Technical Note

SG-ColBase: A Relational Column Database Kernel

Tong Zhou [1], Tingzhen Liu [2]*

[1] Trip (Hotel BG); tong.zhou@trip.com
[2] Tencent IEG; sgfirstliu@tencent.com
* Correspondence: firstsg@outlook.com; Tel.: +86 15042683303)

Abstract: At present, diversified and highly concurrent businesses in the Internet industry often require heterogeneous databases formed by multiple databases to meet the needs. This report introduces database kernel SG-ColBase we developed. After achieving read and write concurrency control, data rollback, atomic log writing, and downtime data redo to ensure complete transaction support. The parallelism of database kernel execution is extended through field level locks and snapshot reads. Use the Bloom filter, resource cache pool, memory pool, skip list, non blocking log cache, and asynchronous data writing mechanism to improve the overall execution efficiency of the system. In terms of data storage, column storage, logical key and LSM-tree are introduced. While improving the data compression ratio and reducing data gaps, all disk data operations are written in incremental order. With the characteristics of asynchronous batch operation, the data writing speed is greatly improved. Thanks to the continuous feature of vertical data brought by column storage, the disk scanning brought by vertical traversal is reduced, which is a qualitative leap in efficiency compared with traditional relational databases in the big data analysis scenario. SG-ColBase can reduce the use of heterogeneous databases in business and improve R&D efficiency.

Keywords: Relational Database; Columnar Storage; Bloom Filter; Skip List; Field Level Lock; Read Write Concurrency; OLTP; OLAP; LSM-Tree; Token Bucket Algorithm

1. Introduction

With the rapid development of the Internet. With the proliferation of users and the continuous expansion of business, the traditional row relational database is often unable to respond well to the following ultra-high concurrency and large data analysis and operation requirements due to its own architecture design, but the business with high data consistency requirements (OLTP) can not be separated from the row relational database that supports complete transactions for the time being[1]. Therefore, many Internet companies choose to use some NoSQL databases to assist row relational databases to form heterogeneous databases[2] to meet various needs. For example, Redis is used as an out of process cache to reduce requests for row relational databases and improve system concurrency[3]. HBase is used for online big data analysis and statistics (OLAP) to make up for the lack of large data operation and retrieval in row relational databases[4]. However, such assistance comes at a cost. The first thing is to set up services for these NoSQL databases. It is not enough to set up a small number of instances of such services. A single data partition often requires multiple instances for high availability, which increases the service operation cost enormously. Secondly, the introduction of databases requires additional consideration of data consistency, which also increases the difficulty of team development. This is a great pressure for small and medium-sized Internet companies.

Improving the concurrency of database systems is an eternal topic in the database field, and researchers have proposed many methods. In terms of reducing the mutual exclusion of read and write operations, MVCC (Multi version Concurrency Control) uses redundant multi version data, operation version number and deletion version number to locate the data version that the transaction should read to achieve read and write paral-

 doi:10.20944/preprints202211.0220.v2

lelism. In addition, the big data analysis and statistics needs are often met by using a column database, which uses column storage[5]. Compared with row storage, it is more compact, and because column data is stored continuously, it has natural advantages in column oriented statistical analysis. C-Store first applies column storage to the database. It redundantly divides data into different projects by column. This allows C-Store to select the optimal group of projects when querying, minimizing the cost of query execution. BigTable[6] innovatively proposed the concept of column clustered storage, which to some extent has the advantages and disadvantages of column storage with high data compression ratio and row storage with convenient data tuple extraction. However, it can only have one sort key in the same logical table. This structure limits some search performance.

This project has completed the module implementation and system combination of SG-ColBase, A Relational column database kernel. The project is independently developed using C++. It surpasses traditional relational databases in parallelism, execution efficiency, big data operation analysis, etc.

2. System Design

In order to surpass traditional relational databases in parallelism, execution efficiency, big data operation analysis and other aspects, we need to make improvements in some key modules and components.

In terms of extending concurrency, use field level locks instead of row level locks, which makes it possible for multiple transactions to operate on the same row of data without competition. An additional snapshot data is maintained so that reads and writes to the same resource can be performed in parallel between transactions.

In terms of improving execution efficiency, each column of data automatically maintains a bloom filter. Before accessing specific data, use the bloom filter to filter requests to reduce data access. Set up cache pool and memory pool in user space to reduce system calls. Use skip list instead of red black tree as the memory structure of the index to improve the speed of index retrieval and modification. Build a non blocking log cache to eliminate the blocking of transaction commit when the log cache is written to disk. Asynchronous data writing mechanism to reduce transaction submission time. The commit operation ends after the synchronized log is appended to the disk.

In the aspect of big data operation analysis, because of B+ Tree index, the data in traditional relational databases are stored in primary key order. A large number of data insertion operations will result in a large number of random disk accesses and disk node splitting. We choose to use the column data storage structure and LSM-tree disk index structure, so that all operations are written in incremental order, which improves the data writing speed. It also brings faster analysis speed of large amount of data.

In addition, consider the future distributed cluster version. In the distributed cluster scenario, data is stored in partitions, and many operations need to be completed jointly by multiple data partitions. The upper layer needs a proxy service to coordinate calls. Therefore, our stand-alone database kernel only provides some basic functions of addition, deletion, modification and query. Multiple simple functions can be used together to realize various complex queries.

To sum up, the system architecture is divided into three layers:

1. **Organization calling layer**. Check the parameter format and data type, and make encapsulation calls for each module component.

2. **Module component layer**. Perform various operations, such as data extraction, data retrieval, data writing, and data filtering.

3. **Infrastructure layer**. Provide underlying services, such as resource cache pool and buddy memory allocation pool established in user mode.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

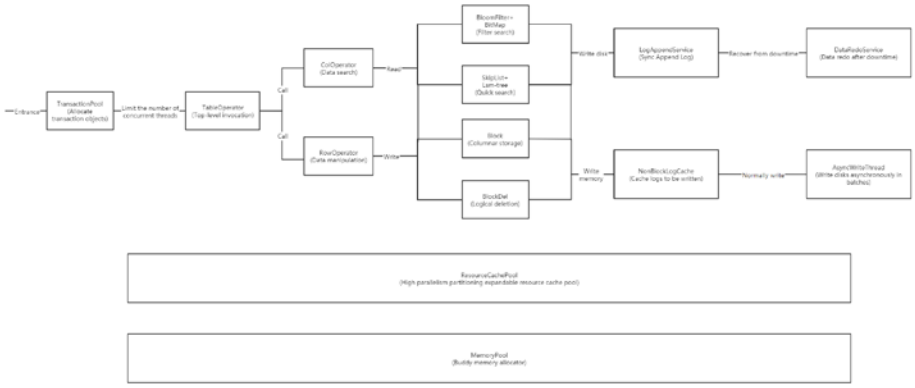

Figure 1. System architecture.

3. Module Design

Resource Cache Pool. As an IO intensive application, the database kernel must use cache to reduce disk IO. However, if you use the page cache provided by the operating system to cache resources, you need to make a system call every time you get resources, and copy the data from the kernel space to the user space. Therefore, we choose to disable page caching and cache resources in user space to reduce the overhead of kernel mode user mode switching. We choose to use the LRU algorithm[7] for cache elimination to balance cache hit ratio and cache elimination efficiency. In order to improve the parallelism of resource cache pools, we store resource partitions in different resource cache pools according to the unique ID of the resource, and lock the partitions to control access. The consistent hash algorithm[8] is used to reduce the impact on the overall resource pool operation when the resource pool expands partitions.

Data Architecture. To reduce the data footprint, we use logical keys instead of physical keys. And partition resources (units loaded into memory from disk) on the column. When accessing data, it will be obtained from the resource cache pool in column segments, and the data in the column segments will be continuously stored in memory. This design, in combination with the user mode resource cache pool, makes large-scale column access almost equal to sequential memory access, greatly improving the efficiency of large-scale data analysis and statistics. The deletion of data requires additional segment records. This logical deletion design reduces data movement. All data modification operations are written in incremental order.

Index Architecture. The index uses partition design to improve execution efficiency and parallelism. During access, the system will perform a binary search in the partition information table based on the key value to determine the index partition where the key value is located.

Disk organization structure. Traditional relational disk databases often use B+ Tree as the disk structure of the index, but it is not friendly to the index modification of large amounts of data[9]. For example, writing a large amount of data in the middle of B+ Tree will cause a large number of disk moves and random accesses. Therefore, we choose to use LSM-tree with high write performance. Each time the index is modified, only the write operation record list (key order) needs to be appended at the end of the file. LSM-tree also has its own problems. Every time it reads into the disk, it needs to merge and sort the operation record lists of multiple key orders and restore them to the key value list of the key order. However, thanks to the partition mechanism, the workload is not too large each time. For further optimization, this partition will be fully merged when the number of operation record lists reaches the threshold to improve the reading efficiency[10].

Memory organization structure. To better match LSM-tree, skip list[11] is used as the data structure indexed in memory. It can directly build the index node on the ordered key value linked list given by LSM-tree. Compared with red black tree, hop table has

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

higher performance. The implementation also provides parameters such as layer node multiple to balance the data retrieval performance and index modification performance.

Bloom filter architecture. In order to reduce the full reading of the bloom filter data, the bloom filter is stored in segments. When judging data, you only need to read the data in the data partition. This can improve execution efficiency and parallelism. Partition according to the hash value of the key, and use the binary search algorithm to locate the partition. Since the hash value is far greater than the range represented by the partition bloom filter, it is necessary to map by taking the maximum value of the modulus range interval of the hash value. When the maximum value of the range interval is the power of 2, the mapping is uniform. We use Bitmap[12] as the underlying data structure of the Bloom filter. When checking whether the data exists, obtain the bit at the location of Bitmap data hash value to determine whether the data exists. Due to data collision, it is difficult to delete the bloom filter. Therefore, the bloom filter does not delete data. In order to prevent the false positive rate of the bloom filter from being too high, all data will be re scanned regularly to reconstruct the bloom filter.

Transaction. (1)**Isolation**: Multiple methods are used to isolate the impact of multiple users on each other when operating the database. In the scenario with high requirements for real-time data, access to data is controlled through the read/write lock on the data. In the scenario where only legal data is required to be read, read snapshot data to isolate read/write concurrency. The isolation level of transactions during snapshot read is read commit. (2)**Atomicity**: Atomicity in memory is realized by locking. In disk operations, atomicity refers to that there are only two states of disk data modification: success and failure, and there is no intermediate state. This is difficult to achieve in the scattered data on the disk. We choose to use an atomic log appending mechanism to achieve this. The appended log may exceed the size of a disk sector, which makes the log writing in an intermediate state. Select to remove the intermediate state by adding the valid file length field to the file header. (3)**Durability**: When submitting, the logs are written to the disk first and then to the memory, with an effective downtime data redo mechanism. The data will not be lost after the commit function is executed successfully.

Asynchronous incremental write. When a transaction is committed, the log is written to the disk and the data is not synchronously waiting to be written to the disk. Instead, the task is handed over to the asynchronous incremental write thread. This thread will batch write data when the number of logs in the cache exceeds the threshold, and update the data writing progress. The problem with this mechanism is to ensure that all operations on the resource have been written to the disk before it is loaded into the disk. Delayed updates are always advocated to improve performance. Therefore, we choose to determine whether there are logs in the cache that have not been written by this resource before reading resources from the disk. If there are logs in the cache, all logs in the cache will be written to the disk. In order to achieve a high level of delayed updates.

Non-blocking log cache. After the log cache is full, the asynchronous write thread will lock the log cache and start writing based on the log increment in the cache. This often takes a long time, during which the log cache cannot provide caching services for transactions. In order to reduce the impact on transaction commit, a non blocking cache pool is developed. The two internal cache pools alternately provide external log caching services. When one cache is full, it writes data to the disk, and the other cache provides external log caching services.

Data redo after downtime. During downtime recovery, read logs that have not been updated to disk according to the effective log progress and data write progress. Because the data writing progress may not be updated in time, the downtime data recovery needs to achieve idempotence.

Data snapshot read. Traditional Copy-On-Write copies a snapshot before the operation, then reads the snapshot during the operation, and then points the pointer to the snapshot after modification. This is used to isolate read and write operations. Under this mechanism, there is no need to use locks for serial read and write operations, which in-

 doi:10.20944/preprints202211.0220.v2

creases the parallelism of the system. But it will bring a lot of copy and memory application operations, and there may be no concurrent read operations during the write operation, which makes many operations useless. We choose to maintain a snapshot data all the time to completely isolate the current data and snapshot data. Update the snapshot incrementally every time the data is submitted, reduce the full copy of the data, and request memory only once when creating the snapshot.

Memory Management. We create memory pool in the user space to reduce memory requests in kernel mode. And use buddy allocation algorithm to reduce memory fragmentation. Synchronous memory release will also consume a lot of service thread time. Therefore, when a large amount of memory is released, we submit the task to the thread pool for asynchronous execution to reduce the blocking of service threads[13].

Flow Control. During business operation, excessive requests may occur at some time. If not limited, slow response and heavy system load will occur due to a large number of thread switches. Traditional flow control methods, such as sliding window and leaky bucket algorithm [14], limit the number of requests accepted per unit time. It does not take into account the problem that a large number of requests are piled up in the system because the request has not ended for a long time. We have developed an **improved token bucket algorithm** to limit the number of parallel service threads in the system: transactions correspond to tokens, and transaction pools correspond to token buckets. When the database kernel is initialized, build a transaction pool with a specified number of transaction objects. When users request, they first need to request a transaction object from the transaction pool. If there is no transaction object in the transaction pool, block until the transaction object appears in the transaction pool and gets successfully. The database operation starts. After execution, put the transaction object back into the transaction pool. This allows the maximum number of parallel service threads in the database kernel to not exceed the threshold set at initialization. The thread safety of the transaction pool is guaranteed by mutex, and the semaphore is used to limit the number of parallel accesses.

4. Data Storage Implementation

Data in SG-ColBase is stored in columns. Compared with row storage, multiple threads in column storage can access different fields of a row of data at the same time without data competition. BigTable model is based on traditional KV database storage, which leads to a large number of redundant physical key storage, and the actual data only occupies a small part of the storage space. Data in SG-ColBase is organized by logical keys, which increases the proportion of effective data in the storage space. To reduce the load on a single data storage module, we divide the data on the column into multiple segments. Each segment serves as a resource (unit of disk storage and loading into memory). Because column data is stored tightly according to the number of bytes occupied by the data type of the field. This feature enables you to directly locate the location of data on the disk and modify it through subscripts when modifying data. Because the data is stored in an unordered and decentralized way, the free column segments can be randomly selected for insertion at the end of the data insertion operation. In order to prevent the relative position of the key value from moving during data deletion, additional resources are needed to incrementally record the subscript of the deleted data (i.e. logical deletion).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

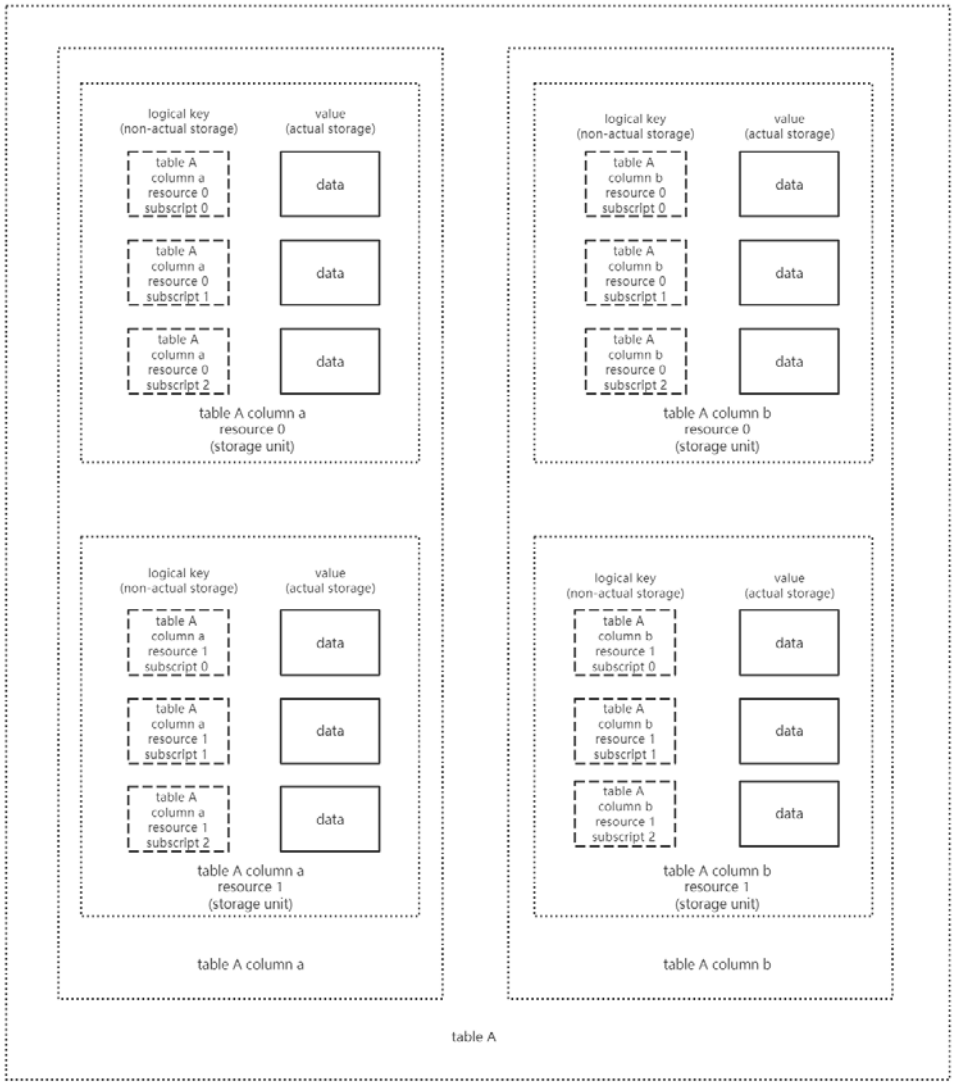

Figure 2. Data structure of column storage.

Because column data is stored tightly according to the number of bytes occupied by the data type of the field. This feature enables you to directly locate the location of data on the disk and modify it through subscripts when modifying data. Because the data is stored in an unordered and decentralized way, the free column segments can be randomly selected for insertion at the end of the data insertion operation. In order to prevent the relative position of key values from moving when deleting data, additional resources are needed to incrementally record the subscripts of deleted data (i.e. logical deletion).

In batch operation, there may be some operations that can be simplified. For example, the same data can be modified to one value first and then to another. Another example is to add a data first and then modify the data to another value. The result of such multi-step continuous operation is the same as that of the final value. For this kind of operation, we have implemented a simplification algorithm, which can identify and simplify the operation chain of data. To reduce unnecessary random writes during incremental data writes. Sort data by disk location before writing.

5. Index Storage Implementation

In order to avoid the cost of node splitting and random access caused by using B+Tree as the disk index, we use LSM-tree as the index disk storage structure; Use skip list as ordered collections in memory. This allows for faster operation.

 doi:10.20944/preprints202211.0220.v2

5.1. LSM-Tree

From a macro perspective, the storage structure of LSM-tree in disk is multiple ordered operation sequences. This allows us to modify the data by appending the operation record sequence to the end of the file. Compared with random writing in the overall ordered set, the efficiency of incremental data writing has been greatly improved. However, such a storage structure brings disadvantages to reading. In B+ Tree, because the data is globally ordered, you can only read some of the data. But in LSM-tree, data is divided into multiple ordered operation sequences. Reading only some of them will not get correct results. Therefore, every time we read, we need to read all the data and merge and restore multiple ordered operation sequences.

To reduce the cost of LSM-tree reading. First, we partition the index on the column by value range. When retrieving data, we only need to read the partition of the data to be retrieved. This greatly reduces the data to be read for data retrieval. In order to keep the complexity of data retrieval unchanged, binary search algorithm is used to locate partitions. When there are too many ordered operation sequences in the LSM-tree disk storage of a single partition, the deserialization will result in a long merge operation. We have developed a merging mechanism. When the number of ordered operation sequences exceeds the threshold, the disk storage will be merged as a whole. Finally, build a resource cache pool in the user space to reduce the number of disk reads.

5.2. Skip List

The biggest problem with red black trees as indexed memory storage structures is that the time complexity of building them is $O(nlogn)$. Even if it is created based on sorted collections read from disk, the complexity remains the same. skip list is actually an ordered linked list with indexes. Compared with traditional ordered linked lists, data access can be accelerated through index nodes. And it still retains the advantage that linked lists are easy to modify. Therefore, it can directly generate index nodes on the ordered linked list provided by LSM-tree after deserialization, so as to achieve $O(n)$ complexity of building skip list. The layer multiple parameter is provided in the implementation to balance the efficiency between modification and query. After testing, compared with red black trees. The execution speed of skip list is improved by about 30% in both modification and query operations.

6. Resource Cache Pool Implementation

To reduce the cost of system calls, we disable the page cache of the operating system and establish a resource cache pool in the user space.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

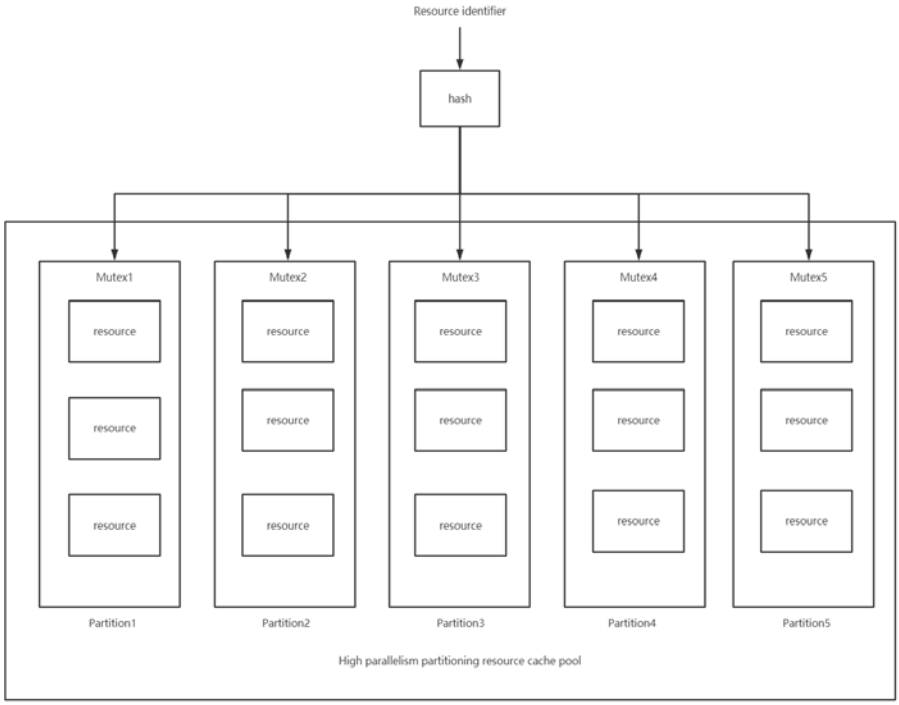

Figure 3. Resource cache pool architecture.

6.1. LRU Cache Elimination Mechanism

LRU generally need a queue to maintain access information and a hash table to quickly extract resources. The key to improving the execution efficiency of the cache pool is to let the data in the two data structures access each other with *O(1)* complexity. The hash value is used as the linked list node, so that the hash table can access the queue *O(1)*. The resource pool can efficiently maintain its position in the queue when extracting resources. Then, access the hash table through the key value *O(1)* in the node through redundant keys. In this way, when resources are eliminated, the storage of the least frequently accessed elements in the queue in the hash table can also be quickly deleted.

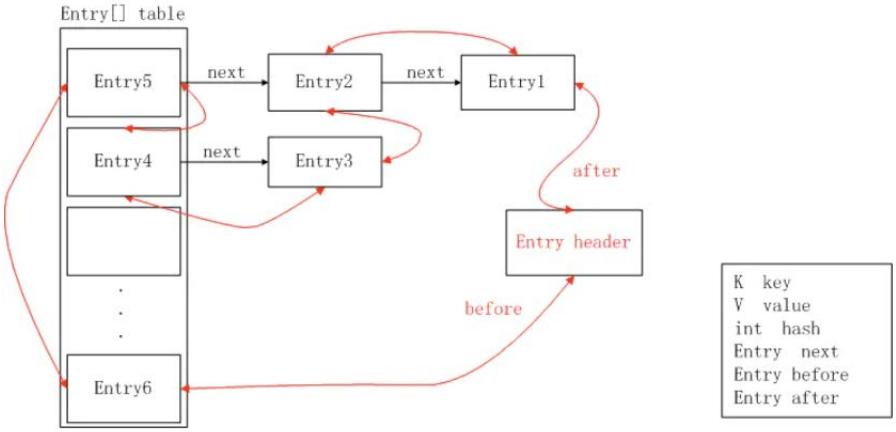

Figure 4. Data structure of LRU.

6.2. Partition Parallelism

When concurrent operations exist in the resource pool, some measures are needed to ensure data consistency. For example, the non-lock CAS operation. It can ensure consistency in user mode and reduce system calls. It is suitable for data concurrency with less

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

competition and short operation time. However, the resource cache pool will have extremely high concurrency due to a large number of calls made by a large number of user threads, and if the cache page is missing, triggering IO will result in long-term operations. Therefore, CAS does not apply to resource pools. The other is to lock data access to ensure consistency. However, when a large number of threads compete for a lock, the parallelism is too low, resulting in a large number of threads blocking. In order to improve the parallelism of resource pools, we separate data partitions. Access to data in the same partition competes for a lock. When locating partitions, binary search algorithm is used. In this way, the parallelism of the resource pool can be controlled by the partition isolation number under the condition that the resource extraction efficiency is almost unchanged.

Hash modulus is usually used as the partition basis, which has the lowest execution complexity. However, when you want to increase partitions to expand the overall concurrency of the resource pool (increase partitions), you need to re hash modulus. This makes almost all data need to move partitions, which will make the entire resource pool unavailable during the move process. In order to reduce the impact of extended parallelism on the overall availability of the resource pool, we use a consistent hash algorithm[8] as the partition basis. This algorithm is based on the range of hash values as resource partitions, and will split an original partition when the degree of parallelism expands. This operation only affects the partition where the split operation is performed, and has little impact on the overall availability of the resource pool.

6.3. Resources Asynchronous Release

When the LRU performs cache elimination, the eliminated resources may still be used by user threads. Therefore, we need a mechanism. Wait for all users to use before releasing. If a variable that guarantees memory visibility is used to mark and wait in a loop. Although waiting can be completed in user mode, other user threads often operate on resources for a long time. This will result in a large amount of CPU time. Therefore, we use read/write locks: all resource pool acquisition operations will apply read locks to the acquired resources. Before releasing resources, we need to obtain the write locks of the resources to be released. In this way, when acquiring resources, there will be no blocking because the read lock and the read lock are shared. The resource release operation will also block until all read locks are released, that is, all user threads use the resource. During specific operations, the operation of removing data from the data structure is performed synchronously to ensure that other user threads cannot obtain obsolete resources. Blocking, waiting and resource release are handed over to the thread pool for asynchronous execution, reducing the lock occupying the resource pool partition.

7. Transaction Implementation

7.1. Concurrency Control based on Read-Write Lock

The operations on resources are divided into read operations and write operations. Multiple read operations can be executed in parallel, but read-write and write-write operations cannot be executed in parallel due to data competition. Based on the above situation, read/write locks are used to obtain resource locks before accessing resources to control concurrent access of multiple user threads to the same resource. In order to balance the consistency and execution efficiency of transactions, different isolation levels are provided through different invocation methods. For example, the read commit isolation level (current read) releases the resource lock immediately after access, and the repeatable read isolation level releases the resource lock after transaction commit. The locking mechanism will ensure the atomicity, visibility and orderliness of resource access, thus further ensuring the atomicity and isolation of data in the database memory.

Considering that OLAP scenarios need to access all data in a single column. Getting a lock once for each data in the column will result in a significant performance degradation. Therefore, the lock granularity is not refined to the level of each row and field, while the resource level is locked (each column segment and field).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

7.2. Concurrency Control based on Snapshot

Generally, the Copy-On-Write mechanism is used to solve the problem that the above read and write operations cannot be concurrent. The data will be copied before the write operation, and the copied data will be written. Finally, the modified data will be overwritten. It is generally used in business scenarios where more reads and less writes are needed. In the large-scale data writing scenario, the execution efficiency will drop sharply due to a large number of memory application operations. MVCC multi version concurrency control used in traditional relational databases will bring the same problem.

In order to solve the above problems, a data snapshot mechanism is developed. The data snapshot is created when the resource is read into memory. It has the same life cycle as the resource and is completely isolated from the resource. Users do not need to make copies before writing resources. When the user can accept the transaction isolation level of read commit (snapshot read), the snapshot data will be read during the read operation. Because snapshot data is completely isolated from resources, read operations on snapshot data and write operations on resources can be performed in parallel. The snapshot data will be updated incrementally according to the log when each transaction is committed. This snapshot mechanism can support both parallel read and write operations and large-scale data writes. Snapshot data requires a read/write lock to control access. The write lock is applied when the snapshot is updated, and the read lock is applied when the snapshot is read.

7.3. Ensure Atomicity of Log Writing

The operation log of the transaction will be written when the transaction is committed. If the operation log is too long, multiple sector writes may be involved. This causes the log write operation to be non atomic. This will lead to data inconsistency during downtime. In order to make appending and writing atomic, a field is set in the file header to record the effective length of the file (8 bytes, atomic read and write). After appending and writing, atomic log appending and writing can be realized by modifying the effective length field of the file.

Figure 5. Sequence structure of logs on disk.

7.4. Non-blocking Log Cache

When a transaction is committed, the log and log cache are written to disk. After the logs accumulate to the threshold, the changes to the data will be synchronized to the disk according to the logs. For the log cache, the transaction commit operation is a write operation, and the read operation is to read the log and synchronize the data changes to the disk. Therefore, data synchronization and submission operations cannot be performed in parallel. In order to reduce the blocking of transaction commit operations during data synchronization, we developed a non blocking cache. It is internally composed of two caches, namely service cache and synchronization cache. The service cache provides log caching services for transaction objects. When logs accumulate to the threshold, the cache switches. The service cache is switched to synchronous cache, and the data changes are synchronized to the disk according to the log. The synchronization cache is switched to the service cache to provide services to the transaction object.

In the implementation, we considered that when the cache switch is triggered, the synchronization cache may not have finished synchronizing changes to disk. Therefore, we choose to block and wait for the synchronization cache to complete before the cache switch.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

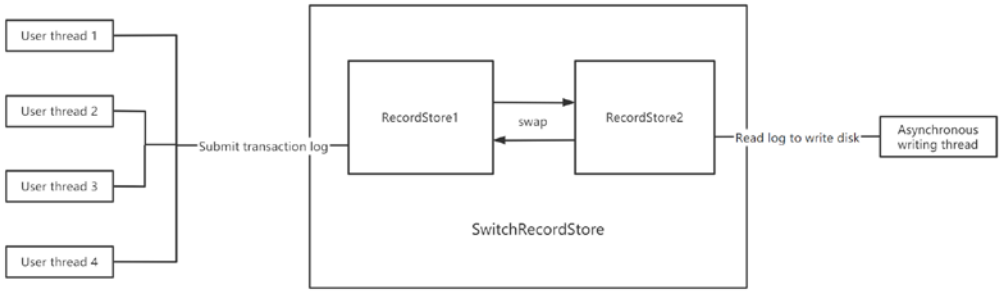

Figure 6. Double Cache Service.

7.5. Asynchronous Data Writing

To reduce the blocking during transaction commit, the commit thread returns immediately after the log and log cache are written to disk. Give the data synchronization task to the dedicated thread for asynchronous execution. Use semaphores to control thread data synchronization. Specifically, the semaphore of the number of logs will be released after the transaction is committed, and the synchronization thread needs to obtain this semaphore before each data synchronization. When there is no acquisition, the CPU will be released. Update the data synchronization progress field in the file after the batch data synchronization is completed.

Asynchronous data writing may result in data changes not being synchronized to disk before resources are re read into memory. To prevent this, we have developed a mechanism to determine whether there are changes in the log cache that have not been synchronized to the disk before each resource is read into the disk. If it exists, the semaphore will be released directly to force data changes to be synchronized to the disk. Read resources into memory after successful synchronization. To achieve efficient and secure asynchronous data writing.

Long log files may cause insufficient disk space. Therefore, after each data synchronization, if the log file length exceeds the threshold and all data changes in the file are synchronized to the disk, the log file is cleared. Reduce the disk space occupied by log files.

7.6. Data Redo after Downtime

Based on the previous work, we have made many preparations for downtime data recovery: atomic logs are written to disk, logs are written to disk first and then to memory, and idempotent data synchronization. During downtime data recovery, the effective file length field and data synchronization progress field are used to determine the logs that are not synchronized to the disk. Read and synchronize data changes to disk to update data synchronization progress. Because the data synchronization operation is idempotent, even after the data synchronization, the data synchronization progress field is not successfully updated due to downtime, resulting in the synchronization of the same data during data recovery. Data consistency can still be guaranteed.

8. Experiment

We compare SG-ColBase with MySQL and HBase. SG-ColBase and MySQL run on Windows PC with i7-9750H CPU / 16G memory. HBase runs on AliCloud ESC with five regional servers.

In order to facilitate the experiment, the test table only has two fields: name and age, which are fixed length strings and four byte integers. And create an index in the age field.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

MySQL additionally creates a self increasing primary key field. Since SG-ColBase has its own logical primary key, it does not need to be created.

When SG-ColBase creates a table, each column of data is stored in 1000 segments, and the auxiliary resources of the field have 10 and 50 partitions respectively.

8.1. Stress Test for Transaction

Test scenario: Each transaction inserts 100 rows of random data into the database, totaling 10000 transactions.

Because MySQL's transaction features are relatively complete. This experiment uses MySQL compare with SG-ColBase.

Table 1. Transaction test results.

Database	1 Thread	4 Thread	16 Thread
SG-ColBase	238	353	436
MySQL	119	104	87

[1] The data displayed in cells is TPS (transactions per second).

As a whole, there is a TPS gap between MySQL and SG-ColBase. MySQL and SG-ColBase transactions adopt asynchronous write mechanism. However, because MySQL uses B+ Tree indexes, when inserting random data asynchronously, it will cause random disk writes, or even disk data moves. This causes asynchronous write threads to occupy disk time for a long time, affecting transaction submission. SG-ColBase uses LSM-tree indexes, which are written incrementally in sequence. It greatly reduces the consumption of asynchronous disk time and the impact on transaction submission. MySQL SQL parsing is also the reason for the gap.

The TPS of SG-ColBase increases with the increase of the number of threads. This is because the index files are written in batches in order at an extremely fast speed, making the CPU the bound of TPS. And the data insertion mechanism randomly selects data segments, reducing the competition of resources between threads. Therefore, the increase of threads improves the TPS.

The TPS of MySQL decreases with the increase of the number of threads. This is because the random writing speed of index files is slow, and disk IO is the bound of transactions. The increase in the number of threads will block due to data competition.

8.2. Stress Test for Query

Test scenario: In the test table of 10000k pieces of data, the test randomly queries 1000k times according to the age field (indexed). In order to simulate real business scenarios, some query databases do not exist.

In this experiment, HBase was compared with SG-ColBase.

Table 2. Query test results.

Database	1 Thread	4 Thread	16 Thread
SG-ColBase	114942	256410	154320
HBase	61266	142654	265394

[2] The data displayed in cells is QPS(queries per second).

The QPS of SG-ColBase is significantly ahead of HBase when it is single threaded. This is mainly because SG-ColBase uses the data architecture of the logical primary key, which saves a lot of memory and allows the memory to hold more effective data. The cluster network IO of HBase is also the cause of the gap. This lead continues at 4 threads. When the number of threads reaches 16, the QPS of SG-ColBase starts to decline. The reason is that the system cannot allocate enough threads for the process, resulting in a large number of thread switches. HBase benefits from the distributed cluster architecture, and

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2022 doi:10.20944/preprints202211.0220.v2

queries are allocated to multiple machines. The future distributed SG-ColBase will improve this problem.

8.3. Stress Test for Big data Writing

Test scenario: Single thread writes a large amount of random data to the database.

Table 2. Writing test results.

Database	150k Data	15000k Data
SG-ColBase	7	603
HBase	23	417

[3] The data displayed in cells is time required (seconds).

Under the scale of 150k data, the write performance of SG-ColBase is significantly better than that of HBase. In addition to the network overhead under the distributed architecture, SG-ColBase's index file segmentation technology has also played a great role. In the scenario where 15000k pieces of data are written, HBase is faster than SG-ColBase. It's because the distributed architecture allocates the disk write pressure during large-scale writes.

9. Conclusions

With the rapid development of the Internet industry. In order to save operating costs, we hope to reduce the use of heterogeneous databases. It is hoped that one database can solve both OLTP and OLAP services. This report shows our self-developed solutions.

In terms of data storage structure, we innovatively use logical keys and store the data in an unordered manner by column segment. To achieve the purpose of improving the proportion of effective data in memory and accelerating the retrieval and analysis of large quantities of data. In terms of index storage structure, we choose to use the combination of skip list and LSM-tree, and store them in segments on the disk. It improves the speed of asynchronous database writing and reduces the occupation of disk. And use the strategy of regular merging to improve the loading speed.

In terms of improving the running speed. We have developed a highly concurrent and scalable resource cache pool in the process to reduce disk access and system calls. Implemented an in-process buddy memory pool to reduce system calls for memory. A non blocking log cache is developed to eliminate the blocking of transaction submission during asynchronous writing. A snapshot data maintenance and read mechanism is developed to make read and write parallel. Implemented the Bloom filter to reduce the invalid execution of the database.

According to the experimental results, the performance of SG ColBase in some scenarios exceeds that of MySQL and HBase. However, in terms of reading and writing large amounts of data, it is limited to the current stand-alone architecture and still slightly inferior to HBase. Future distributed versions will address this issue.

Access Project

SG-ColBase: 一种高效的列式数据库 (gitee.com)

Acknowledgement

Thanks to Qianqian Xiong (Shandong University) for assisting in report writings.

References

1. Hongxue Shen, Yanpu Guo. New Trends in Distributed Database Research[J]. Software, 2021(010):042.
2. Katarina Grolinger; Wilson A Higashino; Abhinav Tiwari; Miriam Am Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. Journal of Cloud Computing 2013, 2, 22.
3. Yuxing Ma. Redis Database Feature Analysis[J]. Internet of Things technology, 2015, 5(3):2.
4. Muhammad Umair Hassan; Irfan Yaqoob; Sidra Zulfiqar; Ibrahim A. Hameed. A Comprehensive Study of HBase Storage Architecture—A Systematic Literature Review. Symmetry 2021, 13, 109.
5. Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. Proceedings of the 2009 ACM SIGMOD International Conference on Management of data 2009, 1-2.
6. Karan B Maniar; Chintan B Khatri. Data Science: Bigtable, MapReduce and Google File System. International Journal of Computer Trends and Technology 2014, 16, 115 -118.
7. Junyao Yang; Yuchen Wang; Zhenlin Wang. Efficient Modeling of Random Sampling-Based LRU. 50th International Conference on Parallel Processing 2021, 1.
8. Ming Yuan; Shulin Yang; Mengdie Gu; Huijie Gu. Microservice: dynamic load balancing strategy based on consistent hashing. International Conference on Electronic Information Engineering and Computer Communication (EIECC 2021) 2022, 12172, 27 - 32.
9. On, ST, Hu, et al. Flash-Optimized B plus -Tree[J]. J COMPUT TECHNOL, 2010.
10. Lu K , Zhao N , Wan J , et al. TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning[J]. IEEE Transactions on Parallel and Distributed Systems: A Publication of the IEEE Computer Society, 2022(33-8).
11. Aduboateng M G , Anyanwu M N . Skip List: Implementation, Optimization and Web Search[J]. 2015.
12. Edans F.O. Sandes; George L.M. Teodoro; Alba C.M.A. Melo. Bitmap filter: Speeding up exact set similarity joins with bitwise operations. Information Systems 2019, 88, 101449.
13. Kousalya, A, Radhakrishnan, et al. K-Level with Buddy Memory Allocation (BMA) Approach for Parallel Workload Scheduling[J]. Wireless personal communications: An Internaional Journal, 2017.
14. Srinivas J , Gowtham Y , Amith S S , et al. Leaky Bucket based congestion control in Wireless Sensor Networks[C]// 2018 Tenth International Conference on Advanced Computing (ICoAC). 2018.