 fields; “Link” has 9 fields. Each table is backed up three times in the HDFS. The average of 10 execution results is taken as the experimental value.

Table 1. Experimental data set

Table name	Number of records	Size after compression	Number of attributes
Company	2+ million	95 M	12
Search	1 million	59 M	15
Link	1.5+ million	53 M	9

The experimental environment is configured follows. Four PCs are configured with 64-bit Windows 7 Ultimate, 4 GB of RAM, a 4-core Intel I5 3470 processor, and a 1 TB hard disk. Each PC is equipped with two CentOS 5.5 virtual machines, with 1 G of memory and a 200 G disk. Eight virtual machines are equipped with Hadoop-1.0.4 and HBase-0.90.4, forming a laboratory cluster.

In the experiment, historical queries are generated by simulation. In order for the simulated query to adhere to real user behavior, the data model of the data set was transformed into a tree pattern, which is traversed layer by layer from top to bottom to generate 1,000,000 different query tree patterns (QTP). From these query trees, 10,000 to 1,000,000 databases with different sizes were randomly selected, then these trees were transformed into SQL-like query datasets.

4.2 Experimental Contents

The effectiveness of optimizing nested data retrieval based on frequent pattern mining of the service

tree and cache is investigated in the following four ways.

Experiment 1: Comparing the performance of different query types by using the service tree

The following three queries are performed on Company. Query 1 is a precise query, Query 2 is an aggregate query, and Query 3 is an aggregate query containing precise information. Each execution traverses the data once. Company contains 2 million nested records, each of which has the duplicated fields of Country and Info, as well as nested subfields of the fields such as Name and Addr. In this experiment, the retrievals based on Drill and UniHash are performed on 7 nodes.

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'

Experiment 2: Comparing the influence of number of executive service tree layers on precise query speed

In this experiment, the three queries Q4 and Q5 are based on different data sets with precise information. Among them, Search comes from 1.5 million records, each of which has duplicate Date and Record fields, and the Record field includes two subfields: UserId and Amount (number of visits). The nested records of Link contain two repeated fields, Backward and Forward, both of which contain Url and visited subfields. To ensure the same scanning speed, each topology has 2 leaf servers, the secondary topology is 1:2, the tertiary topology is 1:2:2, and the quaternary server is 1:2:2:2. This experiment compares the change in query execution time between Q4 and Q5 when the number of service trees is between 2 and 4, and tests the influence of number of service trees on precise query performance.

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')

Experiment 3: Investigating the efficiency when frequent history retrieval is used

The experiment compares the efficiency with frequent historical query to without under different historical query data quantities.

Experiment 4: Testifying the effect of the nested data query based on service tree and cache query architecture

SQL-like queries are used as the input, and in order to test the effect of the cache module and whether the replacement strategy is effective, the query time is tested under different support thresholds and cache update intervals by comparing the LRU, LFU, and MRU replacement algorithms.

4.3 Analysis of Experimental Results

Experiment 1: Comparing the query performance with the service tree

Compared with the query efficiency based on seven nodes without the query execution service tree, according to Fig. 6(a), the time of Q1 query increases somewhat, while the performance optimization is significant for Q2 and Q3 aggregate queries. This result is because for simple and precise queries using a parallel query execution service tree, the servers need time to transmit query results, so the efficiency improvement is limited. For relatively complex clustered queries, on the other hand, the execution service tree breaks them down into simple queries which are executed in parallel, significantly improving the efficiency.

Experiment 2: Comparing the effect of number of tiers in the service tree on precise query efficiency

This experiment tests the time used to execute Q4 and Q5 precise queries on the same dataset on different levels of the service tree, and investigates the influence of number of tiers on exact query performance, in order to determine the optimal number of tiers. Fig. 6(b) shows the experiment results. The horizontal axis represents two queries, and the vertical axis represents the change in their execution time between 2–4 service tree tiers. Increasing the number of tiers from 2 to 3, the execution time decreases, while above 3, the time begins to increase. Analyzing the reasons, the service tree can decompose aggregate queries into query units to reduce the time, but for precise queries, there is no need to access irrelevant DFS blocks because there are few data associations. In addition, data transmission between servers requires time and bandwidth. Too many service tree tiers results in more data being transmitted and an increase in overall query time. Therefore, the experiment shows that a three-tier query is suitable for precise query.

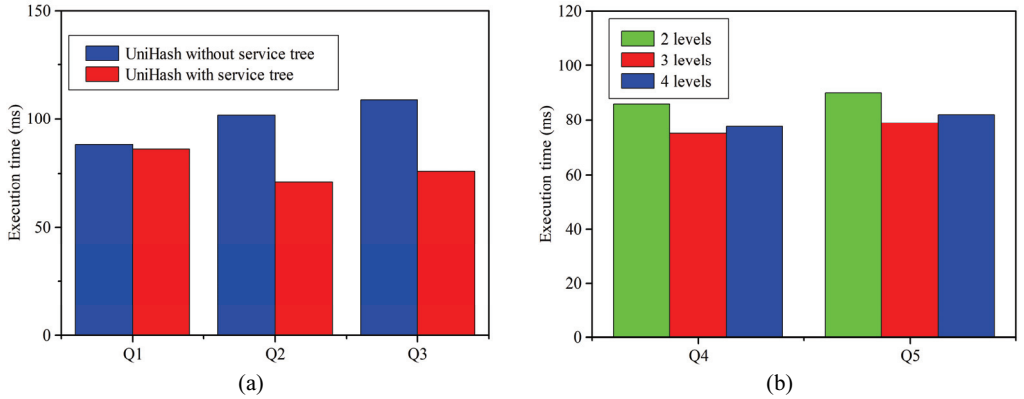
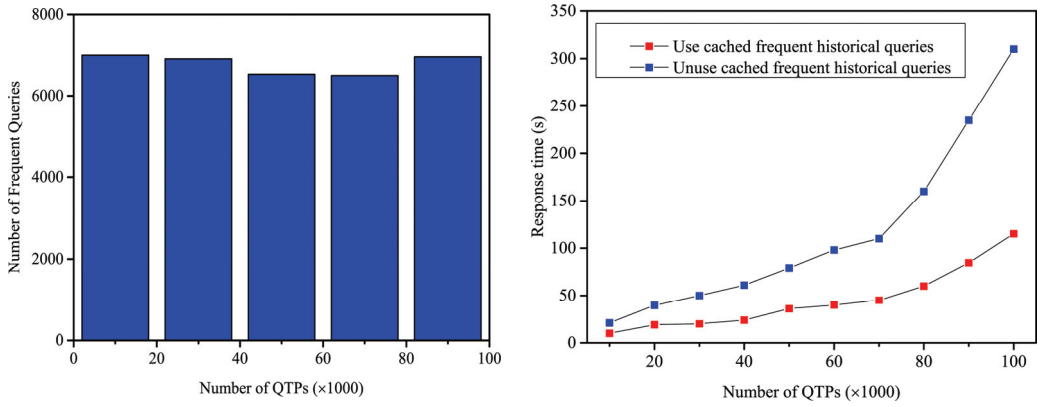


Fig. 6. (a) Comparison of query execution efficiency under different query types. (b) Comparison of precise query execution time of different levels of service trees.

Experiment 3: Investigating the number of frequent historical queries under different amounts of historical query data

During the experiment, the minimum support was set at 1%. Fig. 7(a) shows that with different amounts

of historical queries, the number of frequent historical queries generated is about 7,000. Frequent historical queries are saved in the cache to execute the queries. Fig. 7(b) shows that under different amounts of historical query data, by comparing the query efficiency of cached frequent historical queries with that of unused queries, the query efficiency is about 2–2.7 times that of unused queries, and the query time increases with larger data sets.



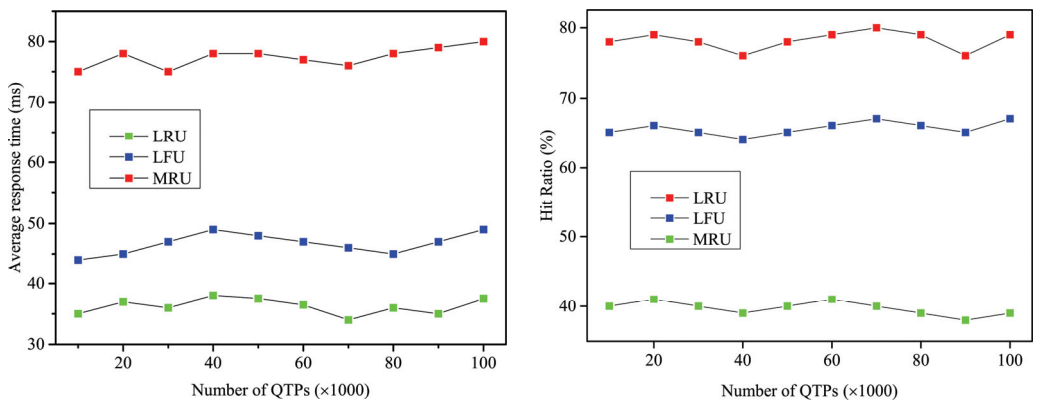
(a) (b)

Fig. 7. (a) Comparison of the number of frequent queries mined under different query data sets. (b) Comparison of query response time under different query data sets.

Experiment 4: Testifying the average response time efficiency of all queries using the three caching strategies under different amount of historical query data

Fig. 8(a) shows the response time of queries for different historical query data when 100 queries are stored in the cache. Using the frequent historical query discovery algorithm, the support threshold is specified as 1%. In this experiment, average response time (ART) is defined as follows:

$$ART = Total\ response\ time\ of\ all\ queries / Number\ of\ queries\ executed \quad (2)$$



(a) (b)

Fig. 8. (a) Comparison of query response time of three update strategies. (b) Comparison of hit rates of three update strategies under different query data sets.

Fig. 8 shows that the MRU caching algorithm has the lowest efficiency, and its average response time is about twice that of LRU caching. LFU is in between. However, the changes in these strategies are relatively small, and they tend to be stable. It is assumed that the hit rate calculation formula is Qcache_hits/(Qcache_hits+Com_select), where Qcache_hits represents the number of hits, and Com_select represents the number of cache misses. Fig. 8(b) shows the hit rate of queries under the three cache replacement algorithms. Because the number of frequent queries mined is different under different datasets, both Qcache_hits and qcache_inserts increase with larger datasets, while the ratio doesn't change much. Finally, because the cache system automatically updates at a certain time, and can adaptively adjust the support in response to changes in the optimal data volume, the query execution time and hit rate of a certain query data set under different support levels does not change much.

5. Conclusion

Nested data is one of the main forms of unstructured data. Its query applications are becoming more and more extensive, and queries are being executed concurrently. Effectively improving the efficiency of nested data retrieval is a research hotspot in various fields. This paper first establishes an index of nested big data in a MapReduce environment. Based on this, a nested data query optimization technology combining a three-tier query execution service tree and cache-based frequent subquery mining technology is proposed to further improve the efficiency of nested data query. Experiments show that this technology has significant advantages over existing technology. However, it takes a lot of time to build the index and space to store it. Therefore, it is necessary to consider an index compression strategy in the future to complete efficient query of big data.

Acknowledgement

The authors gratefully acknowledge the financial supports from Cultivating Academic Key Teacher project by Shanghai Institute of Tourism (No. E3-0250-20-001-031).

References

[1] A. Patrizio, "IDC: Expect 175 zettabytes of data worldwide by 2025," 2018 [Online]. Available: https://www.networkworld.com/article/3325397/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html.

[2] 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. 1-2, pp. 330-339, 2010.

[3] L. Wang, D. Peng, and P. Jiang, "Improving the performance of precise query processing on large-scale nested data with UniHash index," *International Journal of Database Theory and Application*, vol. 8, pp. 111-128, 2015.

[4] J. Ning, J. Liu, and D. Ye, "Novel approach for extracting XML schema definition based on content model graph," *Computer Science*, vol. 37, no. 6, pp. 179-185, 2010.

[5] Y. J. Fan, C. H. Zhang, S. Y. Wang, and Y. F. Hu, "IRST(k,l)-Index: an efficient XML structural index for branching path queries," *Journal of Chinese Computer Systems*, vol. 30, no. 8, pp. 1546-1554, 2009.

[6] Y. Lu, W. Wang, J. Li, and C. Liu, "XClean: providing valid spelling suggestions for XML keyword queries," in *Proceedings of 2011 IEEE 27th International Conference on Data Engineering*, Hannover, Germany, 2011, pp. 661-672.

[7] Z. Y. Qin, Y. Tang, H. Z. Xu, and U. Huang, "Study on keyword retrieval based on keyword density for XML data," *Journal of Software*, vol. 30, no. 4, pp. 1062-1077, 2019.

[8] D. P. Wei and D. Luo, "An XML keyword query algorithm based on interval reserved coding," *Computer and Modernization*, vol. 2019, no. 10, pp. 17-20, 2019.

[9] B. Kimelfeld and Y. Sagiv, "Matching twigs in probabilistic XML," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, Vienna, Austrai, 2017, pp. 27-38.

[10] D. Li, Z. Deng, and Z. Li, "Structural join processing for XML based on MapReduce," *Journal of Frontiers of Computer Science & Technology*, vol. 10, no. 8, pp. 1080-1091, 2016.

[11] S. Rosnan, N. Abd Rahman, S. M. Hatim, and Z. H. Ghul, "Performance evaluation of inverted files, B-Tree and B+ Tree indexing algorithm on Malay text," in *Proceedings of 2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, Kedah, Malaysia, 2019, pp. 1-6.

[12] A. Bandura and O. Skaskiv, "Functions analytic in a unit ball of bounded L-index in joint variables," *Journal of Mathematical Sciences*, vol. 227, no. 1, pp. 1-12, 2017.

[13] C. Ma, H. Xu, B. Yao, L. Wang, and H. Zhu, "XML temporal query technology based on CB+-tree index," *Journal of Chongqing University of Science and Technology (Natural Science Edition)*, vol. 2016, no. 5, pp. 75-77, 2016.

[14] A. V. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies," in *Proceedings of 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA, 2018, pp. 96-109.

[15] R. Tandon, "The capacity of cache aided private information retrieval," in *Proceedings of 2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Monticello, IL, 2017, pp. 1078-1082.

[16] X. L. Qin, W. B. Zhang, O. J. Wei, W. Wang, H. Zhong, and T. Huang, "Progress and challenges of distributed caching techniques in cloud computing," *Journal of Software*, vol. 24, no. 1, pp. 50-66, 2013.

[17] M. S. A. Khaleel, S. E. F. Osman, and H. A. N. Sirour, "Proposed ALFUR using intelegent agent comparing with LFU, LRU, size and PCCIA cache replacement techniques," in *Proceedings of 2017 International Conference on Communication, Control, Computing and Electronics Engineering (ICCCCEE)*, Khartoum, Sudan, 2017, pp. 1-6.

[18] P. Boonma, J. Natwichai, K. Khwanngern, and P. Nantawad, "DAHS: a distributed data-as-a-service framework for data analytics in healthcare," in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*. Cham, Switzerland: Springer, 2018, pp. 486-495.

[19] D. Jiang and L. Li, "Frequent itemset mining algorithm based on UFP-tree," *Computer Technology and Development*, vol. 2019, no. 10, pp. 175-180, 2019.

[20] C. Zhao, Z. Sun, and J. Zhang, "Frequent subtree mining based on projected branch," *Journal of Computer Research and Development*, vol. 43, no. 3, pp. 456-462, 2006.

[21] S. Hido and H. Kawano, "AMIOT: induced ordered tree mining in tree-structured databases," in *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM)*, Houston, TX, 2005, pp. 170-177.

[22] F. Luccio, A. Mesa Enriquez, P. Olivares Rieumont, and L. Pagli, "Bottom-up subtree isomorphism for unordered labeled trees," Dipartimento di Informatica, Universita di Pisa, Italy, 2004.

[23] Y. F. Yang, D. Y. Wang, and Y. J. Hu, "Positive and negative association rule mining on XML data streams in database as a service concept," *Manufacturing Automation*, vol. 34, no. 10, pp. 109-112, 2012.

Li Wang **https://orcid.org/0000-0003-3428-0787**

She received M.S. in software engineering from University of Shanghai for Science and Technology in 2015. She is an assistant in Basic Courses Department, Shanghai Institute of Tourism, Shanghai. Her research interests include cloud computing and data mining.

Guodong Wang **https://orcid.org/0000-0002-9458-4638**

He received M.S. and Ph.D. degrees from Shanghai Normal University. He is currently in Teaching affairs office, Shanghai Institute of Tourism, Shanghai, China. His current research interests include cloud computing, big data in tourism and data mining.