An Indexing Framework for Efficient Retrieval on the Cloud[*]

Sai Wu
National University of Singapore
wusai@comp.nus.edu.sg

Kun-Lung Wu
IBM T. J. Watson Research Center
klwu@us.ibm.com

Abstract

The emergence of the Cloud system has simplified the deployment of large-scale distributed systems for software vendors. The Cloud system provides a simple and unified interface between vendor and user, allowing vendors to focus more on the software itself rather than the underlying framework. Existing Cloud systems seek to improve performance by increasing parallelism. In this paper, we explore an alternative solution, proposing an indexing framework for the Cloud system based on the structured overlay. Our indexing framework reduces the amount of data transferred inside the Cloud and facilitates the deployment of database back-end applications.

1 Introduction

The emergence of the Cloud system has simplified the deployment of large-scale distributed systems for software vendors. The Cloud system provides a simple and unified interface between vendor and user, allowing vendors to focus more on the software itself rather than the underlying framework. Applications on the Cloud include Software as a Service system [1] and Multi-tenant databases [2]. The Cloud system dynamically allocates computational resources in response to customers' resource reservation requests and in accordance with customers' predesigned quality of service.

The Cloud system is changing the software industry, with far-reaching impact. According to an estimation from Merrill Lynch [3], by 2011, the Cloud computing market should reach $160 billion, including $95 billion in business and $65 billion in online advertising. Due to the commercial potential of the Cloud system, IT companies are increasing their investments in Cloud research. Existing Cloud infrastructures include Amazon's Elastic Computing Cloud (EC2) [4], IBM's Blue Cloud [5] and Google's MapReduce [6].

As a new computing infrastructure, the Cloud system requires further work for its functionalities to be enhanced. An area that draws most attention is data storage and retrieval. Current Cloud systems rely on underlying Distributed File Systems (DFS) to manage data. Examples include Google's GFS [8] and Hadoop's HDFS [9]. Given a query, the corresponding data are retrieved from the DFS and sent to a set of processing nodes for parallel scanning. Through parallel processing, the Cloud system can handle data intensive application efficiently. The challenges here lie in how to partition data among nodes and how to have nodes collaborate for a specific job. To simplify implementation, current proposals employ a simple query processing strategy, e.g.,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]Partially supported by Singapore Ministry of Education (T2 grant) for UTab project

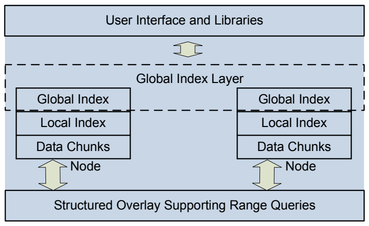


Figure 1: Indexing Framework of the Cloud

parallel scanning the whole data set. Given enough processing nodes, even the simple strategy can provide good performance. However, such an approach may only work in a dedicated system built for a specific purpose of a single organization. For example, Google employs its MapReduce [6] to compute the pagerank of web pages. In the system, nodes are dedicated to serving one organization. In contrast, in an open service Cloud system, such as Amazon's EC2, different clients deploy their own software products in the same Cloud system. Processing nodes are shared among the clients. Data management becomes more complicated. Therefore, instead of scanning, a more efficient data access service is required.

Following this direction, Aguilera et al.[7] proposed a fault-tolerant and scalable distributed B-tree for the Cloud system. In their approach, nodes are classified into clients and servers. The client lazily replicates all inner B^+-tree nodes, and the servers synchronously maintain a B^+-tree version table for validation. This scheme incurs high memory overhead for the client machine by replicating the inner nodes across the clients. Moreover, it is not scalable when the updates follow skewed distribution, invoking more splitting and merging on the inner nodes. In this paper, we examine the requirements for the Cloud systems and propose an indexing framework based on our earlier work outlined in [10]. Firstly, this indexing framework supports all existing index structures. Two commonly used indexes, hash index and B^+-tree index, are employed as examples to demonstrate the effectiveness of the framework. Secondly, processing nodes are organized in a structured P2P (Peer-to-Peer) network. A portion of the local index is selected from each node and published based on the overlay protocols. Consequently, we maintain a global index layer above the structured overlay. It effectively reduces the index maintenance cost as well as the network traffic among processing nodes, resulting in dramatic query performance improvement.

The rest of the paper is organized as follows: We present our indexing framework in the next section and discuss the details of our indexing approach in Section 3. In Section 4, we focus on the adaptive indexing approach. And some other implementation and research issues are introduced in section 5. Finally, we present our preliminary experimental results in Section 6 and conclude the paper in Section 7.

2 System Architecture

Figure 1 illustrates our proposed indexing framework for the Cloud system. There are three layers in our design. In the middle layer, thousands of processing nodes are maintained in the Cloud system to provide their computational resources to users. Users' data are partitioned into some data chunks and these chunks are disseminated to different nodes based on DFS protocols. Each node builds some local index for its data. Besides the local index, each node shares parts of its storage for maintaining the global index. The global index is a set of index entries, selected from the local index and disseminated in the cluster. The middle layer needs to implement the following interfaces:

Map(v)/Map(r)	Map a value or data range into a remote node
GetLI(v)/GetLI(r)	Given a value or data range, return the corresponding local index
GetGI(v)/GetGI(r)	Given a value or data range, return the corresponding global index
InsertGI(I)	Insert an index entry into the global index

All the methods except *GetLI* rely on the *Map* function. Given a value (hash based index) or a range (B^+-tree based index), *Map* defines how to locate a processing node responsible for the value or range. Its implementation depends on the lower layer's interface.

To provide an elegant interface for users, we apply the structured overlay to organize nodes and manage the global index. In the lower layer, processing nodes are loosely connected in a structured overlay. After a new node joins the Cloud, the node performs the join protocol of the overlay. Specifically, the node will accept a few other nodes as its routing neighbors and notify others about its joining. This process is similar to the construction of a P2P network. However, our system differs significantly from the P2P network. In the Cloud system, services are administrated by the service provider, and nodes are added into the system to provide computational resources. On joining the network, nodes must remain online unless the hardware fails. In contrast, in the P2P network, peer nodes are fully autonomous and unstable. A peer joins the P2P network for its own purpose (e.g., to download files or watch videos) and leaves the network on finishing its task. In our system, the P2P overlay is adopted only for routing purposes. The interfaces exposed for the upper layers are:

lookup(v)/lookup(r)	Given a value or a range, locate the responsible node
join	Join the overlay network
leave	Leave the overlay network

In principle, any structured overlays are applicable. However, to support B^+-tree based index, range search is required. Therefore, we adopt structured overlays that support range queries, such as CAN[11] and BATON[12].

In the upper layer, we provide a data access interface to the user's applications based on the global index. The user can select different data access methods for different queries. Scanning is suitable for the analysis of large data sets while index-based access is more preferred for online queries.

3 Indexing Framework

In this section, we shall discuss the implementation issues of the middle layer in the framework. Algorithm 1 shows the general idea of the indexing scheme. First, we apply an adaptive method to select some index values (the adaptive approach will be discussed in the next section). For a specific index value *v*, we retrieve its index entry through the *GetLI* method. The index entry is a value record in the hash based index or a tree node in the B^+-tree based index. Then, we apply the *Map* function to locate a processing node and forward the index entry to the node, where it will be added to the global index. Algorithm 2 shows the query processing algorithm via the global index. The query is forwarded to the nodes returned by the *Map* function, where the query is processed through the global index in parallel. As the algorithms show, the *Map* function plays an important role in the index construction and retrieval. In this section, we discuss how to define a proper *Map* function for different types of indexes.

Algorithm 1 EstablishGlobalIndex(node n)

```
1: ValueSet S=getIndexValue()
2: for $\forall v \in S$ do
3:     I=GetLI(v)
4:     publish I to Map(v)
5: end for
```

3.1 Hash Based Indexing

The hash index is used to support exact key-match queries. Suppose we use the hash function h_l to build the local hash index. For an index value *v*, we can simply define the *Map* function as:

$$Map(v)=lookup(h_g(v))$$

Algorithm 2 SearchGlobalIndex(range r)

```
1: NodeSet N=Map(r)
2: for $\forall n \in N$ do
3:     I=n.GetGI(r)
4:     process queries based on I
5: end for
```

where h_g is a global hash function for the Cloud system and *lookup* is the basic interface of the structured overlay. In the structured overlay, for routing purpose, each node is responsible for a key space. For the hash index, all nodes apply h_g to generate a key *k* for an index value. Given a key, *lookup* returns the node responsible for the key. Note that h_g does not need to be equivalent to the hash function h_l as each node may build their local hash index based on different hash functions.

3.2 B$^+$-tree Based Indexing

The B$^+$-tree based index is built for supporting range search. In an *m*-order B$^+$-tree, all the internal nodes, except the root node, may have *d* children, where $m \leq d \leq 2m$. The leaf nodes keep the pointers to the disk blocks of the stored keys. To define the *Map* function for the B$^+$-tree index, a range is generated for each tree node. Basically, B$^+$-tree nodes can inherit a range from their parents. In Figure 2, node d is node a's third child. So its range is from the second key to the upper bound of a, namely (35,45). The range of *a* is from the lower bound of the domain to the first key of its parent. Thus, *a*'s range is (0,45). Specifically, the range of the root node is set to be the domain range.

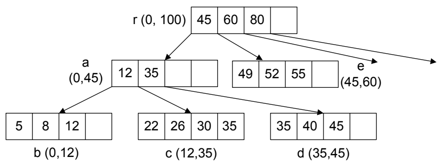


Figure 2: Node Range in B$^+$-tree

After generating the range for a B$^+$-tree node *n*, we define the *Map* function as:

Map(n)=lookup(range(n))

To support the above mapping relation, the underlying overlay must provide the *lookup* interface for a specific range. In this case, only the structured overlays that support range search are applicable, such as BATON [12], CAN [11] and P-Ring [13].

3.3 Multi-dimensional Indexing

A multi-dimensional index, such as the R-tree [14], is useful for spatial and multi-dimensional applications. In the R-tree, each node is associated with a Minimal Boundary Rectangle (MBR), which is similar to the range defined for the B$^+$-tree node. Given an R-tree node, we need to define a *Map* function to locate the processing node. Depending on the characteristics of the underlying overlays, we have two solutions:

If the underlying overlay, such as CAN [11], supports multi-dimensional routing, we can directly use its *lookup* interface. For an R-tree node n, the *Map* function is defined as:

Map(n)=lookup(getMBR(n))

However, most structured overlays have not been designed for supporting multi-dimensional data indexing. In this case, the alternative solution is to map the multi-dimensional rectangle into a set of single dimensional ranges. The space filling curve [15] is commonly used for this task. Given a rectangle R, we can define a function f based on the space filling curve, which maps R to a range set S. Finally, the *Map* function returns the corresponding node set:

$$\textit{Map(n)}=\{\textit{lookup(r)}|\forall r \in S\}$$

4 Index Tuning

The local index size is proportional to the data size. Therefore, we cannot publish all the local indexes into the global index. In this section, we discuss the index tuning problem in the framework.

Algorithm 3 IndexTuning(node n)

```
1:  IndexSet I=n.getAllIndexEntry()
2:  for $\forall e \in I$ do
3:      if needSplit(e) then
4:          IndexSet I'=getLowerLevelIndexEntry(e)
5:          remove e and insert I' into global index
6:      else
7:          if needMerge(e) then
8:              IndexEntry e'=getUpperLevelIndexEntry(e)
9:              remove e and its siblings; insert e' into global index
10:         end if
11:     end if
12: end for
```

Algorithm 3 shows the general strategy of index tuning. If an index entry needs to be split due to the high benefit for query processing, we replace the index entry with its lower level index entries. In contrast, if it needs to be merged with its siblings, we remove all the corresponding index entries and insert their upper layer entry. In this way, we dynamically expand and collapse the local index in the global index. In the above process, we manage the local index in a hierarchical manner. Existing index structures can be easily extended to support such operations. Again, we use hash index and B$^+$-tree index as the examples in our discussion.

4.1 Multi-level Hash Indexing

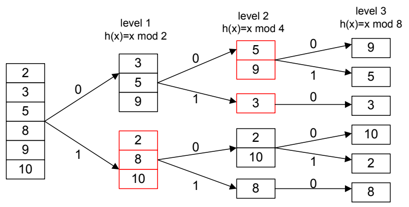


Figure 3: Hierarchical Hash Functions

Linear hashing and extendible hashing can be considered as multi-level hash functions. As shown in Figure 3, the hash function at level *i* is defined as *h(x)=x mod* 2^i. Given two data items v_1 and v_2, if $h_i(v_1) = h_i(v_2)$, v_1 and v_2 are mapped to the same bucket in level *i*. As a matter of fact, the index data are only stored in the buckets of the last level (e.g., level 3 in Figure 3). The other level buckets store a Bloom Filter [16] to verify membership and are maintained virtually. We generate an ID for each bucket based on its ancestors' hash values. For example, the bucket $B_i = \{3, 9\}$ in level 2 has an ID “00” and the bucket $B_j = \{8\}$ in level 3 has an ID “110”. Instead of using the hash value as the key to publish the data, we use the bucket ID as the key. Initially, only level 1 buckets (e.g., bucket “0” and “1”) are inserted into the global index. If bucket 0 has a high query load, it will be split into two buckets in level 2. Then, the query load is shared between the two buckets. The index lookup is performed in a similar way. We generate a search key based on the hash function. For example, to perform search for 9 and 6, we generate keys "000" and "100", respectively. Query for "000" will be sent to the bucket "00", whose id is the prefix of the query.

4.2 Dynamic Expansion of the B$^+$-tree based Indexes

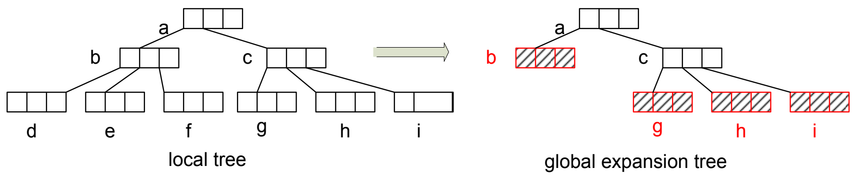


Figure 4: Adaptive Expansion of B$^+$-tree

In the index tuning process, the B$^+$-tree based global index can be considered a result of the dynamic expansion of the local B$^+$-trees. Figure 4 illustrates the idea. Due to network cost and storage cost, we cannot publish all the leaf nodes into the global index. Therefore, it is more feasible and efficient to select and publish some tree nodes based on the cost model. Based on Algorithm 3, the tuning process is similar to tree expansion or collapse. When a new processing node joins the cluster, it inserts the root node of its local B$^+$-tree into the global index. Then, it adjusts its index by expanding the tree dynamically. Figure 4 shows a snapshot of an expanding tree.

4.3 Cost Modeling

In our indexing framework, a cost model is essential to evaluate the cost and benefit of maintaining the global index. In As different system configurations will lead to different cost models, we describe a general approach to estimate the cost. Basically, maintenance costs can be classified into two types, query processing cost and index maintenance cost. Algorithm 2 indicates that query processing cost includes the routing cost incurred by *Map* and the index lookup cost incurred by *GetGI*. Based on the protocol of structured overlays, the cost of *Map* is $O(logN)$ network I/O, where N is the number of nodes in the Cloud. The cost of *GetGI* is the local I/O cost of processing the query via the global index, and it depends on the structure of current global index. For example, in an L-level B$^+$-tree index, if one h-level tree node n_i is inserted into the global index, query processing via n_i requires additional $L - h$ I/O cost. Thus, the cost of *GetGI* must be estimated on the fly. Once the local index is modified, we need to update the corresponding global index. A typical update operation triggers $O(logN)$ network I/Os and some local I/Os. The total index maintenance cost is a function of the update pattern. We employ the random walk model and the bayesian network model to predict update activities in the B$^+$-tree index and the multi-level hash index, respectively. Finally, the cost of a specific index entry is computed as the sum of its query cost and maintenance cost. And to limit the storage cost, we set a threshold for the size of global index. Then, the optimal indexing scheme is transformed into a knapsack problem. And a greedy algorithm can be used to solve the problem.

5 Other Implementation Issues

5.1 Concurrent Access

In an open service Cloud system, registered users are allowed to deploy their own softwares. If some users' instances access the global index concurrently, we need to guarantee the correctness of their behaviors. Suppose an index entry receives an update request and read request simultaneous from different instances. We need to generate a correct serialized order for the operations. A possible solution is to group the relative operations in a transaction and apply the distributed 2-phase locking protocol. However, 2-phase locking protocol reduces the performance significantly. If consistency is not the major concern, more efficient solutions may be possible [17].

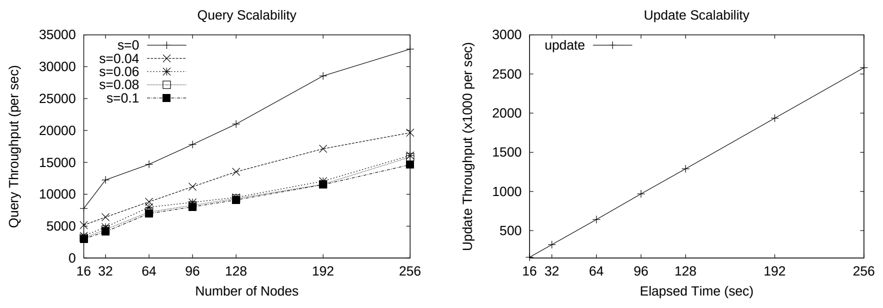
5.2 Routing Performance

As discussed in the cost model, *Map* incurs $O(logN)$ network I/O, where N is the number of nodes in the Cloud. Although nodes in the Cloud are connected via a high bandwidth LAN, the network cost is still dominating the index lookup cost. Some systems [18] apply the routing buffer to reduce the network cost. Generally, after a success lookup operation, the node keeps the destination's metadata in its local routing buffer. In the future processing, if a new lookup request hits the buffer, we can retrieve the corresponding data within 1 network I/O. However, the application of routing buffer incurs new research problems such as how to keep the routing buffer up to date and how to customize the routing algorithm.

5.3 Failure Recovery

In the Cloud system, as the processing nodes are low-cost workstations, there may be node failures at any time. In this case, a master node is used to monitor the status of nodes. And each node will record its running status into a log file occasionally. If a node fails, it will be rebooted by the master node and automatically resume its status from the log file. To keep the high availability of the global index, we write the global index into the log file as well. Moreover, we exploit the replication protocol of the overlay network to create multiple copies of the global index. Therefore, a single node's failure will not affect the availability of the global index. One of the replicas is considered as the master copy, while the other are slave copies. The updates are sent to the master copy and then broadcasted to the slave copies. Once a master copy fails, one of the slave copies is promoted to be the master one. And after a node recovers its global index via the log file, it will become a slave copy and ask the master one for the missing updates.

6 A Performance Evaluation



(a) Query Throughput (b) Update Throughput

Figure 5: Experiment Result

To illustrate the effectiveness of the framework, we have implemented our indexing framework on BATON [12] for the Cloud system (see [10] for more details). In our Cloud system, each node builds a local B$^+$-tree index for its data chunks. The global index is composed by a portion of local B$^+$-tree indexes. We deploy our system on Amazon's EC2 [4] platform. In our system, each node hosts 500k data in its local database. A simulator is employed to issue queries. From the start of the experiment, the node will continuously obtain a new query from the simulator after it finishes its current one. The major metrics in the experiment are query throughput and update throughput. To test the scalability of our approach, Cloud systems with different numbers of processing nodes are created. In Figure 5(a), we generate different query sets by varying the selectivity of the search. When $s = 0$, the query is exact search query. When $s = 0.01$, one percent of the data space is searched in the query. Query throughput increases almost linearly as the number of processing nodes increases. Figure 5(b) shows the update throughput. We generate the insertion request for each local B^+-tree uniformly. In our system, the updates can be processed by different nodes in parallel.

7 Conclusions

In this paper, we study and present a general indexing framework for the Cloud system. In the indexing framework, processing nodes are organized in a structured overlay network, and each processing node builds its local index to speed up data access. A global index is built by selecting and publishing a portion of the local index in the overlay network. The global index is distributed over the network, and each node is responsible for maintaining a subset of the global index. Due to storage cost and other maintenance issues, an adaptive indexing approach is used to tune the global index based on the cost model. Two experiments on a real Cloud environment, Amazon's EC2, illustrate the effectiveness and potential of the framework.

References

[1] Steve Fisher. Service Computing: The AppExchange Platform. *SCC*, 2006.

[2] M. Hui and D. W. Jiang and G. L. Li and Y. Zhou. Supporting Database Applications as a Service. *ICDE*, 2009.

[3] Merrill Lynch. The Cloud Wars: $100+ billion at stake. 2008.

[4] Merrill Lynch. Amazon Elastic Compute Cloud (Amazon EC2) http://aws.amazon.com/ec2/.

[5] IBM. IBM Introduces Ready-to-Use Cloud Computing, http://www−03.ibm.com/press/us/en/pressrelease/22613.wss.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[7] Marcos Aguilera and Wojciech Golab and Mehul Shah. A Practical Scalable Distributed B-Tree. *VLDB*, 2008.

[8] Sanjay Ghemawat and Howard Gobioff and Shun-Tak Leung. The Google file system. *SOSP*, 2003.

[9] http://hadoop.apache.org

[10] Sai Wu and Dawei Jiang and Beng Chin Ooi and Kun-Lung Wu. CG-index: A Scalable Indexing Scheme for Cloud Data Management Systems. *Technique Report (http://www.comp.nus.edu.sg/ wusai/report_09_01.pdf)*, 2009.

[11] Sylvia Ratnasamy and Paul Francis and Mark Handley and Richard Karp and Scott Schenker. A scalable content-addressable network. *SIGCOMM*, 2001.

[12] H. V. Jagadish and Beng Chin Ooi and Quang Hieu Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. *VLDB*, 2005.

[13] Adina Crainiceanu and Prakash Linga and Ashwin Machanavajjhala and Johannes Gehrke and Jayavel Shanmugasundaram. P-ring: an efficient and robust P2P range index structure. *SIGMOD*, 2007.

[14] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD*, 1984.

[15] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the Hilbert space-filling curve. *SIGMOD Record*, 30(1), 2001.

[16] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 2002.

[17] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 51-59, 2002.

[18] Giuseppe DeCandia and Deniz Hastorun and Madan Jampani and Gunavardhan Kakulapati and Avinash Lakshman and Alex Pilchin and Swaminathan Sivasubramanian and Peter Vosshall and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS*, 2007.