A Low Latency Cache for Cloud RDBMs

Guohai Zhang, HashData Ltd., China
Xin Tang, HashData Ltd., China ✉
Qingchen Chang, HashData Ltd., China
Huanchen Zhang, Tsinghua University, China
Kai Hwang, CUHK(SZ), China
Yuesen Li, Chinatelecom Cloud, China
Runhuai Huang, Chinatelecom Cloud, China
Teng Wang, Chinatelecom Cloud, China
Wusheng Zhang, Chinatelecom Cloud, China
Ming Zhang, Chinatelecom Cloud, China
Qingchun Chen, Chinatelecom Cloud, China
Xiaodong Hou, Chinatelecom Cloud, China
Qian Wang, Chinatelecom Cloud, China

ABSTRACT

In contemporary cloud-based analytical databases, the adoption of a disaggregated storage model is a prevalent trend. This model allows the elastic compute layer to access data stored remotely in block-oriented columnar formats in cloud storage. However, the high latency and limited bandwidth associated with remote storage, as well as the limited capacity of local storage, pose significant challenges. Consequently, the imperative of caching data within the compute nodes has gained significant attention, sparking a renewed interest in caching methodologies for enhancing analytical processes. While existing caching solutions focus on improving bandwidth based on file or block-level caching with an average file or block size of tens of MBs, many analytical database scenarios require handling small files (one table consisting of thousands of 10 KB small files), low latency (response time of 100 ms), and high concurrency (hundreds of simultaneous accesses). In this paper, we introduce a new caching system, Gopher, which effectively addresses these challenges. It empowers storage-disaggregated cloud databases to deliver performance comparable to MPP databases while also exploiting the benefits of elastic horizontal scaling.

VLDB Workshop Reference Format:

Guohai Zhang, Xin Tang, Qingchen Chang, Huanchen Zhang, Kai Hwang, Yusen Li, Runhuai Huang, Teng Wang, Wusheng Zhang, Ming Zhang, Qingchun Chen, Xiaodong Hou, Qian Wang. A Low Latency Cache for Cloud RDBMs. VLDB 2025 Workshop: DATAI.

1 INTRODUCTION

In the era of cloud computing, the evolution of cloud-based analytical databases has led to the widespread adoption of a disaggregated storage model [20, 21, 23, 53, 54]. Driven by its flexibility and cost-effectiveness, this architecture decouples the compute and storage layers, enabling elastic compute layers to access remotely-stored data within cloud storage services. Major cloud databases and big data systems like Snowflake [20], AWS Redshift [21], Spark [24], and Presto [25] already support direct querying of cloud storage like AWS S3 [17] and Azure Blob [69]. However, this architecture introduces challenges due to high latency and bandwidth limitations inherent to remote storage, as well as constraints posed by local storage capacity.

Consequently, cache systems have garnered significant attention. Despite the performance gains achieved through conventional caching methods, they struggle to address two challenges effectively.

1) While traditional caching technologies [1, 2, 3, 34, 35, 36, 38, 39] can effectively improve read speeds, they often provide limited benefits for write operations. This is due to the network's inherently higher latency and lower throughput compared to local disk access. Such sluggish write speeds can result in write-intensive processes suffering from substantial delays and severely degrade the efficiency of job pipelines, where the successful completion of one task is contingent upon the quick and reliable processing of its output by the subsequent task.

2) Although certain caching systems [8, 9, 14] are designed to enhance both read and write throughput using file or block-level caching, with typical file or block sizes in the tens of megabytes, many cloud-based database environments present unique challenges. These scenarios often involve dealing with copious amounts of minuscule files. For example, a single database table might contain thousands of 10 KB files. This imposes strict latency thresholds of about 100 milliseconds. Additionally, there is a need to accommodate hundreds of concurrent access requests. To meet these demands, caching strategies must evolve to efficiently handle the granular nature of small files while maintaining the speed and scalability required for demanding cloud database workloads.

In this paper, we present Gopher, a novel caching system that overcomes the aforementioned challenges. Gopher is a distributed caching mechanism designed to handle high concurrency and deliver low-latency access to files of all sizes. Deployed at each node within the data warehouse's computational layer, Gopher is engineered to accelerate I/O operations. To circumvent the potential limitations of network communication within clusters, which are common in centralized master-slave distributed systems, Gopher adopts a decentralized architecture. Each cache instance operates as an autonomous client/server service, functioning as a local file system equipped with a two-tier cache for data warehouse computing services at the node level.

✉ Xin Tang is the corresponding author.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

Gopher stands out as a high-performance, highly available caching system, offering an architecture that supports file prefetching, batch read operations, shared memory, optimized asynchronous writes, concurrent multi-read capabilities, as well as a suite of advanced features including merging for small files and rapid cache reconstruction during cluster horizontal scaling. These features collectively reduce the number of I/O operations and memory copies, facilitating efficient concurrent analytics processing and seamless scaling. In our experiments, we observe that Gopher can significantly enhance the performance of cloud databases by 4x. It accomplishes this while offering query times that are comparable to those of MPP databases and enjoying the advantages of elastic horizontal scaling.

The rest of the paper is organized as follows. Section 2 introduces the background and the motivation of the system design. Section 3 and 4 elaborate on the system architecture, API, and caching optimizations. Section 5 presents and discusses experiments and Section 6 reviews related works. The conclusion is drawn in Section 7.

2 BACKGROUND

This section introduces the architecture of Teleadb for AnalyticDB (TeleDB-ADB) cloud-native data warehouse and describes our target workloads. It provides background information that motivates the Gopher solution, a low latency cache in the compute nodes.

2.1 TeleDB-ADB Cloud-Native Data Warehouse

TeleDB-ADB is a data warehouse to support both SQL and machine learning analytical tasks in the cloud. The system adopts a cloud-native disaggregated design in which the compute, storage and metadata services are managed in separated clusters to enable high elasticity, high availability, and high reliability. Figure 1 illustrates the TeleDB-ADB architecture.

TeleDB-ADB's storage layer supports data persistency and retrieval in object storage, UnionStore, and external big data storage [18, 19]. TeleDB-ADB-managed object storage is the default storage pool for user table data, temporary data, and query results. It is elastic and cost-effective, providing 99.999999999% durability and 99.99% availability SLA, which makes it a natural choice as the main storage engine for massive data. UnionStore is the storage engine to support hybrid transactional and analytical processing. It consists of a WAL service and Page service, which persist and replay redo logs to support transactional data visits and time travel. TeleDB-ADB also supports open lakehouse and can visit external data in HDFS managed by Apache Hive [70] and Apache Spark [24]. The system uses foreign data wrapper to access the Hive, Hudi [71], or Iceberg [72] metadata and then performs data read / write operations in HDFS [19] or other external storage.

The metadata layer is an independent cluster that manages and serves various metadata for the compute clusters. It consists of a coordination service, a metadata service, and a metadata storage service. The coordination service is an etcd [73] cluster that discovers and allocates the metadata serving. The metadata service is a cluster of stateless services that provide metadata access, privilege control, lock management, and distributed transactions,

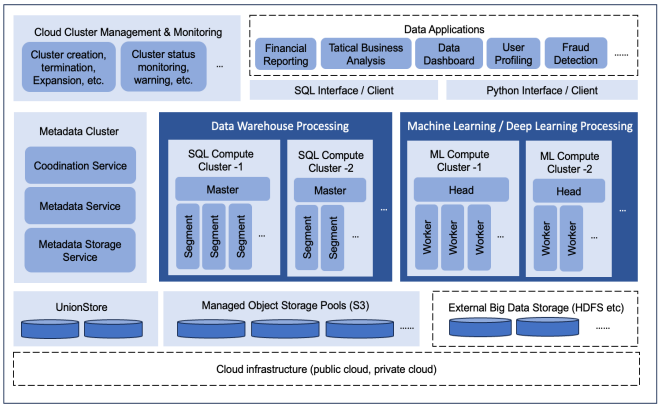

Figure 1: TeleDB-ADB Cloud Database Architecture.

etc. The metadata storage service is a distributed key-value storage engine that stores various metadata such as mappings between tables and storage objects, data dictionary, WAL logs and indexes, etc. When a metadata request arrives, the coordination service allocates a metadata service node to handle this request. This node retrieves the requested metadata from the metadata storage service and returns them.

The compute layer is the core of the system and can provide multiple compute clusters to perform data warehouse or machine learning analytical tasks in scale. Each compute cluster can be created using different hardware and configurations on demand and its resources are completely isolated from other clusters. Currently, the compute layer can support up to 10,000 computer clusters concurrently. Each compute cluster has a coordinator node (also called master or head) as the query entrance and coordinator as well as multiple worker nodes (also called segment) which perform the heavy-lifting computation. When an analytical task arrives, the coordinator node generates a distributed execution plan, driving its worker nodes to complete the task.

We initiated the TeleDB-ADB system implementation in 2016 and it has been deployed to production since 2018. Our largest production deployment entails more than 148 compute clusters and 30,000 VM nodes. It adeptly oversees in excess of one million tables and 19 PB data, processing more than 10 million queries and jobs daily, which is one of the largest cloud data warehouse deployments worldwide. Our customer benefits substantially from the inherent attributes of our cloud-native architecture, witnessing a reduction of 20% in duplicated jobs, 51% in redundant data, and an optimization of over 30% in hardware resource utilization. While deploying the system to a large production environment has given us a deep experience of the benefits of a cloud-native architecture, it has also revealed the urgent need for a new caching system for cloud analytical database workloads. These requirements are elaborated in the next section.

2.2 Target Workload Properties

The integration of a disaggregated storage model demonstrates substantial elasticity and scalability advantages yet brings high latency and restricted bandwidth associated with remote storage to the system. In response to these inherent challenges, we implement a cache system in the compute layer to manage read and write

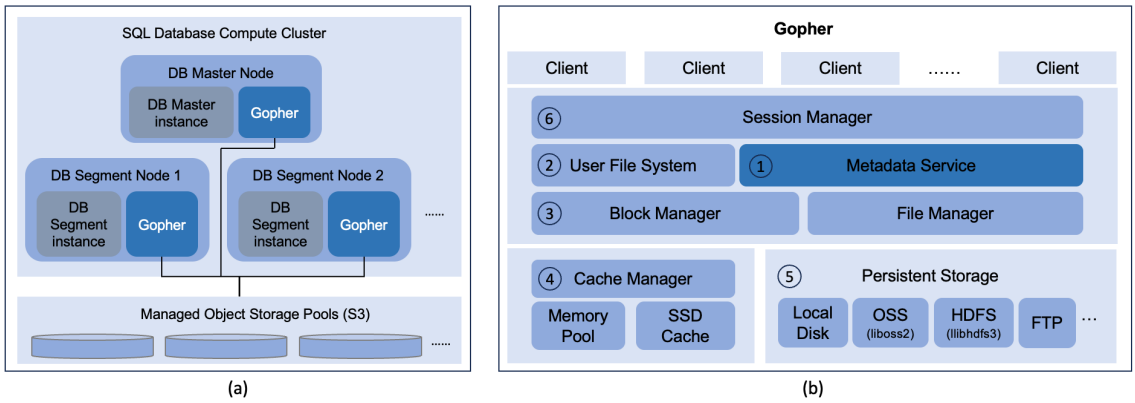

Figure 2: Gopher Cache System Architecture.

operations. This cache is strategically designed to optimize analytical workloads for data warehouse, data lake, and AI/ML processing, which have the following properties:

- Massive small file access: In addition to a large file sequential access and large file random access, millions of 16 KB files are generated in large-scale production environments, which need to be managed carefully to ease disk and network I/O overheads.
- High concurrency: The cache needs to support large volume of concurrent jobs. In one of our large production environments, the system processes over 9 million queries and 1 million ETL jobs daily. During its peak business hours, the system faces the demand of over 8000 concurrent queries, imposing significant strain on the cache (500 concurrent jobs per node) to handle data read/write gracefully with fixed I/O capacity.
- Low latency: Latency stands as a pivotal factor influencing real-time analysis, fast data retrieval, AI/ML processing, and other time-sensitive jobs. These tasks expect query responses like MPP processing, which requires the cache to significantly reduce overall response delays, aiming for a substantial decrease in latency by 2x and more. Such enhancements empower expedited decision-making processes and facilitate seamless execution of real-time analytics, thereby elevating operational efficiency and responsiveness in various applications.
- Seamless access to diverse storage infrastructure: Large enterprise users store data in diverse infrastructure such as local and remote HDFS, S3, and FTP storage, etc to utilize historical IT investments. Building modern analytic and AI pipelines requires the cache being able to flexibly exchange data with different storage systems, minimizing data silos and redundant.
- Shared memory support for query operator pipelining: Shared memory becomes critical when dealing with large amounts of data in parallel operations. Supporting data pipelines through shared memory enables rapid transmission and processing of data at different processing stages, reducing unnecessary data copying and transmission time to the local or remote storage, thereby improving processing efficiency and overall system performance.

3 GOPHER DESIGN OVERVIEW

In this section, we present a comprehensive overview of the Gopher system, elucidating its architecture, API functionalities, and providing practical examples to illustrate its application and usage.

3.1 System Architecture

Gopher is a distributed cache system that supports high concurrent and low latency access to massive small and large files. It is deployed to each node at the data warehouse computer layer to accelerate computing I/O access (Figure 2-a). In order to avoid potential intra-cluster network communication constraints which we have observed in other centralized master-slave distributed cache systems, the cache adopts a decentralized design in which each cache instance is a self-contained client/server service, acting as a local file system with two-level caches for data warehouse computing services at the node. Figure 2-b illustrates the main components of Gopher.

1) The metadata service is the core of the cache system. It manages metadata such as task status, file name, or block ID and interfaces with other modules to facilitate access data for clients. To gracefully handle massive I/O requests, the metadata service employs a working thread pool which consists of separated child thread pools. These pools handle long-executing commands and manage disk space asynchronously, ensuring efficient resource utilization.

2) The user file system stores destination storage information such as target file systems, ports, and buckets, etc.

3) The block manager and file manager offer data access for blocks and files, respectively. They utilize stream reads and writes to process large files in a small memory buffer, improving I/O efficiency and reducing memory usage.

4) The cache manager implements a two-level cache, a memory pool as well as an SSD cache, to productively utilize the limited cache space and improve data access speed. The memory pool supports zero-copy processing, facilitating shared memory access to enhance runtime efficiency. The SSD cache stores data swapped out by the memory pool. As a non-volatile storage medium, the

```
1.  /* a. User Operations */
2.  gopherFS connect(gopherConfig configure);
3.  int disconnect(gopherFS fs);
4.
5.  /* b. File Operations */
6.  gopherFile openFile(gopherFS fs, const char *path,
7.        int flag, toOffset block_size);
8.  int closeFile(gopherFS fs, gopherFile file,
9.        bool sync);
10. tSize read(gopherFS fs, gopherFile file,
11.       void *buffer, tSize length);
12. tSize write(gopherFS fs, gopherFile file,
13.       const void *buffer, tSize length);
14.
15. /* c. Batch Operations */
16. int prefetch(gopherFS fs, int num,
17.       const char **fileList);
18. int batchRead(gopherFS fs, int num,
19.       const char **fileList, void *buffer);
20.
21. /* d. Memory Object Operations */
22. int createObject(gopherFS fs, const char* object_id,
23.       int64_t data_size, char** data);
24. int sealObject(gopherFS fs, const char* object_id);
25. int getObject(gopherFS fs, const char* object_id,
26.       int64_t timeout_ms, const char** data);
27. int releaseObject(gopherFS fs, const char* object_id);
28. int deleteObject(gopherFS fs, const char* object_id);
```

Figure 3: Gopher Client Interface Examples.

SSD cache can be used to restore computing states when the node accidentally crashes.

5) The persistent storage module incorporates network transmission libraries such as liboss2, libhdfs3 and libftp to complete network connection and transmission tasks to S3 object storage, HDFS and local storage, an expanding list of storage options.

6) The session manager utilizes epoll multiplexing to handle connections from multiple clients simultaneously. The client implementation involves the cache system APIs as well as stream objects such as FileInStream and File OutStream to stream read and write data efficiently. The flatbuf protocol is employed to provide high-performance and reusable serialization and deserialization.

As a high-performance and highly available caching system, Gopher supports multiple key features such as file prefetching, batch read, shared memory, asynchronous write optimization, concurrent multiple reads, and small file memory merging, etc. These features can practically reduce I/O operations and memory copies, supporting efficient concurrent processing and rapid scaling-out. Detailed cache optimizations are described in Section 3.3 and 4.

3.2 API

Gopher provides a comprehensive set of client interfaces to manage files and directories, ranging from basic file operations to advanced cache management. The interfaces include C, C++, and Java implementation for flexible invocations. C is the primary interface as it is convenient for direct database calls. Figure 3 shows common interfaces in C for different types of operations.

a) User operations are client interfaces to establish and terminate a connection with Gopher and the remote persistent storage. For instance, the *connect()* method (line 2) passes the destination persistent storage information in parameter configure to ask Gopher to set up a connection with the remote storage system and returns a handle to a file system managed by Gopher.

b) File operations are standard file operations that open, read, write, and close files. Parameter sync in the *closeFile()* method (line 8-9) is to specify whether to synchronize the file to the remote storage immediately.

c) Batch operations are to accelerate reading files smaller than 8 MB. For instance, the *prefetch()* method (line 16-17) instructs Gopher to prefetch files specified in *fileList*. The *batchRead()* method (line 18-19) multiple files into the buffer which is normally set to 8 MB.

d) Memory object operations are methods to instruct storing intermediate results in the memory directly instead of writing to temporary files. These I/O optimizations are discussed in detail in Section 4.3.

In addition to the operations mentioned above, Gopher also provides interfaces to support administrative tasks, UUID handling, and other functionalities, servicing as a comprehensive file system for clients. These interfaces have not been detailed here because they are less directly related to the subject of caching.

3.3 Examples

This section uses an example of reading and writing a sequence of files with various sizes from an object storage service (OSS) to illustrate the cache system's I/O process. This process supports reads, synchronous writes, and asynchronous writes.

The client calls *connect()* and *disconnect()* to establish and terminate a connection with the cache system. Once receiving the OSS connection information from the client, the cache system persists it in the user file system, initializing a working pool at the metadata service to start the interaction with OSS using libraries in the persistent storage module. These allocated resources will be freed when the I/O process completes, and the connection terminates.

3.3.1 Reads

There is a strong correlation between file size and I/O efficiency, and we have observed that a database process reaches good input utilization when it is given a buffer of at least 1 MB or a whole data block. Hence, we first sort the list of files to read based on their file sizes and apply separate read policies according to their sizes.

1) Files less than 1MB. Files smaller than 1 MB are all read in batch. The client calls *prefetch()* to request the list of files and then calls *batchRead()* to read the file data. On the other hand, upon receiving the list of small files, the cache system allocates a 16 MB buffer at its two-level cache, reading files from OSS to the buffer. When either the 16 MB buffer is filled or the batch accumulates 100 small files, a new stream of small files will be returned to the client.

2) Files between 1 MB and 8 MB. Unlike files smaller than 1 MB, the process of transferring files between 1 MB and 8 MB from OSS to the cache and then from the cache to the client occurs asynchronously. As soon as a file is cached in the buffer, a fresh stream is initiated to the client, even as the cache continues to retrieve additional files from OSS.

3) Files more than 8MB. Files exceeding 8 MB in size are moved from OSS to the cache and then served to the client in 8 MB blocks in turn. These data blocks can also be prefetched when we set a block-prefetch configuration flag in Gopher. When the client issues a *read()* request specifying the file's offset and the desired data length, the cache system verifies if the requested block is already stored. If not, it initiates the download of subsequent blocks in the file and forwards the block to the client once it has been retrieved.

3.3.2 Synchronous Writes

To perform synchronous writes, the client first calls *openFile()* to create and specify the block size of the file in the cache. Then it invokes *write()* to ask the cache system to allocate a memory space with the given block size, writing file contents to the memory. When the block is filled, the cache system allocates a new block in memory to the client to continue writing and send the filled block to OSS in the background. When the file is completely written, the client calls *closeFile()* to terminate the write process. As writing data to OSS takes much longer than writing to the local cache, the working thread pool supports simultaneously writing multiple blocks to OSS in parallel to accelerate writing.

3.3.3 Asynchronous Writes

The asynchronous write feature can further expedite the writing process of sequential files by only verifying the file writes at the time of transaction completion. For instance, the database performs 10 inserts during a transaction and instructs the client to write 10 files. When the client calls the *closeFile()* after writing the first file to the cache, the cache system immediately returns write completion, allowing the client to start writing the next file even though writing to OSS is still in progress. On the server side, Gopher writes and aggregates these files in the memory pool and asynchronously sends them to the persistent storage in batch in parallel. Write completion will be verified when the transaction ends and *summit()* is called. If any file write is unsuccessful, the entire transaction will be rolled back. Otherwise, the database will proceed to the next transaction. The asynchronous writes feature transforms a sequential database write process to writing cached blocks to OSS in parallel, thereby substantially accelerating the write process.

4 CACHING OPTMIZATIONS

This section discusses caching optimizations in addition to file prefetching and asynchronous writes covered in section 3, such as small-file merging and rapid cache reconstruction. We design and implement these innovative features in Gopher, enhancing its functionality as a high-performance compute layer cache within an elastic cloud environment.

4.1 Merging Small Files into Blocks

Large-scale analytical computing clusters inevitably deal with massive numbers of small files. As the cluster grows and the number of nodes increases, the same data is divided into more segments, resulting in smaller segments that are then distributed evenly across the compute nodes. Consequently, the files processed on each node continue to decrease in size. In some of our large production environments, which house over 10,000 virtual machines and handle several million queries daily, we have identified millions of files less than 16KB. These files generate significant I/O pressure, creating a bottleneck that impacts overall performance. To address this challenge, we have developed a small file merging and caching strategy to improve the system's efficiency.

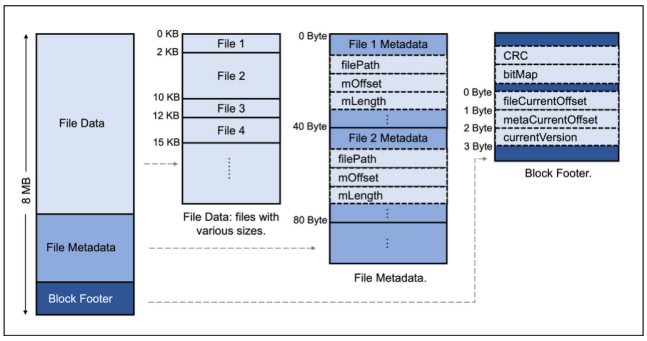

Figure 4: Cached Data Block for Small Files.

The mechanism of merging small files into blocks operates as follows. Files smaller than 16KB are identified and distinguished from other files, encapsulated into 8MB cached data blocks. The data block structure is self-descriptive, containing metadata such as file names, start offsets, file lengths and bitmap locations at the block footer. Interfaces for block-level read, write, delete, and lookup are provided for rapid data access. By accessing just a single data block, the client gains all the necessary information to access 500 files or more, resulting in a highly efficient process.

In addition, data locality, multi-threading, and two-level caches are utilized to enhance data access optimization.

1) To better support range reads, files are aggregated based on data locality rather than random distribution. Files that are frequently accessed together or belong to the same table are arranged into the same block as much as possible. When these files need to be read, the number of data blocks to load reduces significantly.

2) For file updates and deletes, we utilize an append-only method that allows for non-blocking operations. Background threads in the working pool will batch-process the cleanup of outdated files asynchronously.

3) The local SSD pool is utilized as a large volume L2 cache. The cache manager first assigns memory space in the memory pool to support fast data block access. When the memory usage reaches the upper limit, the cache manager moves the least recently used blocks to the local SSD pool instead of destroying them. When files in these blocks are needed again, we can reload the blocks to the memory rather than reconstructing them via fetching hundreds of files from the remote storage. By combining small files into data blocks and employing the optimization measures above within the cache, the utilization of I/O resources is effectively balanced, substantially reducing high I/O issues and thereby enhancing the stability and performance of the system.

4.2 Rapid Cache Reconstruction in Cluster Horizontal Scaling

Horizontal scaling is a common occurrence for analytical systems in massive production environment, which may significantly affect the computing cluster performance. As the priorities of business requests change over the course of the day, different compute clusters scale out and in accordingly. For example, during daytime business hours, interactive queries are frequent and server resources are primarily allocated to clusters that support those queries. At night, when the company needs to consolidate and analyze daily business data, computing resources are shifted to clusters that support batch jobs. Driven by dynamic changes in business requirements, compute clusters are scaled out and in multiple times a day to adapt to workload changes. However, the compute nodes that get assigned new data partitions after the horizontal scaling often become the shortboards of a barrel due to the nature of missing or mismatch of local caches. We have observed that, in large-scale production environments, retrieving massive files from the remote persistent storage to reconstruct the cache can take anywhere from half an hour to several hours, severely impacting the cluster. To address this challenge, we developed a feature that allows the cache to be rapidly reconstructed after the cluster horizontally scales.

The mechanism of rapid cache reconstruction is as follows. Data in the persistent storage is evenly distributed among nodes in a computer cluster in the unit of data logical partitions according to a consistent hash algorithm. When a scaling occurs, the mapping between the data and the compute nodes is changed. For instance, if a cluster of 128 compute nodes is scaled out to 256 nodes, the mapping between the data in the persistent storage and the compute nodes changes from 1:128 to 1:256. The node that newly joins the cluster can use the consistent hash algorithm to calculate and identify the node where its data logical partitions mapped, sending cache synchronization requests to it. The two nodes transfer the cached data in the unit of a data block which is introduced in 4.1 and can encapsulate hundreds of small files through a multi-thread communication module in the cache system. File MD5's are examined to ensure data consistency. If there is an exception in the transmission process, the communication module will restart the cache synchronization until all the requested cache data is synchronized successfully. At this time, the cache system in the old node will mark the transferred cached blocks invalid, and its background thread will perform cleanup, releasing local cache resources to allocate for new tasks. Since it is a many-to-many communication process among nodes within the same cluster, the cache reconstruction is highly efficient and usually completes in seconds.

4.3 Memory-First Intermediate Result Sharing

The execution plans of analytical query jobs are usually complex, involving sequential executions of multiple operators. The output of one operator is the input of the next one. Substantial intermediate results are generated and saved in this process. In traditional database processing, these intermediate results are typically stored as temporary files cached on disk, which may encounter several potential bottlenecks:

1) Disk caching requires data to be written from memory to disk and then read from disk to memory, a process that involves disk I/O and memory copying, which has a significant impact on performance.

2) When the upstream operator is writing a temporary file, the downstream operator needs to wait until the file is completely written, which will hinder the execution of the downstream job, wasting the CPU and other system resources allocated to the downstream operator.

3) In the cloud database scenario, as the number of compute nodes increases, the data partition range controlled by each node is narrower, resulting in the temporary files saved by each node being smaller in size but greater in number. This creates greater pressure on file metadata management, serialization, and deserialization.

In an effort to support intermediate results between different operators in the execution plan effectively, we've introduced a new option to store intermediate results as in-memory objects in addition to saving them as temporary files. By prioritizing in-memory objects for caching, we can reduce the overhead associated with disk writes and memory copies, enhance collaboration between upstream producers and downstream consumer operators, and reduce the costs involved in managing many small files.

Figure 3.d illustrates the client interfaces to manipulate an in-memory object that supports zero-copy write-once-read-many, allowing multiple operators to perform efficient concurrent reads. Line 22-23 is the createObject() method which allocates memory resource and creates an in-memory object, returning the object ID and memory address (specified in the parameter data) to the client. Once obtaining the memory address, the upstream operator writes to the memory. Upon completion of writing, it invokes the sealObject() method (line 24) as a signal to the cache system that the in-memory object has been fully written to and ready for reading. On the other hand, the downstream operators invoke the getObject() method (line 25-26) to read data in the in-memory object and call the releaseObject() method (line 27) upon completion of reading. When all the downstream operators finish reading, the deleteObject() method (line 28) is called by the client or the cache background process to release the memory resources.

In a similar vein, temporary files are written and read in the same way as normal files. Clients use file operations in Figure 3.b to open and close files, writing and reading data.

This hybrid method to cache intermediate results as in-memory objects or temporary files provides clients the flexibility to choose either the memory pool or the SSD cache pool to transmit intermediate results from one operator to the next efficiently. The client can take the actual data sizes and latency needs into consideration, dynamically opt for the most suitable cache, thereby enhancing I/O resource utilization and overall query performance. Below are the intermediate result sharing strategies in TeleDB-ADB database, which exemplifies the advantage of this hybrid caching technique.

1) When the intermediate result is less than 8MB, the client will create an in-memory object to save the intermediate result. Caching these data in memory uses only limited memory resources as well as avoiding frequent disk I/O's. In addition, since memory objects support zero-copy write-once-read-many operations, downstream operators can directly read memory objects that have been sealed

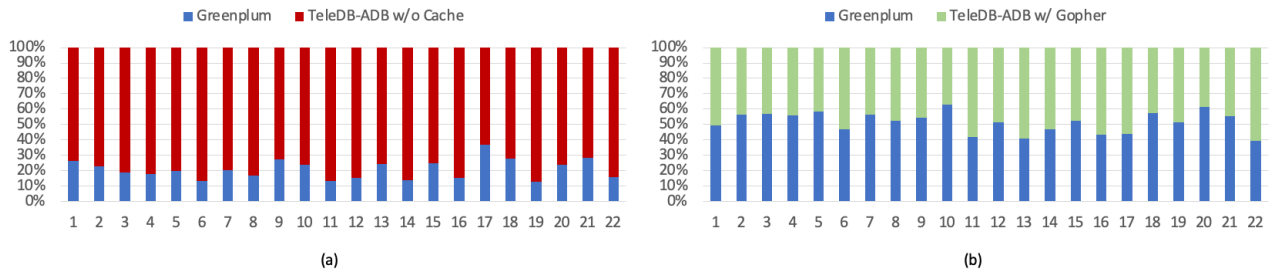

Figure 5: TPC-H Experiment Results for Greenplum vs TeleDB-ADB w/o Cache, and Greenplum vs TeleDB-ADB w/ Gopher.

by the upstream operator without having to apply for additional memory, copy data, and perform serialization or deserialization again. This significantly improves the efficiency of reading intermediate results.

2) When the intermediate result is greater than 8 MB, the client can decide whether to store the data in memory or SSD cache pool based on whether the downstream operators consume the intermediate result immediately. 2-a) Intermediate results need to be consumed immediately. In this case, the client uses the in-memory object interface to cache intermediate results, writing to multiple in-memory objects. Instead of waiting until all the data is written into the temporary file, the downstream operator can start reading the data once any in-memory object is sealed without additional disk I/O or memory copy. Collaboration between operators and system utilization are both more efficient. 2-b) Intermediate results are not immediately consumed. In this case, the client invokes the file operation interfaces to write the intermediate results to the SSD cache directly, which reduces the number of data copying, faster than first writing to the memory and then placing it on the disk.

5 EXPERIMENTAL EVALUATIONS

In this section, we evaluated Gopher's general performance and various features using a collection of experiments that employ the TPC-H benchmark and real-world data workloads. The experiments demonstrate that Gopher excels as a cache system, delivering low latency, high concurrency, and robust throughput when handling data warehouse and machine learning system files of various sizes in the elastic cloud environment.

Unless specified, the experimental configuration is as follows: the TeleDB-ADB database is established to include a compute cluster and a storage cluster. The compute cluster is composed of 1 master node and 8 segmented nodes, with a total local storage capacity of 100 GB. Each node is equipped with 8 core CPUs and 16GB memory resources. For the Gopher caching system, 8 CPU and 16 GB memory are assigned at each node for the cache system. The storage cluster supports s3 I/O interfaces, responsible for persisting the database data. Servers are connected with 30Gbps network.

5.1 End-to-End Evaluation

In this section, we evaluate Gopher's performance in managing database and machine learning workloads through the TPC-H, slowly changing dimensions, and unstructured data file read/write tests.

5.1.1 TPC-H

We utilize the TPC-H 100GB benchmark to evaluate Gopher's efficiency as a cloud database cache. This standard data warehouse workload generates files in the file size range of 0 to 453 MB for Gopher to process. Three experiment control groups are set up to evaluate the results. In the first and second group, the TeleDB-ADB cloud-native database executes the TPC-H benchmark, with and without using Gopher, respectively. In the third group, a Greenplum database (version 6.2) which employs a traditional MPP architecture that stores data locally is configured with the same CPU, memory, and disk resources to execute the TPC-H test. This group acts as a performance baseline for commercial data warehouses.

Figure 5 visualizes the experiment results. The x-axes denote the 22 queries of TPC-H. The y-axis in Figure 5-a denotes the performance comparison between the TeleDB-ADB cloud-native database that does not use its compute node local storage as a compute layer cache (TeleDB-ADB w/o Cache) and the Greenplum MPP database (Greenplum) in percentage. The y-axis in Figure 5-b shows the performance comparison between the TeleDB-ADB database that employs Gopher to cache data in its compute nodes (TeleDB-ADB w/ Gopher) and the Greenplum MPP database (Greenplum), also denoted in percentage.

The test results reveal that Gopher can enhance storage-disaggregated database query performance significantly to process data warehouse workloads and serve enterprise-level critical missions. In the scenario without a compute cache (Figure 5-a), all 22 queries experienced a drastic slowdown, with performance declining by as much as 70.5% to 7x, and an average performance degradation of 4x. The performance downgrade is due to an architecture change where data is now stored in remote storage instead of the local server. As a result, the database must fetch this data via the network, which is slower and less stable compared to reading data from local SSD storage.

In the scenario where Gopher is used, the query runtime for the TeleDB-ADB cloud-native database improves by a factor of 4. As Figure 5-b illustrates, the query times are very similar to those of Greenplum, with individual query difference varying from a decrease of 52.9% to an increase of 41.2%, resulting in an overall difference less than 8.1%. The significant improvement is due to two key factors. First, the overall architecture incorporates a cache layer within the compute nodes, establishing a multi-tier cache

```
1.  Table accounts(
2.    id BIGINT,
3.    balance BIGINT,
4.    start_date DATE,
5.    end_date DATE);
```

Figure 6: Bank Account SCD Table.

system that allows data to be stored in memory, SSD disks, and remote storage. This system facilitates smooth swapping between different storage layers and ensures that data is served to the query job as required, maintaining efficient performance. Second, in addition to this robust architecture, we have implemented a range of caching features, including file pre-fetching, asynchronous writes, and zero-copy. These features further optimize Gopher's resource utilization, throughput, and hit rates. The effectiveness of these features is thoroughly evaluated and discussed in section 5.2.

5.1.2 Slowly Changing Dimensions

To accurately evaluate its performance in real-world business contexts, we use slowly changing dimensions (SCD), a technique commonly employed by banks to manage changes in account transactions, as the database operations to assess Gopher. The streamlined bank transaction scenario we simulate is as follows:

- A regional bank serves 1 million customers, each with a single account.
- The bank's financial transactions encompass four primary operations: account opening, closure, deposit, and withdrawal.
- Every day over a 360-day period:
  - 20,000 new customers open accounts and 20,000 existing customers close theirs.
  - 100,000 customers deposit $100 and another 100,000 withdraw $100.
- To simplify the experiment setup, all accounts are initially credited with sufficient funds and each account performs at most one of the four operations above.

The database system captures these daily business transactions and maintains a detailed historical record. It is vital for the bank's ongoing operations and supports various business needs, such as generating financial reports, managing customer relationships, and conducting audits.

The database technical operations that support this bank transaction business scenario are as described follows:

- A database table, *Accounts*, records the bank's client account information (Figure 6), with each row representing a customer account and a total of 1 million rows.
- Over a 360-day cycle, daily operations include:
  - Insert 20,000 rows.
  - Delete 20,000 rows.
  - Update 200,000 rows, with half increasing the balance by 100 and the other half decreasing the balance by 100.
- All data changes are written to the table as appends. Both old and new records are retained and distinguished by the *start_date* and *end_date* fields as shown in Figure 7.

```
1.  -- Open an account --
2.  INSERT INTO accounts <new account data>;
3.
4.  -- Close an account --
5.  UPDATE accounts
6.  SET end_date = <current date>
7.  WHERE id = <account id to close>;
8.
9.  -- Deposit or withdraw $100 for an account --
10. INSERT INTO accounts
11.   SELECT id,
12.     balance+<dollar change>,
13.     <current date>,
14.     to_date('2999-12-31', 'yyyy-mm-dd')
15.   FROM accounts
16.   WHERE id = <account id to update>;
24.
25. UPDATE accounts
26. SET end_date = <current date>
27. WHERE id = <account id to update>
28.   AND start_date != <current date>
29.   AND end_date = to_date('2999-12-31', 'yyyy-mm-dd');
30.
31. -- Query active accounts
32. SELECT count(*)
33. FROM accounts
34. WHERE end_date = to_date('2999-12-31', 'yyyy-mm-dd');
35.
36. -- Query accounts changed in the previous day
37. SELECT count(*) -- exclude closed accounts
38. FROM accounts
39. WHERE start_date = <previous date>;
```

Figure 7: Bank Account Database Operations.

- To continuously support its mission-critical tasks, the database routinely performs a vacuum every 7 days, cleaning up and optimizing its storage space to ensure it remains in an optimal state.

Similar to 5.1.1, we conduct this experiment with three control groups. First, we use a TeleDB-ADB cloud-native database without employing compute-side storage as a cache (TeleDB-ADB w/o Cache). Second, we utilize a TeleDB-ADB database that employs Gopher as its compute layer cache (TeleDB-ADB w/ Gopher). Finally, we include a Greenplum MPP database (Greenplum). Each of these three groups is configured with identical server resources and runs on 10 concurrent processes.

Figure 8 illustrates the test results. The x-axes denote the 360 days and the y-axes denote the update (line 1-29 in Figure 7) and query (line 31-39 in Figure 7) time, respectively.

The experiment results show that Gopher exhibits excellent performance advantages in complex database operations in the long term. SCD tables, due to their necessity to manage multiple versions of data, are complex to update and can generate a large number of small files, a common issue in database management. For instance, in our simplified bank transaction scenario, executing operations from line 1-29 in Figure 8 every day to update 220,000 rows of data results in hundreds of files less than 8 KB. Despite regular vacuum optimizes storage space, the number of small files remains significant, leading to frequent I/O operations and inefficient use of storage space, which decrease database performance and resource efficiency. Additionally, as the SCD version maintenance continues, the version history becomes extensive and the data volume increases significantly. For instance, the database is needed to handle files over 10GB in size in this

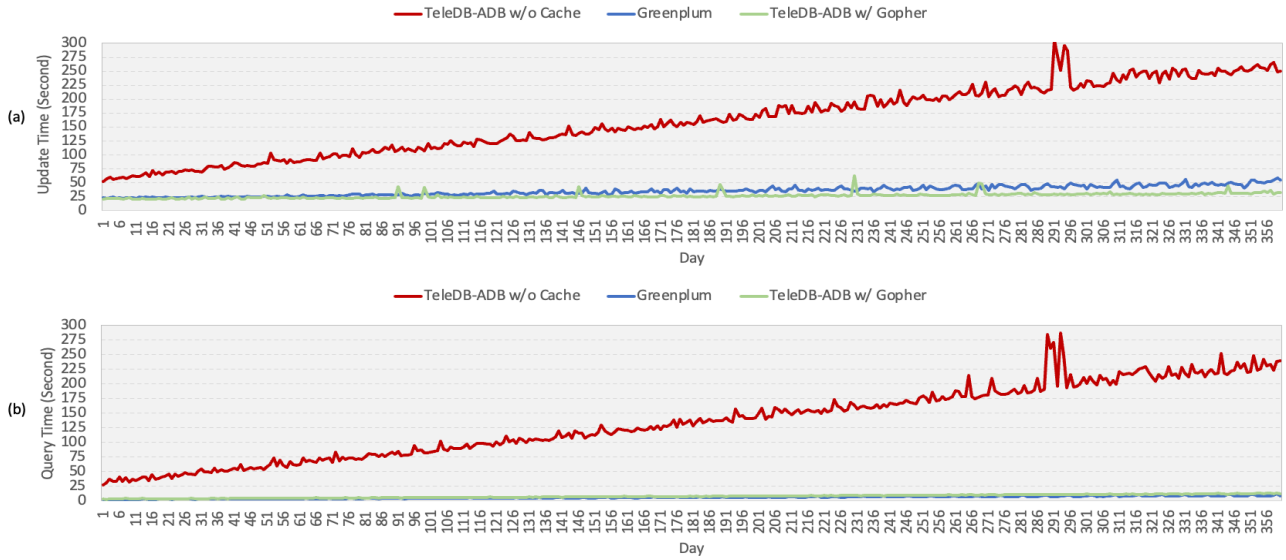

Figure 8: Slowly Changing Dimension Transformation Experiment Results for Greenplum vs TeleDB-ADB w/o Cache vs TeleDB-ADB w/ Gopher.

File Size (KB)	Alluxio Write (MB/second)	Alluxio Read (MB/second)	Gopher Write (MB/second)	Gopher Read (MB/second)	Gopher Write / Alluxio Write	Gopher Read / Alluxio Write
8	1.05	8.45	2.68	55.8	2.55	6.60
32	4.81	34.72	11.16	223.21	2.32	6.43
128	10.42	119.05	35.11	568.18	3.37	4.77
512	48.54	274.12	94.34	1,041.67	1.94	3.80
2,048	78.13	196.08	162.6	1,754.39	2.08	8.95
8,192	107.74	177.78	198.76	1,584.16	1.84	8.91
32,768	103.56	176.8	192.21	317.46	1.86	1.80
131,072	104.73	150.52	237.67	332.55	2.27	2.21
524,288	115.49	163.4	225.05	302.36	1.95	1.85

Table 1: Gopher vs Alluxio Unstructured Data File Read /Write.

experiment, according to our measurements. This massive data volume not only extends query times but also raises cache eviction rates, further diminishing database performance.

Faced with the dual challenges of handling millions of small files and managing large datasets, the database's query and update performance is significantly impacted. However, Gopher has been successful in reducing the negative effects of these challenges. As Figure 8-a illustrates, Greenplum's update time decreases from an average of 22.29 seconds during the first three days to 55.59 seconds during the last three days, showing a 2.49x drop in performance. In comparison, a cloud database with disaggregated storage and no compute cache performs worse, experiencing longer query time and more significant performance degradation over time. The average update time for TeleDB-ADB w/ cache is 55.97 seconds during the first three days, which is 2.5x slower than Greenplum, and it increases to 294.90 seconds during the last three days, which is 5.3x slower than Greenplum, with a performance drop of 4.5x. Conversely, thanks to Gopher's optimizations for handling both small and large files, TeleDB-ADB w/ Gopher performs significantly better. It has an average update time of 21.11 seconds during the first three days and 31.21 seconds during the last three days, with only a 1.48x performance degradation. This is better than Greenplum's performance in both same-day updates and late-stage performance degradation.

5.1.3 Unstructured Data File Reads / Writes

In addition to its support for cloud databases in processing structured data, Gopher can serve as a compute layer cache for machine learning processing in the cloud. It significantly improves the caching of semi-structured and unstructured data throughout the machine learning lifecycle, including training, fine-tuning, and inference phases. To assess Gopher's suitability as a cache for machine learning workloads, we conducted an experiment that focused on its write and read efficiency for unstructured data of diverse sizes.

Two control groups are set up. One group utilizes Gopher while the other uses Alluxio. Both groups are configured with the same hardware resources and preloaded data. To thoroughly evaluate their support for different kinds of data files used in machine learning processes, the test encompasses a wide range of file types, ranging from 8 KB web pages to 512 MB video files.

Table 1 shows the test results. The leftmost column shows the file sizes. The central four columns show the write and read efficiency of Alluxio and Gopher in MB/second, respectively. The two rightmost columns compare the write and read rate of Gopher and Alluxio.

The experimental results indicate that Gopher is an effective compute layer cache when handling unstructured data of varying sizes. With increasing file sizes, both Gopher and Alluxio exhibit improving read and write performance, with Gopher showcasing more pronounced advantages. Specifically, at the time of writing, Gopher outperforms Alluxio by 2x. This superior performance can be attributed to Gopher's fully distributed design, which eliminates the network overhead and the metadata interaction bottleneck associated with Alluxio's Master-Slave architecture. When it comes to read operations, Gopher has demonstrated a significant performance edge, ranging from 3.8x to 8.9x, thanks to its optimized handling of small files. However, as file sizes continue to escalate, the limitations of available memory space necessitate data swapping, causing Gopher's performance lead over Alluxio to diminish to approximately 2x.

5.2 Advanced Caching Feature Experiments

This section evaluates and discusses Gopher's advanced features such as file prefetching, asynchronous writes, small file merging, cache reconstruction, and intermediate result sharing in memory.

5.2.1 File Prefetching

We use a TPC-H test bed to evaluate the file prefetching feature with 100GB data. Two experimental control groups are established. In the first group, file prefetching is activated, while the second group disables file prefetching. To guarantee the integrity of the experimental data, each experimental condition is replicated 10 times.

Figure 9 illustrates the experiment results. Given the varying runtime of different queries, we convert the execution time with file prefetching (FP) and the execution time with no file prefetching (NFP) to percentages to facilitate an intuitive comparison. The test results indicate that enabling file prefetching reduces query run time from 20% to 51%, excluding Q13 and Q22, which see improvements of 5.6% and 2.0% respectively. The less impact on these two queries may be due to both the smaller amount of data involved in the query executions and the likelihood that these data have already been cached during earlier query runs. This suggests that file prefetching is an effective method to reduce data wait time on remote storage, which in turn enhances CPU utilization and query performance. These benefits are pronounced in cloud data warehouse scenarios where the compute node's local storage is limited compared to the volume of data they handle. When local storage is abundant, the effect is correspondingly reduced.

5.2.2 Asynchronous Writes

We use the database load process in the TPC-H benchmark to evaluate the impact of asynchronous writes on the performance of database writes. We first stage the generated data on the local disks of the compute nodes and then write it to the database as append-only tables. The database uses Gopher's multi-level storage capabilities to persist its table data in object storage. We set up two

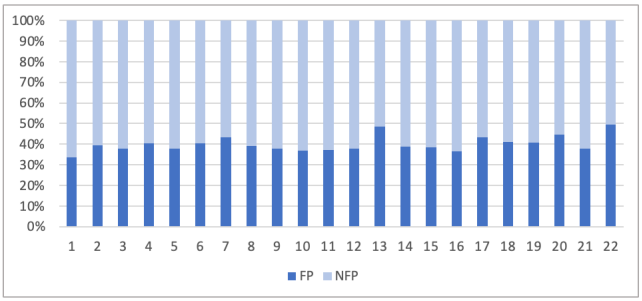

Figure 9: TPC-H 100GB with File Prefetching (FP) vs No File Prefetching (NFP).

control groups for this experiment. In the first group, the asynchronous write function is activated. After writing data to Gopher, the database process can immediately proceed to the next operation without waiting for the data to be completely written to the object storage. In the second group, the asynchronous write function is turned off. The database must wait for Gopher to complete writing data to the object storage. We repeat the test using 25GB, 50GB, 100GB, 200GB, 400GB, and 800GB data respectively.

Table 2 shows the results of the experiment. The leftmost column denotes the data volume in GB (DV). The middle two columns are the time to perform synchronous writes (SW) and asynchronous writes (AW) in seconds. The rightmost column is the improvement of asynchronous write time compared to synchronous write in percentage (IMPV).

The test results reveal that asynchronous writes enhance database write performance by 21%-45%, across varying data sizes. This improvement could be attributable to two reasons. First, the asynchronous write feature decouples the database write operation from the remote object storage write operation. This allows the database to initiate the next task promptly after completing writing to Gopher, reducing the database process's wait time. Second, when Gopher writes data to the object storage, it can aggregate multiple write requests and process them in batch. This substantially improves the efficiency of data writes to object storage over the network. The trend of test results also supports this reasoning. As the data volume increases, Gopher can consolidate more write operations to object storage, thereby amplifying the performance benefits.

5.2.3 Small File Merging

To assess the benefits of the small file merging feature, we conduct a stress test in which we write massive small files to Gopher and monitor changes in I/O utilization and read/write performance. Two control groups are set up for comparison. In the first group, the small file merging function is activated, and the files are merged in 8MB following their entry into Gopher. In the other group, this feature is turned off and all files are stored and read in their original size and quantity. Each of these two groups is configured with identical server resources and runs on 200 concurrent processes. The test workloads are generated from real-world production profiling, encompassing a large volume of files ranging from 0.5 KB and 16 KB.

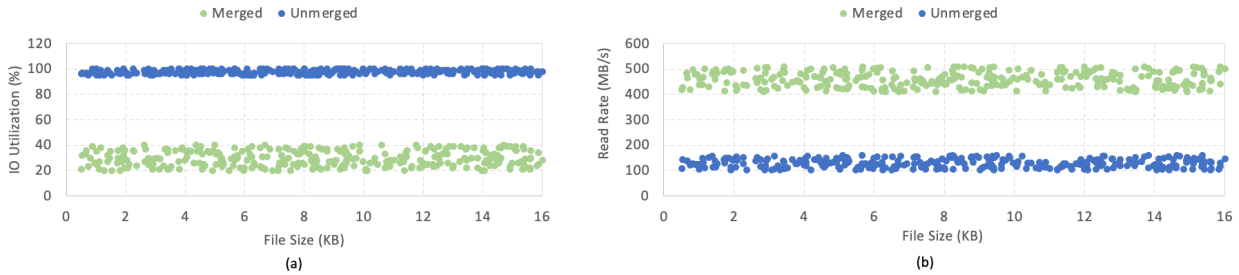

Figure 10: IO Utilization and Read Rates of Merged vs Unmerged Small Files.

DV (GB)	SW (seconds)	AW (seconds)	IMPV (%)
25	396	311	21.46%
50	790	546	30.89%
100	1580	980	37.97%
200	3149	1702	45.95%
400	6231	3394	45.53%
800	11977	6581	45.05%

Table 2: Synchronous Writes vs Asynchronous Writes.

File Num	CR (seconds)	NC (seconds)	NC / CR
320,000	12	440	36.67
640,000	21	860	40.95
1,280,000	36	1700	47.22
2,560,000	72	3369	46.79
5,120,000	130	6721	51.70
10,240,000	258	13429	52.05

Table 3: Cache Reconstruction Time.

Figure 10 shows the experiment results. In Figures X-a and X-b, the X-axis represents the file size, and the Y-axis represents the IO utilization in percentage and read efficiency in MB/second. The data of the two sets of experiments are labeled as "merged" and "unmerged", respectively.

In the test result datasets, the I/O utilization rate decreases from nearly 100% to 20%-40% after file merging. Meanwhile, the read rate jumped from 100-160 MB/second to 400-520MB/second, achieving a 2.5x to 5x performance improvement. These results show that the small file merge function can significantly mitigate disk I/O pressure and improve throughput.

5.2.4 Cache Reconstruction

To quantitively evaluate the effect of the cache reconstruction feature, we conduct an experiment to scale out a working database cluster and observe the cache reconstruction time. The database is first configured to repeatedly execute *select* queries from real production environment to fully activate the Gopher cache. Then we instruct the database to perform a cluster scale out and double its compute nodes, observing the time to reconstruct the cache. Two cache recovery strategies are applied in different experiment control groups respectively. In the first group, the cache reconstruction feature is enabled and cached data are copied from existing compute nodes to newly added nodes. In the second group, the cache reconstruction feature is disabled. The newly added nodes pull data from the storage cluster directly to build its cache. In our production environment, it is common to see a database cluster consisting of several million of files. Hence, we repeat the experiment using various numbers of files to account for different sizes. File numbers in the database cluster are 320000, 640000, 1280000, 2560000, 5120000, and 10240000.

Table 3 displays the result of the experiments. The number of files is listed in the leftmost column. The cache recovery time with and without the cache reconstruction feature (CR and NC, respectively) are shown in the central columns. The rightmost column provides the ratio of these two cache recovery time.

The experiment yielded a 36x or greater improvement in cache recovery time when the cache reconstruction feature is enabled. For instance, for an active database cluster which comprises 10 million small files, the cache reconstruction feature can substantially decrease the cache recovery time from 13429 seconds (3.7 hours) to 258 seconds (4.3 minutes). This improvement aligns with the outcomes observed in our customers' production environment. In addition, as cached data are transferred between compute nodes in 8 MB merged block rather than individual files, the benefit increases as the number of files to recover rises. The experiment results confirm this reasoning, showing that as the file number ranges from 320,000 to 10 million, the improvements of cache recovery time reach from 36x to 52x.

5.2.5 Intermediate Result Sharing in Memory

To assess the intermediate result sharing in memory feature, we use the database to run a simplified embedded query from production environment which first writes timestamps and other data to a table and reads from the writing results. Two experimental control groups are applied. In the first group, memory object is used to enable intermediate result sharing in the memory. In the other group, this feature is disabled. As both the file size and memory object size may have an impact on the execution time, we repeat the experiment using various sizes of file and memory objects. A broad spectrum of file sizes is tested to observe the cumulative effect on complex queries involving sequences of write and read operations. The file sizes are 32 MB, 128 MB, 512 MB, 2048 MB and 8192 MB. The memory object sizes are 0.5 MB, 1 MB, 2 MB, 4 MB and 8 MB.

Table 4 illustrates the result of the experiments. The file size and the memory object size are in the leftmost two columns. The execution time using memory object (M) in millisecond, the execution time without memory object (NM) in millisecond and the percentage improvement of using memory object (IMPV) are in the other columns. In all test cases, the group utilizing memory objects outperforms the other group by 25% or more.

File Size (MB)	Mem Obj Size (MB)	NM (Ms)	M (Ms)	IMPV (%)
32	0.5	28	21	25.00%
32	1	28	19	32.14%
32	2	28	18	35.71%
32	4	28	17	39.29%
32	8	28	11	60.71%
128	8	102	41	59.80%
512	8	339	120	64.60%
2048	8	1397	354	74.66%
8192	8	4813	1420	70.50%

Table 4: Share Intermediate Results with Zero-Copy Memory.

The experiment result demonstrates that memory object can significantly speed up write-read query operations by minimizing data copying, thereby reducing wait times for subsequent tasks. Additionally, performance enhancements are influenced by both the file size and the memory object sizes. This is likely because various workloads require different optimal memory object sizes for efficient data-sharing pipelines. To address this, we have provided a memory object creation interface (line 22-23 in Figure 3) that allows the client, such as a database, to define the desired memory object size, resulting in greater flexibility as well as improved overall performance and resource utilization.

6 RELATED WORK

Gopher operates as an integrated platform that provides both caching and hierarchical storage capabilities for storage-disaggregated analytical systems. This section will discuss research and industrial work in storage-disaggregated analytical systems, with a focus on cache systems and hierarchical storage, respectively.

Storage-Disaggregated Analytical System. Modern cloud databases have embraced an architecture that incorporates storage disaggregation, which include databases natively developed for the cloud (e.g. Snowflake [20], AWS Redshift [21], PolarDB [23], and TiDB [54]) as well as traditional data warehouse systems migrated to the cloud (Vertica [74], Teradata Vantage [22, 75]). In addition to cloud databases, big data processing engines like Spark [24] and Presto [25] are leveraging the benefits of storage disaggregation and are designed to support a disaggregated architecture. This computational and storage decoupling enables each component to seamlessly adjust to dynamic changes in workload demands.

Cache System. Caching systems are one of the key technologies to improve the performance and resource utilization of large-scale analytics jobs. A variety of distinctive caching technologies have been developed in both the open-source and research communities. Alluxio [8] is an open-source distributed caching system that integrates with a variety of data processing frameworks, supporting large-scale data processing tasks through its distributed nature. Yet the master-slave architecture may lead to a single point of performance bottleneck. Memcached [16] is known for its simplicity and high concurrency processing capabilities, but it has limitations in terms of data persistence. Redis [15], as an open-source memory-based key-value store, offers swift data access and comprehensive data structure support, but requires integration with additional tools or technologies in distributed database scenarios. Snowflake implements a multi-tier cache similar to Gopher. Both systems cache files in compute nodes using consistent hashing and opportunistic caching strategies. Different from the Snowflake cache [76], Gopher supports caching features such as file prefetching, asynchronous writes, small file merging, and cache reconstruction, which can significantly improve performance.

The research community has also pioneered a range of caching technologies. Hycache+ [56] allows cached data to be transparently swapped between high-speed network-attached storage and compute nodes by providing memory-level I/O throughput but may require additional optimizations to accommodate the workloads of different storage systems and database applications. Nectar [10] improves data center resource utilization and simplifies the development process by automating data and compute management, but its reuse strategy of incremental computation and shared compute operator can add complexity to the system when combined with a database. CliqueMap [13] employs Remote Memory Access (RMA) and Remote Procedure Calls (RPC) to enhance the performance and scalability of distributed caching systems. CompuCache [14] achieves a cost-effective remote computing caching solution by leveraging VMs for data offloading purposes rather than reducing latency.

Hierarchical Storage. In the cloud environment, hierarchical storage that incorporates different types of media such as DRAM, SSD, and HDD is a common approach to strike a balance among performance, capacity, and cost. [27, 28] leverage SSDs for data prefetching, enhancing the throughput of analytical processing. [57, 58, 59] utilize SSDs as read caches, maximizing the efficiency of data reads through the application of multiple caching strategies. [60, 61, 62, 63] adopt SSDs as write-back caches to implement a two-tier file buffering mechanism, optimizing I/O performance. [64, 65, 66] facilitate user management of optimal configurations for multi-layer cloud storage instances, achieving flexible and rich storage policies. [67, 68, 69, 70] detect I/O access patterns to optimize storage-driven hot and cold data migration.

7 CONCLUSION

The rise of cloud analytical databases has prompted the adoption of disaggregated storage models, which offer flexibility and cost-effectiveness but introduce challenges related to remote storage latency, bandwidth limitations, and local storage capacity constraints. Traditional caching methods, despite enhancing read performance, struggle with write operations or handling small files efficiently. This paper introduces Gopher, a novel distributed caching system designed to overcome these challenges. Gopher's decentralized architecture, enhanced by its cutting-edge features, significantly boosts I/O performance and provides rapid, concurrent access to files of all sizes. This makes it an optimal choice for a computer layer cache system, particularly suited for analytical database and AI workloads in the cloud. Experimental results have shown that Gopher can substantially improve the performance of cloud databases, achieving a fourfold increase. It achieves this by delivering query times that rival those of MPP databases while harnessing the benefits of elastic horizontal scaling.

REFERENCES

[1] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2003. Cache Tables: Paving the Way for an Adaptive Database Cache. In VLDB. 718–729.

[2] Christof Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2004. Adaptive Database Caching with DBCache. IEEE Data Engineering Bulletin 27, 2 (2004), 11–18.

[3] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. 2004. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In ICDE. 177–188.

[4] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. PVLDB 13, 12 (2020), 3411–3424.

[5] Snowflake. 2023. Caching in the Snowflake Cloud Data Platform. https://community.snowflake.com/s/article/Caching-in-the-Snowflake-Cloud-Data-Platform. accessed: 2024-01-16.

[6] Amazon. 2024. Database Caching. https://aws.amazon.com/caching/database-caching/?nc1=h_ls. accessed: 2024-01-16.

[7] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. PVLDB 16, 1 (2022), 15-22.

[8] Alluxio. 2021. Alluxio - Data Orchestration for the Cloud. https://www.alluxio.io/. accessed: 2024-01-16.

[9] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In SoCC. 6:1–6:15.

[10] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In OSDI. 75-88.

[11] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated Memory Caching for Parallel Jobs. In NSDI. 267-280.

[12] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In EuroSys. 228-244.

[13] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. 2021. CliqueMap: Productionizing an RMABased Distributed Caching System. In SIGCOMM. 93-105.

[14] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. CompuCache: Remote Computable Caching using Spot VMs. In CIDR.

[15] Redis. 2024. Introduction to Redis. https://redis.io/docs/about/. accessed: 2024-01-16.

[16] Memcached. 2020. Overview. https://github.com/memcached/memcached/wiki/Overview. accessed: 2024-01-16.

[17] Amazon. 2024. Amazon S3 Cloud Storage. https://aws.amazon.com/s3/. accessed: 2024-01-16.

[18] Ceph. 2024. Intro to Ceph. https://docs.ceph.com/en/quincy/start/intro/. accessed: 2024-01-16.

[19] Apache. 2024. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. accessed: 2024-01-16.

[20] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In SIGMOD. 215-226.

[21] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In SIGMOD. 2205-2217.

[22] Xin Tang, Robert Wehrmeister, James Shau, Abhirup Chakraborty, Daley Alex, Awny Al Omari, Feven Atnafu, Jeff Davis, Litao Deng, Deepak Jaiswal, Chittaranjan Keswani, Yafeng Lu, Chao Ren, Tom Reyes, Kashif Siddiqui, David Simmen, Devendra Vidhani, Ling Wang, Shuai Yang, and Daniel Yu. 2016. SQL-SA for Big Data Discovery Polymorphic and Parallelizable SQL User-Defined Scalar and Aggregate Infrastructure in Teradata Aster 6.20. In ICDE. 1182-1193.

[23] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. In VLDB. 2263-2272.

[24] Apache. 2024. Apache Spark. https://spark.apache.org/. accessed: 2024-01-16.

[25] Presto. 2024. Caching in Presto. https://www.qubole.com/blog/caching-presto. accessed: 2024-01-16.

[26] Junpeng Niu, Jun Xu, and Lihua Xie. 2018. Hybrid Storage Systems: A Survey of Architectures and Algorithms. IEEE Access 6 (2018), 13385-13406.

[27] K. R. Krish, Bharti Wadhwa, M. Safdar Iqbal, M. Mustafa Rafique, and Ali R. Butt. 2016. On Efficient Hierarchical Storage for Big Data Processing. In CCGrid. 403-408.

[28] Bin Dong, Teng Wang, Houjun Tang, Quincey Koziol, Kesheng Wu, and Suren Byna. 2018. ARCHIE: Data Analysis Acceleration with Array Caching in Hierarchical Storage. In 2018 IEEE International Conference on Big Data. 211-220.

[29] Jit Gupta, Krishna Kant, and Ayman Abouelwafa. 2020. FussyCache: A Caching Mechanism for Emerging Storage Hierarchies. In CloudCom. 74-81.

[30] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In SIGMOD. 331-342.

[31] Amit Shukla, Prasad M. Deshpande, and Jeffrey F. Naughton. 1998. Materialized View Selection for Multidimensional Datasets. In VLDB. 488-499.

[32] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. 1996. Answering Queries with Aggregation Using Views. In VLDB. 318-329.

[33] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang (Jim) Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Zhi (Adam) Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Yalan (Maya) Meng, Prashant Mishra, Jay Patel, Rajesh S. R., Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Ye (Justin) Tang, Junichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Gensheng Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divyakant Agrawal, Je! Naughton, Sujata Kosalge, and Hakan Hacıgümüş. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. PVLDB 14, 12 (2021), 2986-2998.

[34] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. 1990. On Rules, Procedures, Caching and Views in Data Base Systems. In SIGMOD. 281–290.

[35] Shaul Dar, Michael J. Franklin, Björn T.Jónsson , Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In VLDB. 330-341.

[36] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. 2000. Cache Investment: Integrating Query Optimization and Distributed Data Placement. TODS 25, 4 (2000), 517–558.

[37] Yannis Kotidis and Nick Roussopoulos. 1999. DynaMat: A Dynamic View Management System for Data Warehouses. In SIGMOD. 371–382.

[38] Peter Scheuermann, Junho Shim, and Radek Vingralek. 1996. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In VLDB. 51–62.

[39] Junho Shim, Peter Scheuermann, and Radek Vingralek. 1999. Dynamic Caching of Query Results for Decision Support Systems. In SSDBM. 254–263.

[40] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. PVLDB 14, 11 (2021), 2432-2444.

[41] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. 2017. Revisiting Reuse in Main Memory Database Systems. In SIGMOD. 1275–1289.

[42] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2009. An Architecture for Recycling Intermediates in a Column-store. In SIGMOD. 309–320.

[43] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. PVLDB 11, 7 (2018), 800–812.

[44] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In SIGMOD. 191–203.

[45] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. 2013. Recycling in Pipelined Query Evaluation. In ICDE. 338–349.

[46] Luis Leopoldo Perez and Christopher M. Jermaine. 2014. History-Aware Query Optimization with Materialized Intermediate Views. In ICDE. 520–531.

[47] Guodong Jin, Haoqiong Bian, Yueguo Chen, and Xiaoyong Du. 2022. Columnar Storage Optimization and Caching for Data Lakes. In EDBT. 2-419.

[48] Tianru Zhang, Andreas Hellander, and Salman Toor. 2022. Efficient Hierarchical Storage Management Empowered by Reinforcement Learning. IEEE Transactions on Knowledge and Data Engineering 35, 6 (2023), 5780–5793.

[49] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. PVLDB 16, 11 (2023), 2769-2782.

[50] Shakil B. Tamboli and Smita Shukla Patel. 2014. A Survey on an Efficient Data Caching Mechanism for Big Data Application. IJSR 11, 3 (2014), 1242-1247.

[51] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. IEEE Transactions on Knowledge and Data Engineering 27, 7 (2015), 1920-1948.
[52] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafni, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. PVLDB 14, 11 (2021), 2101-2113.
[53] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. PVLDB 13, 12 (2020), 3072-3084.
[54] Rong Chen, Haibo Chen, and Binyu Zang. 2010. Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling. In PACT. 523-534.
[55] Dongfang Zhao, Kan Qiao, and Ioan Raicu. 2014. HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems. In CCGRID. 267-276.
[56] Yi Liu, Xiongzi Ge, Xiaoxia Huang, and David H.C. Du. 2013. MOLAR: A Cost-Efficient, High-Performance Hybrid Storage Cache. In CLUSTER. 1-5.
[57] Jianzhe Tai, Bo Sheng, Yi Yao, and Ningfang Mi. 2015. SLA-aware data migration in a shared hybrid storage cluster. Cluster Computing 18 (2015), 1581-1593.
[58] Ningwei Dai, Yunpeng Chai, Yushi Liang, and Chunling Wang. 2015. ETD-Cache: An Expiration-Time Driven Cache Scheme to Make SSD-Based Read Cache Endurable and Cost-Efficient. In CF. 1-8.
[59] Lin Lin, Yifeng Zhu, Jianhui Yue, Zhao Cai, and Bruce Segee. 2011. Hot Random Off-loading: A Hybrid Storage System With Dynamic Data Migration. In MASCOTS. 318-325.
[60] Xian Chen, Wenzhi Chen, Zhongyong Lu, Peng Long, Shuiqiao Yang, and Zonghui Wang. 2015. A Duplication-Aware SSD-Based Cache Architecture for Primary Storage in Virtualization Environment. IEEE Systems journal 11, 4 (2015), 2578-2589.
[61] Taeho Kgil and Trevor Mudge. 2006. FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers. In CASES. 103-112.
[62] Sai Huan, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. 2016. Improving Flash-based Disk Cache with Lazy Adaptive Replacement. ACM Transactions on Storage 12, 2 (2016), 1-24.
[63] Feng Chen, David Koufaty, and Xiaodong Zhang. 2011. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In ICS. 22-32.
[64] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. 2011. HybridStore: A Cost Efficient, High Performance Storage System Combining SSDs and HDDs. In MASCOTS. 227-236.
[65] Gong Zhang, Lawrence Chiu, and Ling Liu. 2010. Adaptive Data Migration in Multi-tiered Storage Based Cloud Environment. In CloudCom. 148-155.
[66] Ajaykrishna Raghavan, Abhishek Chandra, and Jon Weissman. 2014. Tiera: Towards Flexible Multi-Tiered Cloud Storage Instances. In MIDDLEWARE. 1-12.
[67] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. 2011. Cost Effective Storage using Extent Based Dynamic Tiering. In FAST. 273-286.
[68] Hui Wang and Peter Varman. 2014. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In FAST. 229-242.
[69] Azure. 2024. Azure Blob Storage documentation. https://learn.microsoft.com/en-us/azure/storage/blobs. accessed: 2024-01-16.
[70] Apache. 2024. Apache Hive. https://hive.apache.org/. accessed: 2024-01-16.
[71] Apache. 2024. Apache Hudi. https://hudi.apache.org/. accessed: 2024-01-16.
[72] Apache. 2024. Apache Iceberg. https://iceberg.apache.org/. accessed: 2024-01-16.
[73] CNCF. 2024. etcd. https://etcd.io/. accessed: 2024-01-16.
[74] Vertica. 2024. Vertica documentation. https://www.vertica.com/docs/10.0.x/HTML/Content/Home.htm. accessed: 2024-01-16.
[75] Teradata. 2024. Teradata documentation. https://docs.teradata.com. accessed: 2024-01-16.
[76] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In NSDI. 449–462.