Distributed Metadata Search for the Cloud

Yang Yu, Yongqing Zhu, and Juniarto Samsudin
Data Storage Institute, A*STAR, Singapore 138632
Email: {yu_yang, zhu_yongqing, Juniarto_samsudin}@dsi.a-star.edu.sg

***Abstract*—**The ever increasing amounts of digital data being stored in public and private clouds are challenging users to access and manage the data. With the corresponding storage system reaches Petabyte-scale, or even Exabyte-scale, metadata access will become a severe performance bottleneck. Hence, this paper proposes an efficient multi-dimensional metadata index and search solution for cloud data. By proposing several new mechanisms for K-D-B tree based index/search, including two-level space borrowing and signature files for point pages, and implementing index partitioning technique, our system can achieve optimized performance in terms of memory utilization and search speed. Experiments show that our system performs much better as compared with other state-of-art solutions. In addition, our system can safely scale out in a distributed manner with guaranteed performance.

***Index Terms*—**Metadata, index, search, multi-dimension, cloud

I. Introduction

Cloud has become more prevalent in recent years. As a computing utility, it allows users to remotely store their data into the cloud to enjoy the on-demand high quality applications and services from a shared pool of configurable computing resources [1]-[3]. With the increasing amounts of digital data being stored in public and private cloud, it becomes challenging for both users and system administrators to manage and access the data. When the storage system reaches Petabyte-scale, or even Exabyte-scale, locating, retrieving and managing files has becoming extremely difficult.

Usually, users need to retrieve files with particular characteristics in the cloud datasets, and system administrators need to understand the nature of the stored data for management purpose. File metadata, such as inode information (owner, file size, file extension, file path, etc.) and other extended attributes, such as user-tagged information (language, description, subject, etc.) contain essential information to address these requirements [4]. And by properly utilizing metadata, which describe the contents and contexts of data files, the usefulness of the original data/files can be greatly increased. Hence, an efficient search solution over file metadata, which can quickly and accurately answer queries about characteristics and properties of the cloud data being stored, is awfully desirable.

In the new generation of file systems, metadata accesses will become a severe performance bottleneck as metadata-based transactions not only account for over 50% of all file system operations but also result in billions of pieces of metadata in system [5]. Unfortunately, current file systems are unable to support for large-scale metadata search. This is because today's metadata designs still resemble those over forty years ago, when file systems contained much fewer files and basic namespace navigation was more than sufficient [6]. For cloud-scale dataset, metadata search may require brute-force traversal of namespace, which is obviously impractical.

Many previous researches about file/data search focus on content-based search and keywords search [7]-[10], which cannot address the metadata search challenges presented in this paper. Some related works [11], [12] relies on Relational Database Management Systems (RDMSs) to organize and index metadata. However, Spyglass [4] shows that traditional DBMSs are far from satisfaction: the search performance gap between Spyglass and conventional DBMS solution can be as large as 3 orders of magnitude.

In this paper, we propose an efficient metadata index and search solution to address the search challenge. The proposed system contains the following components: 1) A scalable system organized in a distributed manner with each server partially serving index and search assignments. The system can easily scale out by balancing the workloads of each server and by adding more servers as the data size increases. 2) An index space partition mechanism to partition and group the indexed metadata and each partition is indexed by a separated multi-dimensional tree. This index partitioning mechanism utilizes the spatial locality of metadata attribute value distribution to control the metadata indexed in each partition. As a result, it can effectively reduce search space and improve search speed. 3) K-D-B trees are utilized to index each paritition. In this system, we propose two new mechanisms, including two-level space borrowing and signature files for point pages, to optimize K-D-B trees. With our newly proposed mechanisms, the efficiency of K-D-B tree based index and search in terms of memory utilization and search speed can be greatly improved.

The reminder of this paper is organized as follows: Section II reviews and discusses some related works. Section III presents the details of our solution. The performance of our proposed system is evaluated in Section IV. Finally, a briefly conclusion for this paper is drawn in Section V.

Manuscript received October 26, 2015; revised December 28, 2015.
doi:10.12720/jcm.11.1.100-107

II. Related Works

Many research woks in the area of file system search focuses on content-based file search and retrieval [7]-[10]. In [8], a new file system named Hierarchy and Content (HAC) is proposed. It allows the use of the file system as a regular hierarchical file system and adds content-based access under the control of users. Reference [10] propose a file system search tool that combines traditional content-based search with context information gathered from user activity, which can identify temporal relationships between files. By utilizing such context information, the search precision can be improved.

For cloud data search, many people works on the keyword based search [3], [13]-[15]. One of the key concerns to store data in a cloud is to protect the data privacy. Hence, many paper works on how to provide efficient key-word search over encrypted cloud data. In [3], [13], the author propose a secure ranked keyword search, which greatly enhances system usability by returning matching files in a ranked order regarding to the keywords. Reference [14] proposes a privacy-aware fuzzy multi-keyword search solution, which is more practical than search based on exact keywords match. The authors of [15] provide a solution to enable multiple users to perform private keyword based search over encrypted data. This enhances many existing solutions, which only support single-user keyword search in the cloud. A so-called A-Tree solution is proposed in [16] to index multi-dimensional data for cloud environments. It utilizes a master-slave distributed architecture to handle user queries: the master nodes receive user queries and locate the corresponding slave node(s) to handle the queries, and the slave nodes process user queries locally. On each slave node, it combines R-tree [17] and Bloom filters [18] to achieve efficient point query and range query for multidimensional data.

Similar solutions of our work include Spyglass [4] and SmartStore [5]. Spyglass uses an index versioning mechanism and batch update to minimize the interruption of index update over search. The index partitioning technique is also efficiently utilized by Spyglass to narrow down search space and guarantee search speed. In addition, Spyglass utilizes K-D tree [19] to index and search partitioned metadata. However, a K-D tree is poor for frequent updates [4] and our experiments show that its search performance is much worse than that of a corresponding K-D-B tree [20], [21]. SmartStore leverages the semantics of metadata attributes to cluster and partition the indexed metadata. It utilizes R-tree for metadata index and search. An R-tree is good for multi-attribute index; however the indexed pages likely become highly overlapped as time elapses. Consequently its search efficiency may degrade with time.

Although a cloud system may be distributed over wide areas, at the backend of its data centers, the file system efficiency will still be crucial to the whole cloud. Hence, in our current work, we address the metadata search challenges for large-scale file systems within a data center. Different from Spyglass and SmartStore, our system mainly focuses on improving system performance in terms of memory utilization and search speed by optimizing K-D-B tree based index and search.

III. Multi-Dimension Metadata Index and Search System

A. System Overview

In this paper, we aim to provide an efficient multi-dimensional metadata index and search system with good scalability for cloud-scale data. The system architecture is shown in Fig. 1. The system is organized in a distributed manner: the indexed metadata are distributed among multiple servers. They can be stored directly on disks or through various databases. For example, the NoSQL database, MongoDB [22], can be deployed to store the metadata. By utilizing MongoDB sharding process, metadata can be easily distributed across multiple servers. Accordingly, the indexer/searcher on each server handles partial index update and query tasks of the whole system. The coordinator will decide which server an index update or a user query should be forwarded to. With such a distributed architecture, our system can naturally scale out as data size grows.

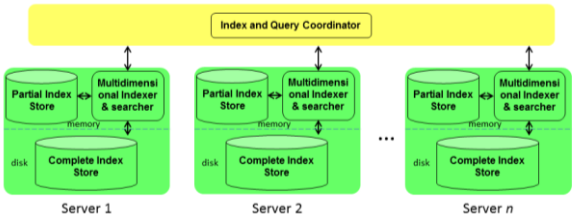


Fig. 1. System architecture

The metadata handled by each server could be still very large, and thus the indexed metadata on each server will be further partitioned into multiple pieces. Each piece is served by a separate K-D-B tree. And this is called index partitioning. In our system, the index partitioning utilizes the skewness distribution of file system metadata attribute values [4] to partition the whole index space. As a result, the metadata indexed in the same partition share more similar or even common attribute values as compared to those in different partitions. By deploying the index partitioning, the search space can be effectively reduced because the results, which can satisfy a user query, especially a multi-dimension query, are normally located in just a few partitions, and accordingly the search speed can be improved.

For each partition, its complete index is stored on disk, while a copy is kept in memory except the actual indexed metadata, which is normally too large to be kept in memory. In other words, the on-disk index store is a complete K-D-B tree. And the in-memory index store is a partial K-D-B tree without the indexed metadata. The

rationale of keeping the index/search trees in memory is to facilitate index updates and user queries by minimizing disk accesses.

The index update process, including insertion, deletion and update is described here: when an index update request arrives at the system, the coordinator will decide which server this request should be channeled to by comparing the metadata attribute values of this index request with the attribute value ranges of the metadata served by every individual server. When the index update reaches a server, it will be forwarded to some partition(s) by investigating the metadata attribute values served by each partition. A metadata update can be considered as a deletion of the old metadata plus an insertion of the new metadata. After decision, the index update will store the new metadata and/or delete the old metadata from the corresponding index store and update the index tree if necessary.

The query process is somewhat similar: when a user query arrives, the coordinator will decide which server(s) this request should go to. For each server with possible matching results, the query is further compared with the metadata served by each individual partition to decide which partition(s) need to be searched and which partition(s) can be skipped. Then each partition tree with possible matching results is traversed. All possible results which can satisfy the query request will be returned to the coordinator from multiple partition trees and even multiple servers. The search coordinator will do filtering and merge and then pass the final results to the user.

The system deploys the following mechanisms in order to achieve satisfying performance: 1) Optimization mechanisms for K-D-B trees. K-D-B trees, which are used to index every partition, can be considered as the basic element for our solution and hence its performance is crucial to the whole system. Two optimization mechanism for K-D-B tree based index and search, including two-level space borrowing and signature files for point pages, are proposed and implemented to reduce its space overhead and improve its search efficiency, and consequently guarantee the overall system performance in the fundamental level. 2) Index partitioning decomposes the index into separate partitions based on the skewness distribution of indexed metadata. Each partition is deployed with a K-D-B tree, and the metadata with similar or close attribute values are indexed by the same tree. This mechanism allows the indexed metadata to be managed and searched at the granularity of sub-trees [4], which is critical to provide scalability and efficiency as the system grows.

B. Index and Search with Optimized K-D-B Tree

1) Overview of K-D-B tree

A K-D-B tree is a k-dimensional B-tree data structure to subdivide a k-dimensional space. And it provides point, range and nearest-neighbor search over the indexed space. A K-D-B tree normally contains two types of pages [20]:

- Region page (non-leaf page): a collection of (region, child) pairs, in which a region describes a boundary, and a child is the pointer to the child page corresponding to the region.
- Point page (leaf page): a collection of (point, location) pairs, in which a point is the identifier of the indexed data and a location represents the point's coordinates in the space.
- And an example 2-D-B tree is illustrated in Fig. 2.

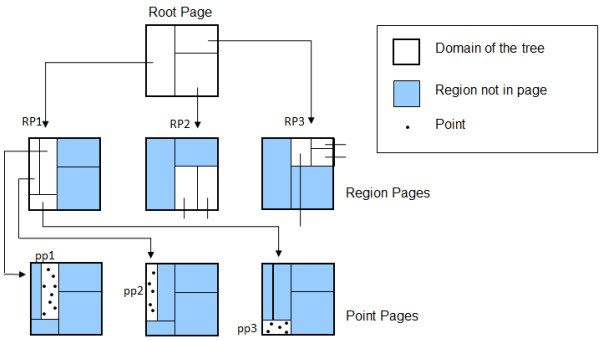


Fig. 2. An example 2-D-B tree

We deploy K-D-B tree in our system to serve index and search for every partition because it can provide fast and efficient multi-dimensional search over all metadata of a partition. Each metadata attribute is represented by a unique dimension in the K-D-B tree.

2) Two-level space borrowing

Updates in an original K-D-B-tree may result in the splitting of several nodes of the tree recursively. This is because while a point page is always split in a way that maintains a balance between the resulting pages, a region page is split along the same point on the same partition axis [20]. This forced-splitting (FS-splitting) policy is incredibly inefficient and can result in poor memory utilization. Several improvements on splitting are derived, among which we believe the first-division splitting (FD-splitting) [23] is promising as it significantly reduces forced splits and recursive tree updates. Hence we adopt K-D-B tree with FD-splitting policy to index metadata of every partition.

Although a K-D-B tree with FD-splitting policy has significantly reduced unnecessary splits and tree structure updates, its memory utilization is still far from optimization. This is because for a conventional K-D-B tree, if a point page gets overflowed due to a data point insertion, the page is always required to be split to two. Such a splitting rule may propagate the split to upper levels of the tree and create many new pages with small number of children. Hence, in our system, we propose a two-level space borrowing mechanism. This mechanism can make the number of points contained in each point page approach its upper limit as much as possible. Consequently, the page splits due to data insertion can be minimized and the memory utilization can be further improved.

Fig. 3 gives the comparison of tree structure updates between a conventional K-D-B tree and a K-D-B tree with our proposed space borrowing mechanism due to a data insertion. Suppose the page limit for this example 2-D-B tree is 3 for a region page and 8 for a point page respectively.

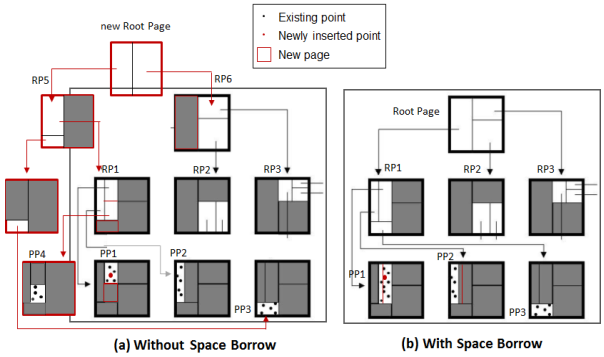


Fig. 3. Comparison of tree updates for a convetional K-D-B tree and a K-D-B tree with two-level space borrowing mechanism due to a data inertion

Fig. 3(a) illustrates how the tree structure is updated for a conventional K-D-B tree: A data point insertion into *PP*1 makes *PP*1 overflowed. The overflow of *PP*1 will make *PP*1 split into two point pages. Such a split will in turn make *RP*1, *PP*1's parent page, get overflowed. Hence, *RP*1 will be also split into two pages accordingly. The split of *RP*1 then makes the root page become overflowed and get split. Consequently, a new root page is created. From this example, we can see that the tree structure is completely affected by such a single data insertion and 4 new pages, including 1 point page and 3 region pages, are created to cater for this change.

Fig. 3(b) shows how our space borrowing mechanism deals with such a data insertion: As *PP*1 gets overflowed due to a new data insertion, our space borrowing mechanism checks if *PP*1's neighbors, including its 1-level neighbor *PP*2 and 2-level neighbor(s) *PP*3, are full. If some of the neighbor is not full, the overflowed *PP*1 can 'borrow space' from the neighbor page by shifting the corresponding split axis. In this example, *PP*2 only contains 5 data points, which means it has space to hold more data. As illustrated in Fig. 3(b), our space borrowing mechanism will then shift the split axis between *PP*1 and *PP*2, which splits *PP*1 and *PP*2, so that some data points will be allocated from *PP*1 to *PP*2. As a result, both *PP*1 and *PP*2 are kept non-overflowed and contain similar number of data points. By utilizing our two-level space borrowing mechanism, such a data insertion operation only affects two point pages and the tree structure remain completely unchanged. As a result, the memory utilization is improved as point pages can hold more nodes and the creation of unnecessary new pages can be avoided.

The detailed two-level space borrowing mechanism is described in the flowchart as shown in Fig. 4. In this mechanism, 1-level neighbor refers to the neighboring page of a point page, which shares the same parent and exactly the same split axis sequence with that point page. And 2-level neighbors refer to the neighboring page(s) of a point page, which share the same parent and the same split axis sequence except the last split with that point page. We restrict the space borrowing to be occurred within 2-level neighbors because space borrowing only from 1-level neighbor may not always be able to satisfy the requirement. However, borrowing space from too far (3-level or more) neighbors may affect too many pages and make split axis shift very complicated.

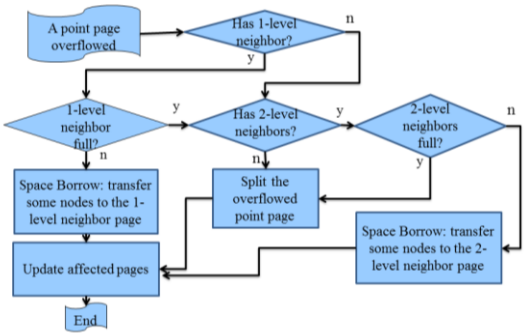


Fig. 4. Flowchart illustration of two-level space borrowing mechanism

3) *Signature files for point pages*

As mentioned before, normally the indexed metadata in a partition are too large to be kept in memory. Hence, the in-memory index tree alone cannot complete an index update or user query task. It needs to access on-disk index store to retrieve all the point pages, which store the actual indexed metadata that may satisfy the corresponding index update or user query. Disk access is more costly as compared to in-memory search, and hence it is always desired to reduce the disk access as much as possible.

In our system, we propose to utilize signature files [24] to describe the attribute values of each metadata stored on a point page. Hence, when an index update or a user query needs to access a point page on disk to retrieve the indexed metadata, it can first check the in-memory signature files of this point page. By checking the signature files, the system can decide whether actual metadata that can satisfy the index update/user query are contained in the point page or not. If no such matches, the point page retrieval from disk can be skipped. In such a way, the disk accesses for index and search can be minimized.

As we deploy signature files at the point page level, they represent the compact summaries of a point page's contents. In other words, for each point page, multiple signature files are utilized, and each signature file presents all the values of a metadata attribute indexed in this point page. A signature file, or just a signature for simplicity, is a bit-array with an associated hashing function. All bits in the signature are initially set to zero. When a metadata value is inserted, it is hashed to a bit-position, modulo the size of the signature, which is then set to one [24]. For example, supposing a 3-dimension metadata (**owner** = *tom* & **size** = *11MB* & **ext** = *pdf*) is

inserted into an empty point page, the corresponding signature files will be updated as illustrated in Fig. 5. Consequently, a user query will only retrieve a point page from disk if signature bits for all predicates in the query are set to one [4].

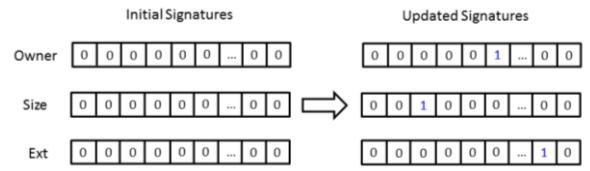


Fig. 5. Example signature files

Spyglass [4] also utilizes signature files, but it implements the signatures for each partition. This means the signature files are implemented at the root page level of each partition tree in Spyglass. Normally, each partition contains hundreds of thousands of files. Although the distribution of metadata attribute values shows spatial locality, the number of different attribute values indexed in a partition could still be very large. Then how to choose an appropriate bit array size for signature files in Spyglass will be a dilemma: a too small bit array size will cause serious false-positives due to hashing collision [4] and consequently degrade the search performance; a too large bit array size will make the signature file too large. The sequential scan or other operations on the signature files to decide whether the corresponding partition needs to be searched or not will become time consuming. As a result, the search performance is adversely affected.

In our system, the signature files are deployed at the point page level in our system, and the number of data contained in each point page is much smaller. So we can easily choose a proper bit array size, a size close to the number of different metadata values indexed in the corresponding point page, for each signature. Thus, the common 'false-positive' problem, which greatly affects the efficiency of signature files, can be eliminated. As a result, we can make full use of signature files in minimizing costly disk accesses.

C. Index Partitioning

In our system, the metadata to be indexed are partitioned into multiple pieces, and each piece is indexed by a separate K-D-B tree. Generally, the metadata of files belonging to the same directory or sharing similar file paths are grouped to the same partition piece. Such a partition rule utilizes the skewness distribution of metadata attribute values to effectively reduce the number of partitions need to be searched for answering user queries.

It is observed that for a hierarchical file system namespace, the distribution of its metadata attribute values shows spatial locality [4]. In other words, similar or close attribute values tend to cluster under a few namespace directories. For example, files with the **ext** value *html* are likely to reside under directories related to web pages, and files with the **owner** value *Jane* tend to reside in the directory */home/Jane*.

For the namespace hierarchy illustrated in Fig. 6, it is split into 4 index partitions, and files in the same or close directories are grouped into the same partition. Then the metadata of each partition is indexed by a separate K-D-B tree. Files in the same partition likely share some common metadata attribute values. For example, for partition 4, all files have the same owner value hduser. If a user issues a query with '**owner** = *tom*', our system can quickly conclude that partition 4 will not satisfy the query. Hence, the traverse of sub-tree for partition 4 is skipped and in turn the search space for this query request is effectively reduced.

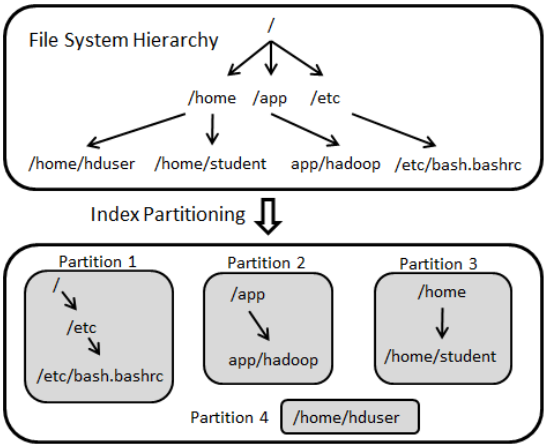


Fig. 6. Index space partitioning example

TABLE I: EXAMPLE SEARCH TABLE FOR PARTITIONS

Index Attribute	Partition	Value Range
Owner	1	[jack, perry]
	2	[bob, zoe]
	3	[green, student]
	4	[hduser]
File Extension	1	[alias, d]
	2	[lock, txt]
	3	[avi, c++]
	4	[bmp, tiff]
File Size	1	[10KB, 500KB]
	2	[100KB, 2GB]
	3	[5KB, 100MB]
	4	[32KB, 1MB]
…	…	…

In our system, we introduce a table to record the value ranges of each metadata attribute indexed by each partition as shown in Table I. Because the distribution of metadata attribute values shows spatial locality for different partitions, such a table can effectively present the different metadata attribute value ranges indexed in each partition. When the system receives an index update or a user query, it first checks this table to decide which partition(s) need to be searched and which partition(s) can be skipped. Then it traverses the corresponding partition trees with possible matches to obtain the final results. Due to the skewed distribution of metadata attribute values of different partitions, checking this table

can effectively narrow down the search space, and in turn the search speed can be significantly improved.

IV. Experimental Evaluation

We focus the performance evaluation on two aspects: 1) How much the memory utilization can be improved by deploying our two-level space borrow mechanism for K-D-B index trees. 2) How fast our system can provide search results to user queries by utilizing index partitioning and signature files of point pages for K-D-B tree based search. The K-D-B tree implemented in our system is a K-D-B tree with our proposed optimization techniques, and hence we name it as the optimized K-D-B tree. For the first evaluation, we compare the space utilization of a conventional K-D-B tree and the optimized K-D-B tree. For the second evaluation, we compare the search performance of a single optimized K-D-B tree with that of a single K-D based search tree because the K-D tree is also a popular choice for multi-dimension indexing. In addition, to evaluate the effectiveness of index partitioning in narrowing down search space and guaranteeing search speed, we study the search performance of a system indexed by a single optimized K-D-B tree and that of a larger system indexed by multiple optimized K-D-B trees implemented with index partitioning.

Table II shows the data size that we used for experimental evaluation in this paper. The maximum data size that we tested is 9-dimension metadata for 1M files, which is about 320MB. For metadata with such a size, they can be stored in a single node and safely kept in memory. Hence, in this paper, a complete copy of the index tree is kept in memory. And the index and search processes are completely performed in memory without disk access.

Table II. Dataset Size Used for Experiments

	Memory Utilization	Search Performance of K-D-B tree vs. K-D tree	Search Performance of K-D-B trees with index partitioning
No. of files	50K	[10K, 50K, 100K]	1M
Metadata Size	16MB	[3.2MB, 16MB, 32MB]	320MB

As part of our future work, we will increase the tested data size so that the indexed metadata, i.e. partial information of the point pages of index trees, will be stored on disk. Hence, both index updates and user queries will involve disk access.

Fig. 7 compares the number of region pages generated by a conventional K-D-B tree and that generated by our optimized K-D-B tree by varying the index batch update size. The indexed metadata size in this experiment is about 16MB, corresponding to 50K files. It can be seen that on average the number of region pages in our optimized K-D-B tree can be reduced by 28% as compared with that in a conventional K-D-B tree. Since region pages of an index/search tree always needs to be kept in memory for fast index update and query, our proposed two-level space borrowing mechanism effectively improves the memory utilization by generating fewer number of region pages.

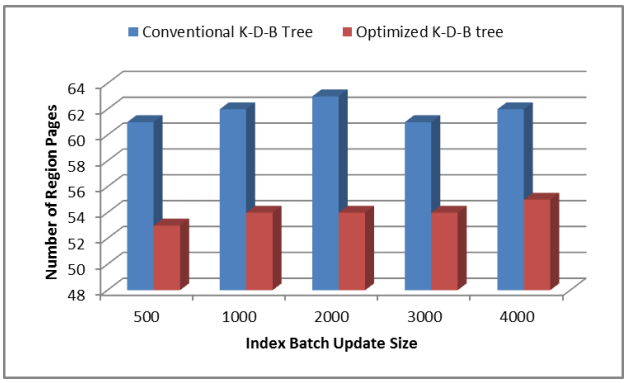


Fig. 7. Number of region pages vs. Index batch update size

Fig. 8 compares the number of point pages of a conventional K-D-B tree with that of our optimized K-D-B tree by varying the index batch update size. Similar to the results shown in Fig. 7, our optimized K-D-B tree on average can generate 10% fewer point pages as compared to the conventional K-D-B tree. This is because the two-level space borrowing mechanism effectively makes each point page hold more data points by reducing point page splits. Another observation from Fig. 8 is that the number of point pages generated by the conventional K-D-B tree is kept almost the same for different batch update size. However, the generated point pages of our optimized K-D-B tree increases slowly with the batch update size. This is because with larger batch size, the chance that neighboring pages concurrently get overflowed during the same batch update increases. Thus the chances of borrowing space from neighbors are reduced, and this results in increased point page number.

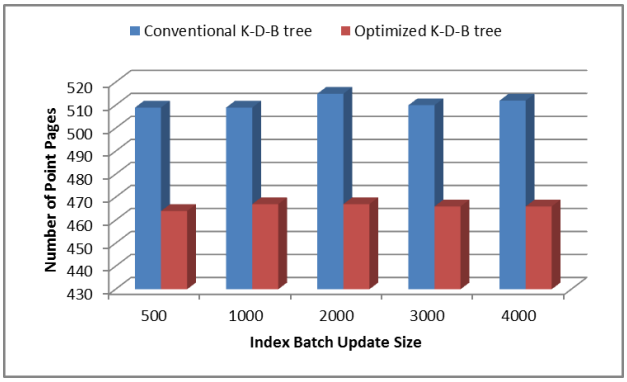


Fig. 8. Number of point pages vs. Index batch update size

Fig. 9 shows the search performance of a K-D tree vs. our optimized K-D-B tree by varying dataset size and the number of queries issued simultaneously. It can be seen that the optimized K-D-B tree performs hundreds of times better than the K-D tree search. This is mainly because the K-D tree is a binary tree and each single data point forms a point page, which makes the number of

pages and the tree depth in a K-D tree are much larger than those of a corresponding K-D-B tree. Hence, the tree traversing to reach point pages and get the final results consumes much more time. Of course, our proposed optimization mechanisms for the K-D-B tree also bring benefits to its search.

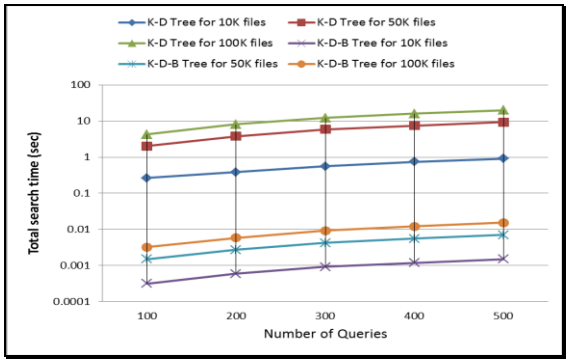


Fig. 9. Seach performance comparison between a K-D tree and our optimized K-D-B tree

Table III below gives the comparison of page number and tree depth for a K-D tree and a K-D-B tree, both indexing 100K file's 9-dimension metadata. For a K-D tree, each node can at most have 2 child nodes and each point page only contains a single data point because it is a binary tree. On the contrary, a K-D-B tree is a b+ tree, and hence each tree node can have multiple child nodes. In our experiment, the child number limit for a region page and a point page is set to 16 and 150 respectively for the K-D-B tree. From the results, we can see that the number of pages and the tree depth are much smaller for a K-D-B tree as compared to those of a K-D tree. These numbers give solid evidence for the significant performance gap between the K-D tree search and the K-D-B tree based search as shown in Fig. 9.

TABLE III. PAGE NUMBER AND TREE DEPTH COMPARISON FOR K-D TREE AND K-D-B TREE

	Region Page Number	Point Page Number	Tree Depth
K-D Tree	102360	100000	48
K-D-B Tree	56	753	3

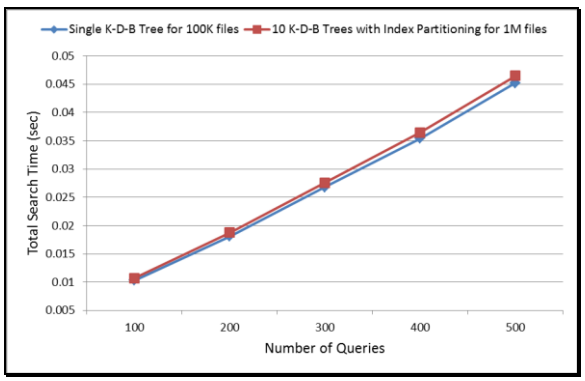


Fig. 10 Search performance evaluation for multiple K-D-B tree with index partitioning

Fig. 10 shows the search performance of a K-D-B tree indexing 100K files' metadata and that of 10 K-D-B trees deployed with index partitioning indexing the metadata for 1M files in total. It can be seen that these two experiments show very similar results, which means the index partitioning technique can successfully reduce search space and guarantee the search performance. Hence, when the data size grows, we can partition the corresponding metadata into multiple pieces by utilizing the spatial locality distribution of metadata attribute values. And the corresponding system can safely scale out as the search speed is maximally guaranteed.

V. CONCLUSION

In this paper we proposed an efficient multi-dimensional metadata index and search system targeting the challenges of metadata-based transactions for cloud data. The system can safely scale out by organizing it in a distributed manner: each server stores partial indexed metadata and handles partial index updates and user queries. On each server, the indexed metadata is partitioned into multiple pieces and each partition is indexed by a separate K-D-B tree. The index partitioning depends on the skewness and spatial locality distribution of metadata attribute values and promises the metadata indexed by the same partition tree share more common or similar attribute values. Hence, the search space can be effectively reduced and the search speed can be significantly improved. Several mechanisms are proposed to optimize K-D-B trees in this system, including two-level space borrowing and signature files for point pages. Since K-D-B trees are used to serve each partition's metadata, their performance is critical to the overall system. With the proposed optimization mechanisms, the performance of a K-D-B tree in terms of memory utilization and search efficiency can be greatly improved.

Experiments are conducted to evaluate memory utilization and search performance of our implemented system. Results show that our system successfully achieves the targeted goals as compared to other state-of-art solutions. For the next step, we will increase the metadata size. So the indexed metadata needs to be stored on disks and may even need to be spread on multiple servers. In such a more challenging environment, we can then evaluate the scalability and performance of our proposed system in a more comprehensive way.

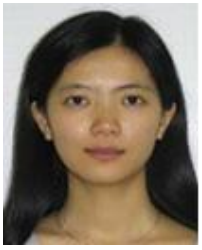
REFERENCES

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres and M. Lindner, "A break in the clouds: Towards a cloud definition," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50-55, 2009.

[2] N. Cao, S. Yu, Z. Yang, W. Lou, and Y. Hou, "LT codes-based secure and reliable cloud storage service," in *Proc. IEEE Infocom'12*, 2012, pp. 693-701.

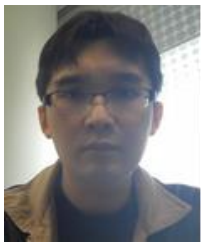
[3] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-Preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel and Distributed Systems*, vol. 25, no. 1, pp. 222-233, 2014.

[4] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proc. 7th FAST*, Feb. 2009, no. 9, pp. 153-166.
[5] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, "SmartStore: A new metadata organization paradigm with metadata semantic-awareness for next-generation file systems," in *Proc of IEEE SC'09*, Nov. 2009, pp. 14-20.
[6] A. W. Leung, I. F. Adams, and E. L. Miller, "Magellan: A searchable metadata architecture for large-scale file systems," *Technical Report UCSC-SSRC-09-07*, Nov. 2009.
[7] C. M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti, "A file system for information management," in *Proc. International Conference on Intelligent Information Management Systems*, Mar. 1994.
[8] B. Gopal and U. Maner, "Integrating content-based access mechanisms with hierarchical file systems," in *Proc. 3rd USENIX OSDI*, Feb. 1999, pp. 265-278.
[9] O. Laadan, R. A. Barratto, D. B. Phung, S. Potter, and J. Nieh, "DejaView: A personal virtual computer recorder," in *Proc. of the 21st ACM SOSP*, 2007, pp. 279-292.
[10] C. A. N. Soules and G. R. Ganger, "Connections: Using Context to Enhance File Search," in *Proc. 20th ACM SOSPs*, 2005, pp. 119-132.
[11] Apple. (2008). Spotlight Server: Stop Searching, Start Finding. [Online]. Available: http://www.apple.com/server/macosx/features/spotligt/
[12] MetaTracker. (2008). Metatracker for Linux. [Online]. Available: http://www/gnome.org/project/tracker/
[13] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, "Secure ranked keyword search over encrypted cloud data," in *Proc. of IEEE ICDCS'10*, June 2010, pp. 253-262.
[14] M. Chuah and W. Hu, "Privacy-Aware bedtree based yongqing solution for fuzzy multi-keyword search over encrypted data," in *Proc. IEEE ICDCSW'11*, June 2011, pp. 273-281.
[15] Y. Yang, H. Lu and J. Weng, "Multi-User private keyword search for cloud computing," in *Proc. IEEE CloudCom'11*, Nov.-Dec. 2011.
[16] A. Papadopoulous and D. Katasaros, "A-Tree: Distributed indexing of multidimensional data for cloud computing environments," in *Proc. IEEE CloudCom'11*, Nov.-Dec. 2011
[17] A. Guttman, "R-Trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD'84*, 1984, pp. 47-57.
[18] B. H. Bloom, "Space/Time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422-426, 1970.
[19] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509-517, 1975.
[20] J. T. Robinson, "The K-D-B Tree: A search structure for large multi-dimensional dynamic indexes," in *Proc. ACM SIGMOD*, 1981, pp. 10-18.
[21] B. Nam and A. Sussman, "A comparative study of spatial indexing techniques for multidimensional scientific datasets," in *Proc. 16th IEEE SSDBM*, 2004, pp. 171-180.
[22] MongoDB, Sharding. [Online]. Available: http://docs.mongodb.org/manual/sharding/
[23] R. Orlandic and B. Yu, "A retrieval technique for high-dimensional data and partially specified queries," *Data and Knowledge Engineering*, vol. 42, no. 1, pp. 1-21, 2002.
[24] C. Faloutsos and S. Christodoulakis, "Signature files: An access method for documents and its analytical performance evaluation," *ACM Trans. on Information Systems*, vol. 2, no. 4, pp. 267-288, 1984.



Yang Yu received her B. E. degree from Tianjin University, China, in 2000 and PhD degree from Nanyang Technological University (NTU), Singapore in 2007. After graduation, she first worked in NTU as a research engineer. Since 2009, Dr. Yu has been working with Data Storage Institute, A*STAR, Singapore as a research scientist. Her current research interest covers various data center related technologies, including data management, big data analytics, cloud storage, data center communication protocols and so on.

Yongqing Zhu received her B.E. degree from Beijing University of Post & Telecommunications in 1996 and PhD degree from Nanyang Technological University in 2006. Dr. Zhu has been working as a Research Scientist in Data Storage Institute, A*STAR, Singapore since 2006. Her current research interests include parallel and distributed systems, data management and analytics, cloud storage and cloud computing, I/O scheduling, search technologies, etc.



Juniarto Samsudin received his BSc. in Mechanical Engineering from University of Trisakti, Jakarta, Indonesia, and MSc. in smart product design from Nanyang Technological University, Singapore. He is currently a research engineer at Data Storage Institute, Singapore. His research includes Hadoop, distributed computing, and web service architecture. He spends his free time, tinkering with linux embedded system, raspberry pi, and Arduino. He is also music enthusiast, particularly with MIDI, Digital Audio Workstation (DAW).