A Fast Data Ingestion and Indexing Scheme for Real-time Log Analytics *

Haoqiong Bian, Yueguo Chen†, Xiongpai Qin, and Xiaoyong Du

Key Laboratory of Data Engineering and Knowledge Engineering (MOE),
Renmin University of China, Beijing 100872, China
†Corresponding author: chenyueguo@ruc.edu.cn

Abstract. Structured log data is a kind of append-only time-series data which grows rapidly as new entries are continuously generated and captured. It has become very popular in application domains such as Internet, sensor networks and telecommunications. In recent years, many systems have been developed to support batch analysis of such structured log data. But they often fail to meet the high throughput requirements of real-time log data ingestion and analytics. An efficient index is very important to accelerate log data analytics, and at the meanwhile to support high throughput data loading. This paper focuses on designing a specialized indexing scheme for real-time log data analytics. The solution adopts a dynamic *global hash index* to partition the tuples into hash buckets. Then the tuples in the hash buckets are sorted and buffered in the *sort buffer queue*. When the amount of data in the queue reaches a threshold, the data is packed into *segments* before spilling to the disks. Moreover, an *intra-segment index* is maintained by *meta database*. With such an indexing scheme, the database system achieves high throughput and real-time data loading and query performance. As shown in the experiments, the data loading throughput reaches 5 million tuples per second per node. The delay of data loading does not exceed 10 seconds, and a sub-second query performance is achieved for the given queries.

Keywords: log data analytics, index, real-time, high throughput

1 Introduction

Log data has become a major type of big data. Some typical examples of log data include user click logs of Web pages, transaction records of sensor networks, or machine-generated records in a telecommunication network. Real-time analysis of log data is critical for many applications such as anomaly detection and event tracking. In enterprises, although SQL-on-Hadoop systems such as Pig [21] and Hive [27] have been widely used for batch analysis of big log data, MPP (Massively Parallel Processing) databases are however more efficient in handling real

* This work is supported by the Outstanding Innovative Talents Cultivation Funded Programs 2014 of Renmin University of China.

time analytical workloads [23, 25]. In MPP databases, various indexes can be easily applied to further improve the query performance.

Real-time log data analytics however brings some challenges to the existing indexing solutions of MPP databases. In the last second Chinese national contest of big data technical innovation[1], as a leading database manufacturer in China, GBase proposed the problem of designing an efficient indexing solution for GBase MPP cluster products [9] to support real-time log data analytics. The contest provides a typical real-world log data analytical workload. The queries in the workload aim to track the crucial Internet usage information from specified IP addresses. The data grows in a high speed and the queries must be capable of analyzing the fresh data in real time. Current index solutions in RDBMSs can hardly handle such real-time analytical workload of high volume log data.

To address the problem, we propose an innovative indexing scheme for real-time log data analytics: (1) The scheme, from the perspective of system architecture, is a multi-layer solution. Each layer is scalable so that the solution can adapt to the dynamics of data loading throughputs. (2) The layers in the scheme are pipelined, and each layer can be divided and conquered so that the schema can take full advantage of the hardware resources. (3) The in-memory buffers are separated to the mutable and immutable parts. As shown in Section 3.1 and 3.2, the *hash buckets* are the mutable part which is very small and the *sorted buffer queues* are the immutable part which is relative larger and available to queries. The data in hash buckets are continuously flushed into the sorted buffer queues so that the queries can access the fresh data in memory. (4) The scheme combines the features of hash index, in-cluster index and statistical information to support high performance indexing on the time-series data, and fast query processing for queries on attributes of high selectivity. (5) The scheme merges in-memory data into large segments. The segments are spilled to disks as sequential files. Within the segment, rows are split to row groups. *Intra-segment index* is built to address the right row groups in the segment. Within each row group, other database techniques such as column store [26], bloom filter [12] and other indexes can also be applied.

In this paper, we first give details of the real-world workloads used in the contest. Then we take the workloads as an example and summarize the main challenges and features of typical real-time log data analytics. After that, We introduce the details of our scheme and show some implementation details. Finally, we perform experimental evaluations of our solution, give the related work study, and conclude the paper.

2 Preliminaries

Real-time log data analytical workloads are different from the workloads in traditional decision-support systems. We introduce a typical workload in this section and summarize the challenges and features of real-time log data analytics.

[1] http://bigdatacontest.ccf.org.cn/gindex.html

2.1 Typical Data and Workload

The log data used in this paper is the records of Internet accesses. The log records are stored in a large table. The schema of the table is shown in Table 1. The name of the table is $WebInfo$.

Table 1. Table schema of $WebInfo$

Field name	Data type	Description	Cardinality
SrcIP	integer	source IP	∼30M
DestIP	integer	destination IP	∼30M
SrcPort	integer	source port	∼300
DestPort	integer	destination port	∼300
CaptureTime	integer	capture time (seconds)	
Flag	integer	flag bits	∼200
Protocol	integer	protocol code	<10
ISP	string	Internet Service Provider	<10
Area	string	area of location	∼2000
QueryType	integer	code of request type	<20

The $WebInfo$ table is stored in Gbase MPP cluster. The data is loaded into the table in a streaming manner. The coming data is partitioned to the $GNodes$ in the cluster. On each $GNode$, 30 billion tuples (log records) of data must be loaded into its local RDBMS engine per day. In peak hours, about 3 billion tuples of data should be loaded within one hour. The data must be loaded into the table in real time, i.e. the received data needed to be loaded into the table in seconds and be available to analytical queries immediately.

The queries in the workloads select the data by a range condition on the $CaptureTime$ column and two equi-conditions on the $SrcIP$ and $DestIP$ columns. The range of the capture time may be relatively wide, for example several hours, one day or even wider. Generally, the data in the past one month is considered as warm data and are analyzed by queries in the MPP database cluster. More earlier historical data are archived in other low-cost batch analytical systems such as Hive. The equi-conditions on $SrcIP$ and $DestIP$ specify the target Internet usage information we want to track. Based on the selected data with these three conditions, the query may further conduct aggregate and order-by operations. An example of the basic form of the query is shown as:

```
SELECT ISP, Area FROM WebInfo
WHERE CaptureTime BETWEEN 1377221057 AND 1377303456
AND SrcIP=<IP 1>
AND DestIP=<IP 2>
```

The selectivity of $SrcIP$ and $DestIP$ is relatively high. With the equi-conditions on these two columns, only a small amount (generally several hundreds or thousands) of tuples are selected from the billions of total records. But the workload is time critical, so that each query in the workload are expected

to be executed and finished in sub-second. To support the real-time analytical workloads on the log data, we need to build appropriate indexes for a large volume of data stored in the table.

2.2 Challenges and Features

We can summarize the challenges of building indexes for real-time log data analysis as following. The index solution must give a good tradeoff between the following two major challenges.

1. **Real-time ETL.** Data comes to database systems in a high throughput, and ETL (extract, transform and load) must be done in real time. Log data are usually generated by machines or user activities. So the data are growing rapidly and the amount of accumulated data is very large. In the example in Section 2.1, each node in the cluster has to load 830,000 tuples into the table per second on average during the peak hours. Such real-time ETL brings huge challenges to the storage module of database systems.
2. **Real-time query evaluation.** Queries in real-time log data analysis need return the results in sub-second. The queries in the workload are relative simple, but the queries may be submitted by many external clients such as web site users, unlike the report and decision support queries which are generally submitted by a few data analysts. So that the throughput of the queries may be relatively high and the clients can not wait for couples of seconds or even minutes.

The above two challenges are the two ends of a teeterboard: the first one is the challenge on write performance and the second one is the challenge on read performance. Tradeoff between these two challenges must be well considered in the indexing solution. Besides the above challenges, real-time log data analysis also has some other important features, from which we are able to develop dedicated indexing solutions to address the above challenges.

1. **Easy to scale out.** The queries in real-time log data analytical workloads are relatively simple and easy to scale out. Unlike typical report or decision support queries, there are neither joins between two large tables, nor complex aggregations on massive intermediate results. The main operations of the query in such log data analytical workloads is selection and projection, which are easy to scale out for a share nothing architecture.
2. **High selectivity columns.** The table has several columns with high selectivity. Most of the queries have predicates on these columns. Real-time queries should not touch a major part of the large data set. Otherwise it will be impossible for databases to support real-time and high throughput queries.
3. **Time series.** Log is a type of time series data. Generally, there is a timestamp field in the table schema of the log data. The field records the occurrence time of events or the generating time of the log entries. Besides, the value of the log data decreases as time goes by. The latest data is usually most important and valuable.

3 Indexing Solution

In this section, we introduce the details of our indexing solution. The framework of the solution, which has four layers, is shown in Figure 1. From top to bottom, in the first layer, we use global hash index to shuffle the log data based on the hash key. In the second layer, the data from each hash bucket is sorted and buffered in the corresponding sorted buffer queue. In the third layer, when the number of splits in a certain sorted buffer queue reaches a specified threshold, the splits are merged into a segment. An intra-segment index is built for each segment. In the bottom layer, the intra-segment index is stored in the meta database, the segment is stored into the file system. The technical details of the framework are discussed in the following subsections.

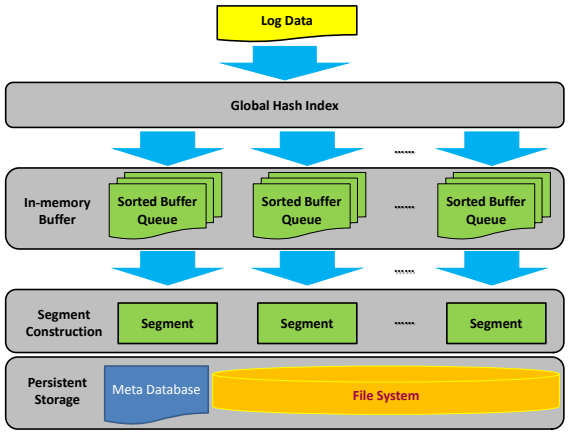

Fig. 1. Framework of the indexing solution

3.1 Global Hash Index

We build global index on the column(s) with high selectivity. If there are more than one column that always co-occur in query predicates, they can be considered as a combined key. This key is called *hash key*. For example, in the typical workloads shown in Section 2.1, $SrcIP$ and $DestIP$ are used by all the queries so that can be combined as a 64-bit integer ($SrcIP$ stored in the first 32 bits and $DestIP$ stored in the last 32 bits).

For the global index, we apply a strategy like consistent hashing [18] to supply a scalable hash index. We have a hash function $h = H(key)$ and calculate the hash value of the hash key. The range of the hash value is split to k intervals. The intervals are mapped to a set of hash buckets. An example is shown in Figure 2. The hash value ranges from 0 to $(2^{32} - 1)$. The range is split into 8 intervals which are mapped to 3 hashing buckets. The hash bucket is a small container in main memory (2MB by default) in which the tuples are temporarily buffered.

The global hash index is scalable so that the number of hash buckets are automatically and dynamically tuned by the database system. Tuples in hash

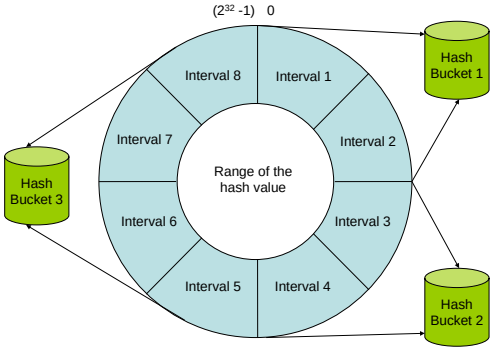

Fig. 2. An illustration of the global hash index

buckets can not be accessed by queries in our solution, because hash bucket is a mutable data structure, and synchronizing read and write operations on it will be complex and costly. So more hash buckets means more fresh tuples are not addressable by the analytical queries. This is not good for real-time ETL.

By changing the mapping between intervals and buckets, the database system can change the number of hash buckets dynamically in runtime. When the throughput (of new data) is low, the number of hash bucket decreases. At peak hours, the number of hash bucket will be increased. The up to date hash value range of the hash bucket is recored. When the bucket is full, an empty hash bucket is instanced to replace the full one. The full bucket is then set to immutable and sent to the in-memory buffer layer. In this paper, the immutable full hash buckets is called *buffer split*.

3.2 Sorted Buffer Queue

In the in-memory buffer layer, there are a set of *sorted buffer queues*. One hash bucket in the global hash index corresponds to one sorted buffer queue. The sorted buffer queue is a queue for the *buffer splits*. It is stored in main memory and can be accessed by the analytical queries.

After the buffer split is received by the in-memory buffer layer, it is sorted on the *sort key* and put into the corresponding sorted buffer queue. The *sort key* can be the same column used by *hash key* or another frequently used column in the table schema. The sorted buffer queue has a threshold of maximal size. If the number of buffer splits in the queue reaches the threshold or the flush buffer signal is emitted by the database system, an empty sorted buffer queue is instanced to replace the current sorted buffer queue. The replaced sorted buffer queue is then send to the *intra-segment index* layer to be merged into a segment and to build the intra-segment index.

The sorted buffer queues in the in-memory buffer layer must be flushed before the global hash index is scaled in/out. When the database system decides to scale the global hash index by changing the number of hash buckets and the mapping between the hash value intervals and the hash buckets, it must emit a flush buffer

signal to the in-memory buffer layer. After the sorted buffer queues are flushed, the global hash index and the in-memory buffer can be scaled.

3.3 Build Segments and Intra-segment Indexes

The *segment construction layer* is responsible for constructing the *segment* and *intra-segment index* from the *sorted buffer queue*. Intra-segment index is compatible with other database techniques such as column store [26] and other indexes such B^+-tree and bitmap. The diagram of building segment and intra-segment index is shown in Figure 3.

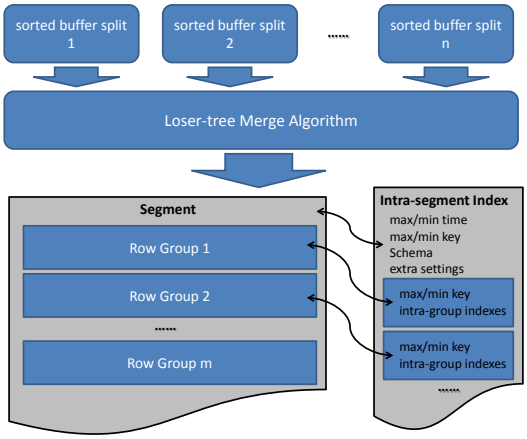

Fig. 3. Diagram of building segment and the intra-segment index

The sorted buffer queue contains a set of buffer splits which are sorted on the sort key. The sorted buffer queue also has attributes to record the min and max value of the hash key, the sort key and the capture time. The sorted buffer splits in the sorted buffer queue is merged by loser-tree merge algorithm [5] of which the computation complexity is $O(tlog^n)$, where t is the number of tuples in all the buffer splits and n is the number of buffer splits.

After merged by the loser-tree merge algorithm, the tuples form a *segment* which is split into m row groups. Then the *intra-segment index* is built for the segment. The intra-segment index contains the meta information of the segment and the meta information for each row group. The meta information of a segment includes (1) min and max values of the capture time, hash key and sort key, (2) the schema of the tuples and (3) the extra settings of the segment such as the compression codec of the row groups. The meta information of row group includes (1) the offset of the row group in the segment, (2) the min and max values of the sort key, and (3) the intra-group index.

The intra-group index is optional, it is the index on other columns except the capture time, hash key and sort key columns. It can be any type of index such as bloom filter[12], bitmap index[14] and B^+-tree. Intra-group index can be applied to accelerate the queries which have predicates on other columns.

It can be seen that within a row group, database techniques such as column store [26], B$^+$-tree, R-tree [16] and bitmap can be applied. Default in this paper, the size of the segment is 64MiB and the size of the row group is 4MiB which is a suitable size for column store of big Internet data [17]. The constructed segment and intra-segment index are then stored in the file system and the meta database. The file system can be local file system or distributed file system such as HDFS or GlusterFS [10]. The meta database can be stored in transactional databases such as MySQL or PostgreSQL.

4 Implementation

In this section, we discuss the implementation details of the indexing solution. The solution in this paper is implemented as a database storage engine in Java, and it is moving into the RDMDS engine of GBase MPP cluster.

4.1 Data Loading Pipeline

The framework of the indexing solution is implemented as a pipeline. As shown in Figure 4, the pipeline has six nodes. Each node in the pipeline is a group of parallel threads. The threads of a node do the same things concurrently. There are buffers between the consecutive nodes. Two consecutive nodes and the buffer between them constitute a producer-consumer model.

The number of threads in each node of the pipeline is automatically tuned by the database system. The indexing solution provides APIs to monitor the number of items in the buffers of the pipeline. So that the database system can tune the number of threads if any buffer in the pipeline is always full or empty.

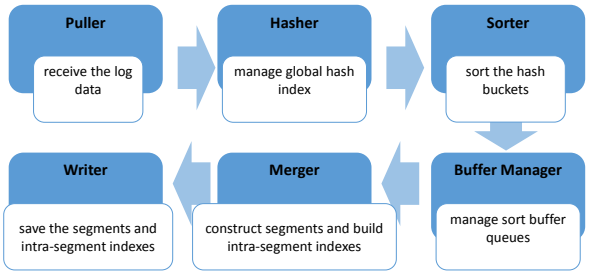

Fig. 4. The pipeline of data loading threads.

4.2 High Availability of Sorted Buffer Queue

The sorted buffer queues in the framework (Figure 1) is managed by *buffer manager* in the pipeline. To avoid losing data when the database system is crashed, the buffer manager stores the buffer splits in a local Redis[4] instance. The buffer split is stored as the binary value in Redis. Redis is run in another process and has data dumps and HA solutions. We define the abstract interface between our indexing solution and Redis, so Redis also can be replaced by other memory caches.

4.3 Lock on the Segment Construction Layer

During the time that the sorted buffer queue is merged and the intra-segment index is constructed, the system is in an uncertain condition. The data in the constructing segment is not available to queries. To solve this problem, we use a file lock on the constructing segment, so that the segment construction thread and the query execution threads are synchronized. The time of constructing or reading a segment is generally no longer than 200 milliseconds on PC servers. So this lock is a lightweight lock.

4.4 Avoiding Unnecessary Segment Accesses

The chance of synchronizing the queries and the segment construction threads is not very large because they are more likely to handle different segments. But to further improve the performance, we try to avoid unnecessary segment accesses of the queries by recording the earliest capture time of the tuples in all the buffer splits in memory. If a query is willing to access the data with earlier capture time than that, the query only need to access the segments in the file system. This helps to reduce the chance of lock competition.

4.5 Metadata Management

The meta database is responsible for storing the in intra-segment indexes. It can be an RDBMS such as MySQL. The queries firstly query the meta database to find the right segment for themselves. The intra-segment indexes are also save as the header of the corresponding segments. The intra-segment index includes all the metadata which is needed when reading the segment. So, if the segments are copied to another cluster node, the meta database can be easily upgraded.

5 Experimental Evaluations

5.1 Experiment Environment

We are focusing on the indexing solution in the storage engine on nodes of a MPP database cluster. We care about the performance of the indexing solution on one node. So in our experimental evaluations, we only use one PC Server. The environment of the server is shown in Table 2.

5.2 Data Loading Performance

We use the data in Section 2.1. We use 3 billion tuples of data to test the data loading performance. MonetDB and TokuDB support insert while Infobright does not. But the insert is an transactional operation of which the throughput is generally lower than 10,000 ops/s. So we split the data into mini batches, each mini batch contains 0.1 million tuples. The mini batches are loaded with the copy/load command in each system. The data loading throughput is shown

Table 2. Settings of the experiment environment

Item	Setting
CPU	Intel Xeon E5-2640 2.50GHz × 2
Memory	64 GB
Disk	1TB 7200RPM SAS × 4, RAID5
Operating System	CentOS release 6.4 x86-64 (Final)

in Figure 5. It can be seen that the throughput of our solution is one order of magnitude higher than those of the three systems. Infobright does not supprt index. Indexes are built on *SrcIP*, *DestIP* and *Capture Time* on the other three. The time between the moment at which data is passed to the system and the moment at which data is available to the queries of all the four are no longer than 10 seconds.

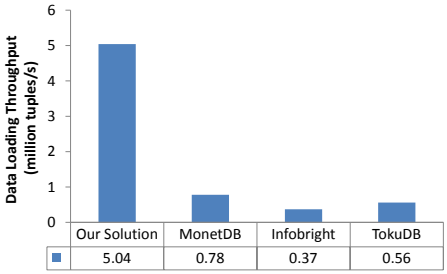

Fig. 5. Data loading throughput

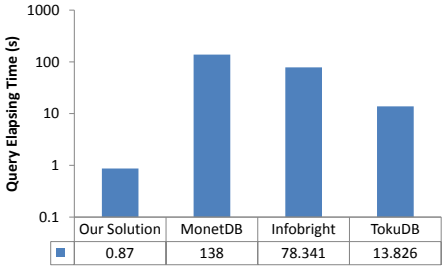

Fig. 6. Query elapsing time

5.3 Query Performance

After the data is loaded, we execute the query in Section 2.1 to evaluate the query performance. The result is shown in Figure 6. It can be seen that the query performance of our solution is two order of magnitudes higher than Infobright and MonetDB and one order of magnitude higher than ToluDB. The reason why Infobright and MonetDB is so slow is that they do not support the real index. Index is just an advice to MonetDB and may be ignored. More important is that, when loading data, queries on the other three systems are almost blocked while in our solution the queries are not significantly affected.

6 Related Work

Traditional index such as B-Tree is widely used in databases and data warehouses[1]. It supports range condition and equi-condition on columns. But it is a kind of heavy wait index. When log records arrive rapidly, the b-tree will keep

updating the tree nodes and hierarchical structure frequently[20], which limits the throughput and performance of data insertion. Write times of b-tree grow considerably as the data becomes scattered in different disk sectors over time. More importantly, when the index is updated frequently, the select tasks and the data insertion tasks must be frequently synchronized and the performance of both select and insertion will be restricted. Hash index used in most RDBMS does not support range queries so that it is not suitable for our workload although it is faster than b-tree[6]. Bitmap index is also a type of common used index in RDBMSs, but it is only suitable for low cardinality columns[11].

Besides the traditional b-tree, hash and bitmap index, there are some other research prototypes and system implementations of novel indexing techniques. log-structured merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for high-rate record inserts over an extended period[22]. But LSM-tree sorts the data on the index key so that it can only index one column in the table. Systems such as LevelDB[7], Bigtable[15], Hbase[3] and Cassandra[19] implement LSM-tree. They are key-value storage systems do not support both equi-conditions on the high selectivity column and the range conditions on the time series column. But another index called *fractal tree* is a tree data structure supplies the same queries performance as a B-tree but with insertions and deletions that are asymptotically faster than a B-tree[2]. It is currently used in TokuDB[8].

In recent years, column store is a hot topic in analytical database area. Techniques to improve the query and insertion performance of DBMS are well researched. In columnar databases such as C-store[26], infobright[24] and monetdb [13], column statistics and just-in-time indexes are built to speed up analytical queries[24, 13]. These can be considered as lightweight indexes and are low-cost and efficient for data insertion. Infobright is a disk-based columnar storage engine while MonetDB uses memory aggressively. In this paper, we compare our solution with Infobright, MonetDB and TokuDB.

7 Conclusion

Index is very important to real-time log data analysis. It can be seen from Section 5.3 that the two state-of-the-art open-source analytical databases, MonetDB and Infobright are failed to provide good query performance due to the lack of indexes. To support high throughput and real-time data loading and query evaluation, we designed an efficient indexing scheme of which the loading throughput and query performance are 1-2 order of magnitudes higher than the existing baselines.

References

1. http://docs.oracle.com/cd/b28359_01/server.111/b28313/indexes.htm#i1006549.
2. http://en.wikipedia.org/wiki/fractal_tree_index.
3. http://hbase.apache.org/.

4. http://redis.io.
5. http://sandbox.mc.edu/ bennet/cs402/lec/losedex.html.
6. https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html.
7. https://github.com/google/leveldb.
8. https://github.com/tokutek/tokudb-engine.
9. http://www.gbase.cn/comcontent_detail1/&i=30&comcontentid=30.html.
10. http://www.gluster.org.
11. http://www.oracle.com/technetwork/articles/sharma-indexes-093638.html.
12. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.
13. P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.
14. C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD*, volume 27, pages 355–366, 1998.
15. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
16. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
17. Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE*, 2011.
18. D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.
19. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS*, 44(2):35–40, 2010.
20. P. L. Lehman et al. Efficient locking for concurrent operations on b-trees. *TODS*, 6(4):650–670, 1981.
21. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.
22. P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
23. A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.
24. D. Ślzak, J. Wróblewski, V. Eastwood, and P. Synak. Brighthouse: an analytic data warehouse for ad-hoc queries. *PVLDB*, 1(2):1337–1345, 2008.
25. M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: Friends or foes? *CACM*, 53(1), Jan. 2010.
26. M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.
27. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.