Cache System for Frequently Updated Data in the Cloud

Fusen Dong, Kun Ma, Bo Yang
Key Laboratory of Network Based Intelligent Computing
University of Jinan
No. 336, West Road of Nan Xinzhuang, Jinan
CHINA
fusendong@hotmail.com, ise_mak@ujn.edu.cn, yangbo@ujn.edu.cn

Abstract: - Maintaining data indexes and query cache becomes the bottleneck of the database, especially in the context of frequently updated data. In order to reduce the burden of the database, a cache system for frequently updated data has been proposed in this paper. In the system, update statements are parsed firstly. Then updated data are saved as key-value pairs in the cache and they are synchronized into the database at idle time. Experimental results show that the proposed cache system cannot only accelerate the data updating rate, but also improve the data writing ability in maintaining indexes and consistency of cache data greatly.

Key-Words: - Frequently updated data, Query cache, Data index, Cache system, Update merging method, Cloud database

1 Introduction

With the development of cloud computing, cloud databases play a basic role in data analysis [1]. Since distributed cloud data are large-scale, data access becomes a terrible problem in cloud data processing [2] [3] [4]. Generally, query cache and data index are utilized to increase the performance of data reading. By storing the query data into high-speed hardware and getting them from the cache instead of databases, query cache speeds up data reading. Besides, the process of data access can be accelerated more by indexes which sort data logically in database.

Therefore, the combining of these two methods is used to improve the performance of data reading. However, data are usually updated frequently in many cases, e. g. mobile network. As it is known that the data index and cache data should be maintained with data updating, the work maintaining data indexes and consistency of cache data becomes difficult for database when data are frequently updated. Herein, the performance of database is seriously affected in this case [5] [6].

A cache system for frequently updated data (CSFUD) is introduced in this paper, which addresses the maintenance issue of the index and query cache in database. The cache system saves the updated data over a period time and then the data are synchronized to the database at system's idle time. In this process, update operations are optimized by update merging method (UMM), which merges operations for the same data. By using the system, data updating speed is improved and the database becomes much more effective in processing frequently updated requests.

The system can be applied not only to document-oriented NoSQL databases in the cloud, such as MongoDB [7], CloudDB [8], but also to the rational databases like Oracle and DB2. In order to facilitate the discussion, MongoDB is chosen as an example to introduce the system in this paper. So far, related researches of maintaining index have be done, e.g., IBM and Microsoft. However, the existing methods can quicken index maintenance rate, but they could not reduce request times.

This paper is organized as follows. Section 2 discusses the related work recently. The data model is introduced in section 3. In section 4, cache system architecture is addressed. Section 5 presents and discusses the experimental results. Brief conclusions and future research directions are outlined in the last section.

2 Related Work

When data are updated, indexes and query cache need to be maintained to ensure their availability and consistency. That is why database performance is affected when the data are frequently updated. Therefore, related work mainly focuses on maintaining indexes and consistency of cache data.

For the query cache, there are few appropriate approaches at present. Query cache is considered as an effective way to speed up data reading and has

been widely used. Traditionally, cached data are maintained in a fixed period. Nevertheless, maintaining cache data in a fixed period is not so effective though it is very easy，because data keep updating and the cache data are not always consistent with them in database [9]. To solve the problem, Sugon and many other companies have proposed the solution that utilizes middleware to detect data updating operations and maintains cache data [10]. Unfortunately, it is not clear that the middleware has the ability to adapt the frequently updated data.

For data indexes, researchers have made great achievements. As data index is a kind of data structure that is stored in the database, index maintenance is optimized by changing data structures and maintaining strategies. The main methods can be concluded as follows:

- **Improved tree structures**. Several researchers have proposed the improved tree structures to store indexes, which include TPR*-tree, EuTPR*-tree, Rsb-tree, and so on [11] [12] [13]. The new structures can be used for e.g., multidimensional data, mobile network data. Experiments have demonstrated that the improved structures have accelerated data indexes maintaining speed at the cost of slight query performance degrade.
- **Auxiliary index**. IBM has proposed an auxiliary index method to assist index maintenance in large-scale data environment [14]. This method uses other data structure as an auxiliary index to assist maintaining index. In case of data updating, database maintains index and auxiliary index at the same time. If one of these two maintenances is completed, the data updating operation will complete. The method improves individual index maintenance speed, but it is known that many indexes may be created to index data and every index need an auxiliary device. Therefore, the method may become low efficient when data have many indexes.
- **Asynchronous index maintenance**. Asynchronous index maintenance method is another effective way. It provides techniques for asynchronously maintaining database index or sub-indexes. Indexes maintenance actions are postponed when data are updated [15]. Besides, the index is divided into many sub-indexes according to its length. Data updating information is stored in a table. When a sub-index is used, database maintains it by update information stored. The method reduces the burden of maintaining index greatly. However, indexes must be maintained once the data are queried in this method, thus it may be not so effective for the data which are frequently read.

These methods mentioned above have made great contributions to maintaining the indexes and consistency of the query cache. However, in the case that data are updated frequently, they cannot reduce the update times like CSFUD.

3 Data Model in Cache

3.1 Storage of Updated Data

Updated data are saved as key-value pairs in the cache. As shown in Fig. 1, id is the primary key of a record in database, and it is used as the key in this model. Value contains two parts: update type and update fields. Update fields are the fields of a record being updated. Updated data are stored in datasets. The cache may include many datasets. Update data, belonging to the same collection, will be stored into a dataset. That is to say, datasets are used to correspond with the collections or tables in database.

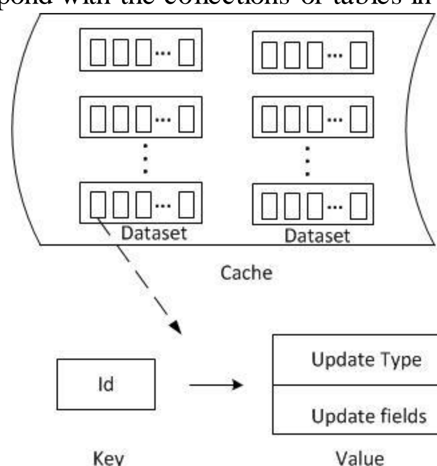

Fig. 1 Storage of updated data in cache.

Update operations are divided into three types: *Insert*, *Update* and *Delete*. For *Insert* type, the update statement includes all non-null fields of a record except id. Therefore, key is not confirmed and it is necessary to calculate a virtual id instead of the id from database. The virtual id can be created by checking id generation strategy in database. For example, id is usually generated automatically and the process is controlled by the Java class "ObjectId" in MongoDB. Therefore, the java code can be used in the cache to generate a virtual id as the key of inserted data. For *Update* type, update data usually include a part fields and the id also cannot be obtained from statements directly, but it can be gotten from the database by query the update criteria. For *Delete* type, update criteria is to get the

id of the record, too. The id and update type is stored but update fields are null in the model.

3.2 Update Merging Method

In the cache, data may be updated more than once in a short time. Therefore, update merging method (UMM) is introduced to merge update operations of the same record. This method improves the cache efficiency.

UMM includes two parts: type merging and field merging. Type merging merges several update types to one type when data are updated several times. Field merging means that the updated fields are merged together according to the mergence rules.

For type merging, mergence rules shown in Table 1. Action one is the update operation before and action two is the update operation later.

Table 1 Update Type Merging Rules

Action One	Action Two	Merging Results
Insert	*Insert*	*Fault*
	Update	*Insert*
	Delete	*Ignore*
Update	*Insert*	*Fault*
	Update	*Update*
	Delete	*Delete*
Delete	*Insert*	*Insert*
	Update	*Fault*
	Delete	*Fault*

Different mergence rules are shown in Table 1. In merging results column, *Fault* means that the later action will return false while *Ignore* shows that all the actions of the data will be ignore and the update data should be deleted from the cache.

For field merging, the operations should follow the rules below:

- The same fields: later covers before. Later actions are priority to the earlier. If the same fields have been updated repeatedly, the later operation will cover the earlier operation.
- The different fields: combine together. If actions update the different fields of a record, the updated fields should be combined.

4 System Architecture

The cache system operates the database as a middleware directly. As shown in Fig. 2, data requesting from the server should be processed by CSFUD firstly. Then the system controls update requests and synchronizes them at idle time. Updated data synchronization is a delayed operation and the synchronizing time is determined by the synchronization module in CSFUD, which is introduced in section 4.1. It is the delayed synchronization action that reduces the burden of database in maintaining indexes and caches.

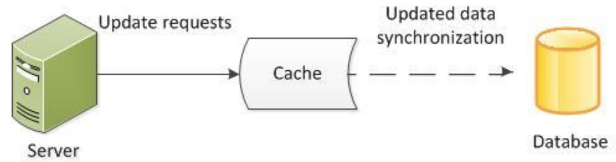

Fig. 2 Architecture of cache system for frequently updated data.

As shown in Fig. 3, the system contains several modules. Update query separator separates the data update and query requests (data definition requests will be sent to databases directly). Data update requests are sent to the cache controller, which parses update statements and updated data in the cache, while reading requests are processed by the query controller. Query controller executes query requests separately in the database and cache. Then the result corrector corrects the result from database according to the cache data. The dashed line shows that cache controller needs to query the update criteria in order to confirm that the record is to be updated. Through the query results, the key will be fetched.

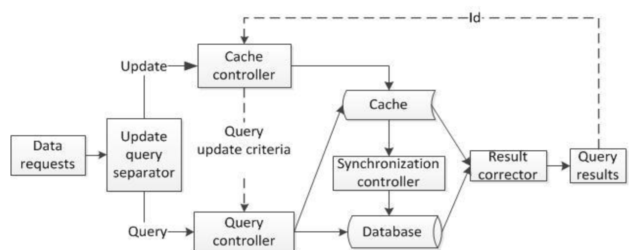

Fig. 3 Logical diagram of cache system for frequently updated data.

4.1 Cache Writing

The cache controller processes the update statements. It saves id as the key while the updated types and fields are stored in value. Unfortunately, the id usually cannot be gotten from the update statements directly. The statement usually contains two parts, i.e., update content and update criteria. The update content is that the update fields and update criteria show the record to be updated.

Therefore, cache controller parses the statement and transforms update criteria to query statement. The process has been shown in Fig. 4. After the cache controller receives an update statement, it parses update statement and gets the update content and update criteria firstly. Then, update criteria is transformed into query statement. Through querying the data, id is gotten from database. Finally, cache controller combines the update content with id and saves them.

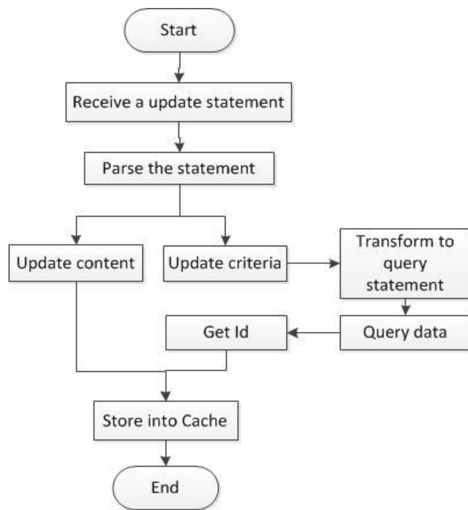

Fig. 4 Flow diagram of cache writing.

4.2 Cache Reading

In CSFUD, it is difficult to get the accurate data from the database since updated data were stored in cache. Therefore, the query results should be corrected by cache data. The detail process is depicted in Fig. 5, data queries are handled by the query controller and result corrector.

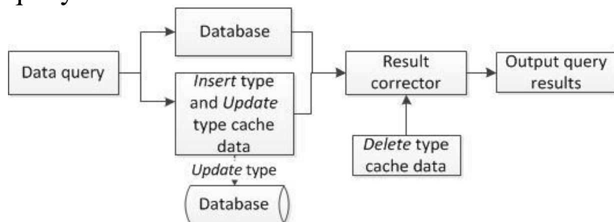

Fig. 5 The process of data query.

When query controller receives a simple query statement, it executes the query request in the database and cache at the same time. Database and the cache execute the request separately and deliver query results to result corrector. For the cache data, only *Insert* and *Update* types are suitable to execute the query statement. For the *Update* type data, there are only a part of record fields stored in the cache. Therefore, CSFUD needs to connect the database to get all the fields of the record if the stored fields meet the query criteria. Result corrector merges the result delivered from the database and cache, and then corrects them by *Delete* type cache data. The merging operation needs to meet the mergence rules shown below:

- The same records: The query result from the cache covers that from database. From section 3, *Update* type data are stored in both database and cache. Apparently, the records in the cache are more accurate than them in the database. Therefore, if both of a record in database and cache meets query criteria, it means the record can be gotten in the query results of database and cache. Herein, the query results from cache need to replace them from database.
- The different records: Combine together. The *Insert* type data is only stored in cache, thus we cannot get them from database. When the data meet query criteria, we need to combine the result from cache with it from database.

4.3 Cache Synchronization

Cache synchronization is controlled by the cache controller module in the cache system. Update data will be written to database in this process. The action is triggered by the trigger variable θ and the flow chart of cache synchronization process has been shown in Fig. 6.

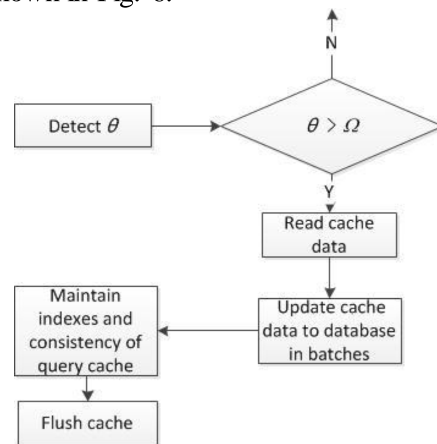

Fig. 6 The process of Cache synchronization.

Cache controller detects trigger variable θ. If θ is greater than Ω, (which means that the θ meets the trigger condition) cache controller will trigger the synchronization action. Synchronization controller reads t e cache data and transforms them to update statements. When the database receives these statements, it updates data firstly, then maintains

data indexes and query caches. Data are updated in batches since it is more effective than individual update [16]. At last, synchronization controller flushes the cache after data are updated successfully in database.

As the θ controls synchronous operation, it is very important in the module, which is affected by many factors. The calculation formula is:

$$\theta = k_3 \frac{PN(k_1 s/S + k_2 t/T)}{pn}. \quad (1)$$

In (1), the parameters *p, n, s, t* imply the amount of the page views of application system, connection numbers of database, the cache data size and cache time after-flushed. *P* and *N* represent the average of *p* and *n* in a certain period of time, and *S* and *T* show the appropriate cache size and cache time respectively. In different environment, *S* and *T* may vary greatly, because they are affected by system factors and data conditions. For example, the memory size and CPU performance may have great influences on them. In addition, data size and the frequency of updates may affect them a lot, too. Generally, the value of *S* and *T* are obtained by two ways: manual setting and calculated by experiments. For manual setting, it is the administrator that set them through their experience and knowledge. This is easy but not very reliable. In contract, experiment calculation is complex but very reliable. In this paper, *S* and *T* is set by manual way. The parameters k_1, k_2, k_3 are used to adjust factor weights and the range of θ.

Through comparing θ with Ω (the threshold set in advance), cache controller determines whether data need to be synchronized.

5 Experiments

In order to exclude the impact of the network and node efficiency, experiments are performed on a six-core server (X-eon(R) CPU E7-4807 @1.87GHz ×4, 8G RAM) rather than distributed environment. MongoDB and Oracle are used to test the performance of CSFUD. More than 5 million records are saved in database. The server runs Linux RedHat 64-bit. Five groups of tests were used to verify the performance of CSFUD. Firstly, data updating rate was compared with different number of indexes. Then the update rate and performance of UMM were compared. At last, the influence of CSFUD in querying data and its performance in application were tested.

5.1 Influence of Indexes in Data Updating

In order to display the influence of data indexes on data updating, date updating speed was tested under different number of indexes. The result is shown in Fig. 7. About 10000 records were updated in batches at the same time which was executed in MongoDB. According to the test, the update time slows nearly 1 second if an index was added to the data. Additionally, the index for different type data influences update speed differently. *String* is usually the worst data type which needs more time than other types to maintain its indexes.

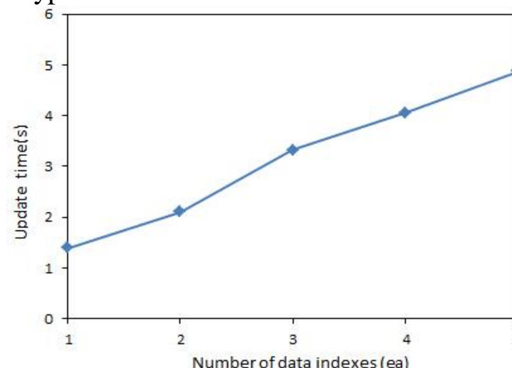

Fig. 7 Influence of data indexes in updating data.

5.2 Performance of CSFUD

The update performance of CSFUD is shown in Fig. 8 and Fig. 9. In this experiment, 4 ways (database without helper method, auxiliary index method, asynchronous index method, and CSFUD) of updating data were compared together and five indexes were created for the data. Data updating of these three types were distributed evenly. The experimental result by MongoDB is shown in Fig. 8. For a rational database, oracle was tested whose result is given in Fig. 9.

From Fig. 8 and Fig. 9, it is obvious that all of these three methods can speed the data updating, but their performances vary a lot. It is mainly caused by the strategy they adopted. For auxiliary index method, it is less effective compared with asynchronous method and CSFUD. The reason may be that it can hardly reduce the cost of the superposition influence in maintaining many indexes, although it can speed index updating singly. Moreover, five indexes were created in the experiment. For asynchronous index method, index maintaining is delayed, which can help to reduce the index maintain influence sharply. CSFUD has great advantage to other ways and its advantage becomes larger when the updating data increase. It can save about 80% time when 20000 need to be updated.

That is mainly because CSFUD stores the updated data to cache instead of database and the time consumption is mainly on the querying the update criteria. It is known that data query is much faster that update. Besides, the CSFUD performance is better in MongoDB than Oracle since MongoDB is quicker in processing *Update* and *Delete* type update.

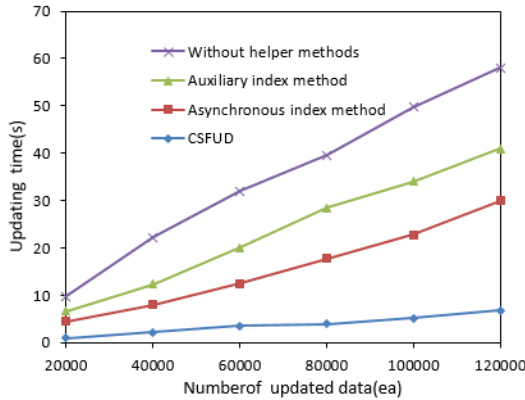

Fig. 8 Update performance of cache system for frequently updated data in MongoDB.

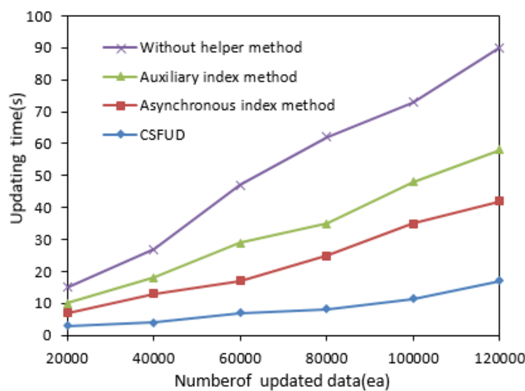

Fig. 9 Update performance of cache system for frequently updated data in Oracle.

From section 3, UMM merges update operations to the same record, which reduces the total update times towards the database. Therefore, the merging performance of UMM were tested. Food traceability data were utilized in the test, which include much location information and data that were frequently updated and queried. The result is shown in Fig. 10. When the number of updated data is less than 10000, the performance of UMM is not apparent, but as the updated data number becomes larger, especially the number of update data more than 12000, the advantage of CSFUD becomes obvious and about 3000 times update operations are merged.

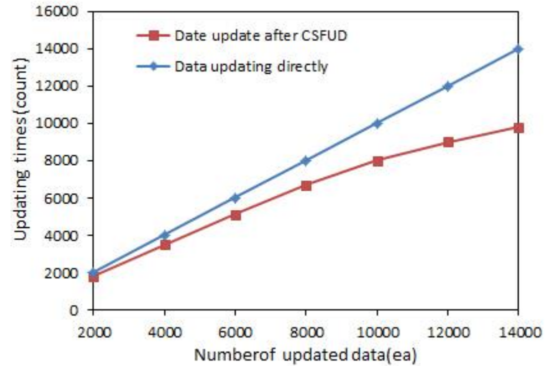

Fig. 10 Merging performance of update merging.

As data queries in CSFUD need to correct the results from database by the cache data, it is apparent that the correction process affects data query speed. Therefore, query speed has been tested and the results are shown in Fig. 11 and Fig. 12. Fig.11 is the test result from MongoDB, while Fig. 12 is from Oracle.

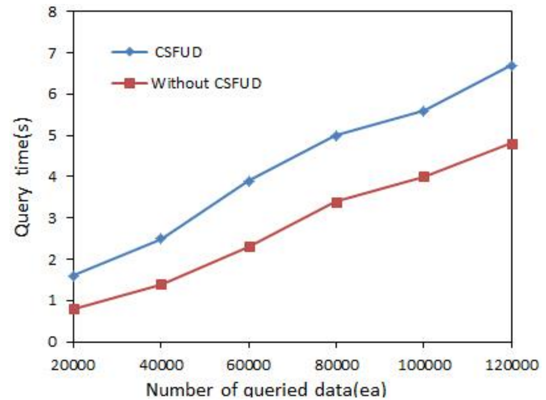

Fig. 11 Query performance of cache model for frequently updated data in MongoDB.

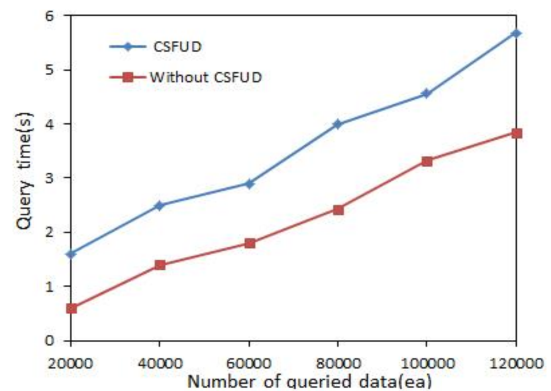

Fig. 12 Query performance of cache model for frequently updated data in Oracle.

Data query becomes slower with CSFUD. By the test result, it needs about 1 second every 10000 query requests are responded. The influence of document-oriented database and rational database is similar. Query speed is decreased with CSFUD. The

influence becomes larger with the number of queried data increased. But it is still very fast as the excellent performance of MongoDB.

Fig. 13 shows the performance of CSFUD for application system. In the experiment, a system for food traceability was used in which the data are read and write frequently. According to the test before, the system bottleneck is mainly on data writing speed. In the system, average response time of system accessing was tested and the experimental result is shown in Fig. 13.

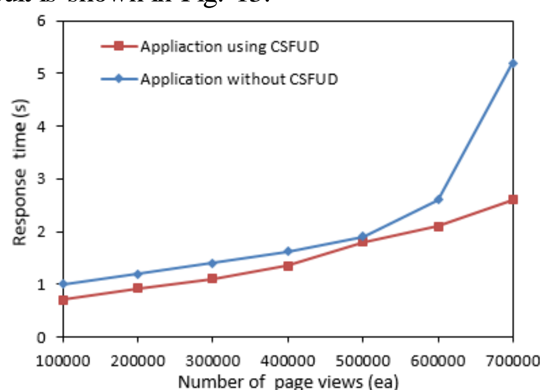

Fig. 13 The performance of cache system for frequently updated data in application.

From the experiment, CSFUD contributes to accelerating system respond in the environment under 500000 page views. It can save about 0.3 second when users interact with the system. Particularly, when the page views number reach 500000 in the system, it meets the bottleneck of data accessing. After using CSFUD, the system bottleneck was lightened.

6 Conclusions and Future Work

CSFUD provides a cache strategy in solving the index and cache maintenance problem. It caches the data by key-value pairs, provides a data merging method and synchronizes cache data in batch at idle time. In this way, the burden of database is lightened greatly. For data queries, CSFUD corrects the database query results by caching data to ensure the accuracy of queries.

Future work is targeted in two directions to complete and improve the current proposal. The first target is adopting automatic way to get the value *S* and *T* in (1) and they may be more reliable by this way. Secondly, the system needs to be improved to support transaction management.

Acknowledgements

This work was supported by the Doctoral Fund of University of Jinan (XBS1237), the Shandong Provincial Natural Science Foundation (ZR2014FQ029), and the National Natural Science Foundation of China (61173078).

References:

[1] Z. Lin, Y. Lai, C. Lin, Y. Xie, and Q. Zhou, “Research on cloud databases,” *Journal of Software*, Vol. 23, No. 05, 2012, pp. 1148–1166.

[2] Y. Shi and X. Meng, “A survey of query techniques in cloud data management systems,” *Chinese Journal of Computers*, Vol. 36, No. 02, 2013, pp. 209–225.

[3] Kun Ma and Bo Yang, “Introducing Extreme Data Storage Middleware of Schema-free Document Stores using MapReduce,” *International Journal of Ad Hoc and Ubiquitous Computing*, online, 2014.

[4] Kun Ma, Bo Yang, and Ajith Abraham, “Toward Full-text Searching Middleware over Hierarchical Documents, ” *Proceedings of the thirteenth International Conference on Intelligent Systems Design and Applications (ISDA 2013)*, Serdang, Malaysia, December 8-10, 2013, pp.194-198.

[5] Sidlauskas, Darius, et al. "Thread-level parallel indexing of update intensive moving-object workloads," *Advances in Spatial and Temporal Databases*. Springer Berlin Heidelberg, 2011, pp. 186-204.

[6] Lietsalmi, Mikko, Jaakko Vanttila, and Seppo Alanara. "Mobile station and network having hierarchical index for cell broadcast service." U.S. Patent No. 6,201,974. 13 Mar. 2001.

[7] Kyle Banker: *MongoDB in Action*. Manning Publications Co., CT. 2011.

[8] MC Brown: *Getting Started with CloudDB*. O' Relly Media Inc. , MA. 2012.

[9] Xiang, Peng, Ruichun Hou, and Zhiming Zhou. "Cache and consistency in NoSQL," *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*. Vol. 6. IEEE, 2010.

[10] P. Gupta, N. Zeldovich, and S. Madden, “A trigger-Based middleware cache for ORMs,” *Middleware 2011*, Springer Berlin Heidelberg, 2011, pp. 329–349.

[11] D. Sun, X. Hao, and Z. Hao, “Indexing method of moving objects with frequent update,” *Computer Engineering*, Vol. 39, No. 11, 2013 pp. 52–56.

[12] J. Pan, T. Ma, and J. Liu, “Research on moving objects' index in the update-intensive environments,” *Journal of Wuhan University of*

Technology, Vol. 32, No. 16, 2010, pp. 164–168+176.

[13] MoonBae Song, “Managing frequent updates in R-Trees for update-intensive applications,” *Transactions on Knowledge and Data Engineering*, Vol. 21, No. 11, 2009, pp. 1573–1589.

[14] Bahle, Dirk, Hugh E. Williams, and Justin Zobel. "Efficient phrase querying with an auxiliary index." *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2002.

[15] Per-Ake Larson, Jingren Zhou. "Asynchronous database index maintenance," U.S. Patent No. 8,140,495. 20 Mar. 2012.

[16] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylonen, “Concurrency control in B-trees with batch updates,” *IEEE Trans. Knowl. Data Eng*., Vol. 8, No. 6, 1996, pp. 975–984.