CacheServer: Disaggregated Caching for Cloud Databases

JUNYONG ZHAO, University of Arizona, USA
JIA YUAN, University of Arizona, USA
LEI CAO, University of Arizona, USA

Cloud databases typically cache data fetched from cloud storage to compute nodes and conduct *affinity scheduling* to guarantee data locality – scheduling queries accessing the same data segment to the same node. Although this strategy improves the cache hit rate and effectively reduces the prohibitive data transmission cost due to the low bandwidth of cloud storage, coupling cache with query execution causes major issues. First, it excludes other scheduling mechanisms from exploiting the merit of data caching. Second, it causes load imbalance under skewed workloads. To address these issues, we propose CacheServer, a system that decouples data caching from query execution, introducing dedicated cache nodes to manage data caching. Under this architecture, query execution nodes only fetch data from cache servers. Therefore, it always benefits from the high bandwidth of intra-data center network no matter how a cloud database schedules queries to query execution nodes, thus enjoying the advantage of data locality while not suffering from stragglers caused by hot data. To make this architecture effective, we propose a data prefetching mechanism that maximizes the availability of the data when a query requests it from cache servers, by overlapping fetching data from cloud storage with the execution of previous queries. Furthermore, CacheServer designs an automatic *resource allocation* algorithm, which given a resource budget, automatically allocates resources between cache and query execution nodes to minimize query latency. Our experimental evaluation on various workloads under different scheduling mechanisms shows that CacheServer outperforms alternatives from 2 to 10× in average execution time and tail latency, given the same resource budget.

CCS Concepts: • **Information systems → Parallel and distributed DBMSs**.

Additional Key Words and Phrases: Cloud Databases, Cache, Query Scheduling

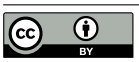
ACM Reference Format:
Junyong Zhao, Jia Yuan, and Lei Cao. 2026. CacheServer: Disaggregated Caching for Cloud Databases. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 264 (September 2026), 27 pages. https://doi.org/10.1145/3837102

1 Introduction

Modern cloud databases widely adopt a *storage disaggregation architecture*, which separates compute nodes from data storage nodes, offering the flexibility to scale the computation and data storage separately. However, when queries are assigned to compute nodes, the nodes have to fetch data segments from the remote cloud storage to compute nodes through an inter-data center network with low bandwidth. This typically constitutes the performance bottleneck.

A common solution to address this bottleneck is *local caching*, where a compute node keeps the data segments fetched from cloud storage for subsequent queries to reuse. To improve the cache hit rate, cloud databases typically use *affinity scheduling* [21, 30, 51] to schedule queries accessing the same data segments to the same nodes. A typical way to realize affinity scheduling in cloud databases [21, 33, 51, 58] is *consistent hashing* [35, 36].

Authors' Contact Information: Junyong Zhao, University of Arizona, USA, junyong@arizona.edu; Jia Yuan, University of Arizona, USA, jiayuan@arizona.edu; Lei Cao, University of Arizona, USA, lcao@csail.mit.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/9-ART264
https://doi.org/10.1145/3837102

However, local caching fundamentally breaks the principle of separating computation and storage in a storage disaggregated architecture, which brings major issues into the cloud database system, including *load imbalance* and *inflexible query scheduling*.

First, in real-world applications, query workloads are typically skewed where a portion of data segments is disproportionately popular, often due to temporal or spatial factors. For example, viral social media posts can quickly attract massive numbers of views. Such skewed workloads are challenging with affinity scheduling as many queries accessing the same hot data segment are directed to a single node for data locality, causing it to become overloaded and straggle. This *load imbalance* on compute nodes results in significantly increased *tail latency* and degraded system performance.

Moreover, local caching brings *minimal benefits to other scheduling strategies*. Beyond affinity scheduling, other strategies are also used in cloud databases for different purposes. For example, to achieve balanced load on compute nodes, *random scheduling* [44] is used to spread queries evenly to all compute nodes. In a multi-tenant environment where multiple users share the same set of compute nodes, *fair scheduling* [25, 26] and *capacity scheduling* [62] ensure each user gets a fair share or at least a certain amount of node capacities. As these strategies do not enforce data locality, they benefit little from local caching due to the low cache hit rate.

Proposed Architecture. We propose CacheServer, a new cloud DB architecture that *disaggregates data caching* from *query execution*. Rather than caching data segments and executing queries on the same set of compute nodes, CacheServer divides the compute nodes into *cache nodes* and *execution nodes*. Cache nodes act as dedicated caching servers that retrieve data segments from remote cloud storage, store them, and serve data requests from execution nodes. Because both cache and execution nodes are provisioned within the same data center, execution nodes can leverage the high-speed intra-data center network to access data segments when they are available in the cache. This design mitigates the high data transmission costs caused by the limited bandwidth of inter-data center networks. As a result, the cloud database can flexibly employ any scheduling strategies while still benefiting from data caching, without incurring load imbalance under skewed workloads.

To the best of our knowledge, this is the first work in cloud databases that decouples data caching from query execution to address the issues caused by local caching. While some works such as Alluxio [41, 54] provide a unified abstraction to access different cloud storage platforms, they focus on user-friendly APIs and managing intermediate results through lineage. In serverless platforms, Pocket [37] offers a storage system to hold ephemeral intermediate results. However, it focuses on automatically rightsizing resources such as CPU cores and storage capacity to minimize cost, orthogonal to CacheServer's focus on benefiting from data caching while not introducing load imbalance. In areas such as network and web servers, improvements of consistent hashing [29, 43] have been proposed to mitigate load imbalance under skewed query workloads. Such designs steer queries accessing hot data to other nodes if the load on a node exceeds a pre-determined threshold. However, unlike CacheServer that eliminates the root cause of load imbalance while still benefiting from data locality, these designs suffer from high data transmission costs due to replicating data all over compute nodes, as confirmed in our experiments (Sec. 6.2). Moreover, they are still based on affinity scheduling, thus not solving the issue of inflexible scheduling strategies.

Challenges. In order to make CacheServer effective, CacheServer has to address the following challenges:

- **Maximize Cache Hit Rate.** As with local caching, the performance of CacheServer depends heavily on achieving a high cache hit rate, meaning that requested data segments are already present on cache nodes when accessed by execution nodes. If a cache miss occurs, the system

must retrieve the data from cloud storage before responding, which introduces substantial data transmission overhead and ultimately degrades overall query performance.

• **Trade-off Query Execution and Data Transmission.** CacheServer requires additional resources for cache nodes. Under a computing resource budget, although allocating more resources to cache nodes will reduce data transmission cost due to larger cache capacity and network bandwidth, this weakens the computation power of execution nodes, inevitably slowing down query execution and introducing higher query execution latency. Thus, it is critical to carefully allocate resources between cache nodes and execution nodes, i.e., trade-off query execution and data transmission to minimize the overall query latency.

Proposed Solution. CacheServer addresses these challenges with the following key techniques: (1) *predictive prefetch* that maximizes cache hit rate and (2) *latency-aware* algorithm that determines the optimal execution-cache node allocation to minimize query latency.

• **Predictive Prefetch.** The decoupling of data caching and query execution is suitable for CacheServer to maximize cache hit rate via prefetch because of its *global view* to all segments on cache nodes. CacheServer constantly monitors upcoming queries in the coordinator and predicts the data segments that will be used in the near future; and it only prefetches a segment if it is *guaranteed* to improve the cache hit rate. To achieve this, CacheServer designs a *score function* to evaluate the impact of cached data segments on cache hit rate. This score function makes it possible to maximize the gain of prefetching a segment, i.e., discovering a subset of cached segments to evict that has the minimal total score and is large enough to accommodate the new prefetched segment. We formally show that this is an NP-hard problem and solve it with an efficient algorithm with a 2-approximation guarantee.

• **Resource Allocation.** We design a *latency-aware* method to allocate resources under a given resource budget. The key idea is to model the *expected latency* of a query workload as a function that takes the number of execution and cache nodes as variables. In particular, the latency function considers (1) the average execution time determined by the number of execution nodes; (2) the time transmitting data from cache nodes to execution nodes determined by the number of cache nodes; and (3) the time transmitting data from cloud storage to cache nodes on cache miss, determined by the ratio between execution nodes and cache nodes. Based on this latency function, we formulate the resource allocation problem as a single-variable optimization problem and mathematically derive the optimal solution by calculating the critical point of the function.

Combining the predictive prefetch and latency-aware resource allocation, CacheServer demonstrates the advantage of cache disaggregation and bring benefits of caching to any scheduling strategies.

Contributions. This paper makes the following key contributions:

• We propose a *cache-disaggregated architecture* that for the first time addresses the problems caused by local caching and affinity scheduling in cloud databases.

• We propose a data *predictive prefetch* mechanism that fully takes advantage of decoupling query execution and data caching, maximizing the cache hit rate of a query workload.

• We introduce a *latency-aware resource allocation* method that determines the optimal number of execution nodes and cache nodes to minimize the overall query latency under a fixed resource budget.

• Our experiments with various workloads and scheduling strategies confirm that CacheServer outperforms the local caching-based alternatives from 2× to 10× in average execution time and tail latency under the same resource budget.

2 Background

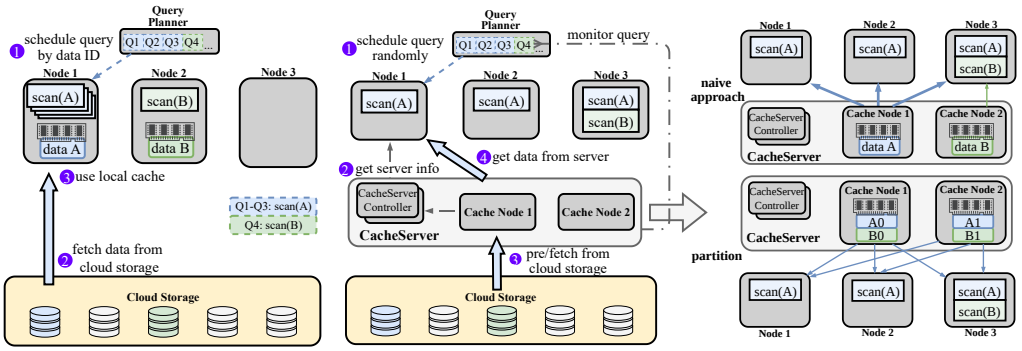
This section overviews local caching in storage-disaggregated cloud databases and its coupling to affinity scheduling (Sec. 2.1), as well as the limitations (Sec. 2.2).

2.1 Local Caching and Affinity Scheduling in Cloud Databases

Storage Disaggregation Architecture. Databases designed for cloud platforms, such as Presto [52], F1 Query [51], Snowflake [21, 58] and SparkSQL [10], commonly adopt a storage-disaggregation architecture. This design decouples computation and storage, bringing major advantages such as better scalability and higher elasticity.

Local Caching. For a disaggregated storage architecture, when a node receives a query, it has to fetch these data segments from the distributed storage through a network. Due to the overhead of the distributed file system and the low bandwidth of the inter-data center network, *accessing* and *transmitting* data from cloud storage to compute nodes constitutes the performance bottleneck. This often leads to performance degradation compared with a shared-nothing database [53]. Caching addresses this problem where compute nodes cache a data segment locally and reuse it on other queries rather than repeatedly fetching it from cloud storage.

Affinity Scheduling. To improve cache hit rate, cloud databases use *affinity scheduling* [15, 21, 24, 30, 51] to distribute queries to compute nodes, ensuring that subsequent or concurrent queries accessing the same data segment will be assigned to the same nodes. Many of these systems implement *consistent hashing* [35, 36], one of the most popular form of affinity scheduling. This *data locality* guarantee greatly improves the system's performance.



(a) Local Caching (b) CacheServer (c) CacheServer with Partitioning

Fig. 1. CacheServer Architectures and Workflow

2.2 Limitations of Local Cache

Although improving the performance of cloud databases, caching data locally on compute nodes fundamentally violates the principle of storage disaggregation, as data cache essentially corresponds to a particular type of data storage. This brings severe limitations.

Load Imbalance Under Skewed Workloads. In real applications, query workloads are typically very skewed, where some data segments (hot data) are accessed much more frequently than others. Since local cache relies on affinity scheduling to ensure high cache hit rate, the nodes that host hot data segments will receive many more queries than those caching cold data segments. This leads to load imbalance and large query latencies on hot nodes.

Minimal Benefit to Other Scheduling Strategies. In addition to affinity scheduling, there exist other popular scheduling strategies, such as *random scheduling* [44], fair scheduling [25, 26] and

capacity scheduling [62]. *Random scheduling* evenly distributes queries to compute nodes, naturally ensuring load balance across compute nodes. For example, ScyllaDB [5] uses random scheduling to distribute queries to compute nodes. In a multi-tenant environment where different users share resources, *fair scheduling* and *capacity scheduling* are commonly adopted to ensure that each user gets its fair share of compute nodes or receives at least a certain amount of resources measured by node capacities. These strategies, which schedule queries either randomly or based on the resources consumed by users, do not enforce data locality, thus gaining little from local cache due to low cache hit rate.

3 CacheServer Overview

CacheServer addresses the limitations of local caching with a novel architecture that disaggregates data cache with query execution.

3.1 CacheServer Architecture

Fig. 1b illustrates the high-level architecture of CacheServer. Instead of caching data segments on nodes that execute queries, CacheServer divides compute nodes in a cloud database into *cache nodes and execution nodes*, where the cache nodes fetch data segments from remote cloud storage, cache them, and respond to data requests from execution nodes.

We assume that the cloud DB has already *partitioned* tables into data segments using any partitioning method such as range partitioning or hash partitioning. It stores each data segment in cloud storage, such as Amazon S3, as a file in open file format. A data segment is the basic unit to be processed. When a query is submitted to the coordinator, its *query planner* first determines the data segments the query accesses. Metadata such as the minimum and maximum values within each data segment could be used to filter out segments that do not satisfy the query predicates. The query is thus divided into multiple sub-queries, each processing a specific data segment.

The CacheServer Workflow. Next, we show how CacheServer works in a cloud DB with a sub-query Q_1 as an example.

Suppose Q_1 accesses data segment A. Then, it distributes Q_1 to an execution node $Node_1$ using a scheduling mechanism, e.g., random scheduling. As shown in Fig. 1b Steps 1 & 2, $Node_1$ keeps heartbeat communications with the CacheServer controller and requests segment A from cache nodes. If it is not available, CacheServer fetches A from cloud storage, maintains it on cache nodes, and sends it to $Node_1$, as shown in Fig. 1b Steps 3 & 4. Otherwise, CacheServer sends A to $Node_1$ (Step 4). Finally, $Node_1$ executes Q_1 on the received data segment A.

Because both cache and execution nodes reside in the same data center, as long as a data segment is available on cache nodes, any execution node can take advantage of the fast intra-data center network to fetch it, avoiding the high data transmission cost due to the low bandwidth of inter-data center network. In this way, cloud DBs achieve *data locality*, i.e., high cache hit rate, while *not coupled* with affinity scheduling. As shown in Fig. 1b, even if a cloud DB randomly schedules queries (Q_1, Q_2, and Q_3) accessing data segment A to three different execution nodes, all execution nodes equally benefit from the low transmission cost of fetching A from cache nodes. Moreover, unlike local cache, CacheServer does not suffer from load imbalance caused by affinity scheduling, which as shown in Fig. 1b, has to schedule all three queries to one execution node $Node_1$ to hit the cache.

Caching Raw Data. Similar to other works in cloud databases [21, 58, 61], CacheServer focuses on caching raw input tables and optimizing the performance of scan operators, as scan operators involve fetching data from cloud storage.

Architectural Advantage. Coupling data caching with query execution introduces complexity in resource configuration and fault tolerance. In particular, because the query engine potentially consumes a huge amount of memory when processing complex queries, co-locating caching with

query execution has to conservatively reserve memory for data caching such that the query engine always has sufficient amount of memory to use. Therefore, it tends to have less memory to cache date compared to CacheServer.

Local cache also makes the system hard to scale. That is, it cannot separately scale the capacity for data cache and for query execution. The disaggregation of data caching and query execution makes CacheServer more elastic, i.e, being able to up-scale or downscale the cache nodes and execution nodes independently.

3.2 CacheServer Key Designs

To effectively realize the novel CacheServer architecture has to solve the following problems:

• **(1) Avoid Network Congestion.** As discussed in Sec. 2, query workloads are often very skewed. Although CacheServer effectively avoids load imbalance on execution nodes, a cache node that keeps hot data segments tends to incur a large number of data requests. For example, as shown in Fig. 1c, assume node $Cache_1$ keeps data segment A and $Cache_2$ keeps B. The three execution nodes that process queries accessing A will fetch data from $Cache_1$, while only one execution node that processes a query accessing B will fetch data from $Cache_2$. Therefore, $Cache_1$ incurs 3 times larger network traffic than $Cache_2$, potentially leading to network congestion.

• **(2) Maximize Cache Hit Rate.** Similar to local cache, the performance of CacheServer relies on a high cache hit rate, i.e., the probability that a data segment is already available on cache nodes when it is requested by execution nodes. Otherwise, it has to fetch data from cloud storage on the fly. The decoupling of cache nodes and execution nodes brings new opportunities to optimize cache hit rate. That is, it makes it possible for CacheServer to *prefetch* the data segments that queries will access in the near future, agnostic to query execution. That is, CacheServer only has to be aware of the next batch of queries to schedule, without worrying about how these queries will be scheduled and executed on execution nodes. However, given the limited memory on cache nodes, it naturally raises the problem of how many and what *new data segments to prefetch* or *old data segments to evict* from cache.

• **(3) Trade-off Query Execution and Data Transmission.** The benefit of CacheServer is not a free lunch due to the additional resource consumption of cache nodes compared with local cache. Given a budget, how to allocate resources between cache nodes and execution nodes will greatly impact the overall performance of the CacheServer. Allocating more resources to cache nodes will result in larger cache capacity and network bandwidth, thus reducing data transmission cost. However, this inevitably slows down the query engine, leading to large query execution latency. Alternatively, provisioning more compute nodes potentially improves the speed of query execution with the price of increased data transmission cost. This trade-off between query execution cost and data transmission cost introduces a critical problem of how to allocate resources to minimize the overall query latency.

Next, we present our *partition*-based approach to solve the problem of skewed data requests in this section. We then discuss in Sec. 4 our *predictive prefetch* that maximizes cache hit rate. In Sec. 5, we present our *query latency-aware* approach that determines the optimal resource allocation under a budget.

Partition-based Balancing. An intuitive idea to mitigate network congestion caused by skewed data requests is to replicate hot data segments to multiple nodes [9, 29, 46] and distribute requests to different replications. However, this incurs high memory overheads, leading to reduced cache capacity and hence cache hit rate.

To solve this problem, CacheServer designs a *partition*-based approach to balance network traffic across cache nodes without introducing extra memory overhead. Specifically, CacheServer partition each data segment into multiple chunks and distributes these chunks across different cache nodes

through, for example, consistent hashing. For example, as shown in Fig. 1c, CacheServer could split data segments A and B into A_0, A_1 and B_0, B_1, respectively, and store A_0 and B_0 on $Cache_1$ but A_1 and B_1 on $Cache_2$. When an execution node requests A, it retrieves A_0 and A_1 from both $Cache_1$ and $Cache_2$. This evenly splits the traffic across cache nodes, even if some data segments are much hotter than others.

4 Predictive Prefetch

We first present the objective and overview the process of predictive prefetch in Sec. 4.1. We then discuss in detail the cache eviction strategy in Sec. 4.2 that coordinates with prefetch.

4.1 The Overview of CacheServer Prefetch

Intuition. Prefetching a data segment to cache nodes before a query requests it will greatly improve cache hit rate, thus reducing query latency. As shown in Fig. 2, a query will access data segment D in the future at $t = 3$. However, D is not in the cache. In this case, it will take CacheServer $100ms$ to fetch D from cloud storage to cache nodes, and another $30ms$ to transmit it to execution nodes. The total data transmission cost will be $130ms$. However, if there is a way to prefetch it, the cost will be down to $30ms$.

The decoupling of data cache and query execution offers more opportunities for CacheServer to design an effective prefetch strategy. As discussed in Sec. 3.1, the CacheServer controller constantly monitors the upcoming queries that are queued in the coordinator of a cloud DB and will be scheduled later. Therefore, CacheServer can predict what data segments will be accessed in the near future and fetch them beforehand.

Moreover, execution nodes share the data segments on all cache nodes. This creates a large optimization room for us to design smart prefetch strategy that globally benefits all queries on different execution nodes. For local cache-based architecture, prefetching data consumes extra CPU, memory, and network capacity of execution nodes, slowing down query execution and potentially degrading the robustness of query engine.

CacheServer Prefetch Objectives. The objective is to maximize the cache hit rate, ideally ensuring that a data segment is always available on cache nodes when a compute node requests it. Under limited cache resource, to achieve this, we have to solve two problems: (1) what data segments to prefetch and (2) what existing data segments to evict. As shown in Fig. 2, at time $t = 1$, CacheServer finds that data segment D will be used at time $t = 3$ and is currently not available. Potentially, it could prefetch D. However, since the cache is full now, CacheServer has to evict some data segments. B will be accessed earlier than D, while A is accessed now and will be accessed again immediately after D at time $t = 4$. Therefore, to reduce the chance of prefetch causing future cache misses, we should evict C and E to make space for D, because they will be requested later than A and B. Thus, we have more chance to bring them back on time when they are queried. However, prefetching D does not always win, if, for example, D is too big to accommodate without kicking out B additionally.

To solve the above problems, we design a *score function* to evaluate the *value* of a to-be-fetched segment and a set of to-be-evicted candidate segments. Then we design an algorithm to discovers a set of evict candidates that have the smallest possible value and decide whether to prefetch a segment.

The Prefetch Process. As shown in Alg. 1, we denote the current cached segments as C and the maximum cache capacity as S. CacheServer continuously monitors data segments L that will be accessed by the to-be-scheduled queries in the coordinator (Line 2). Next, CacheServer iterates through each data segment d_i in L but not cached in C (Line 3). For each d_i, if there is space to hold it, CacheServer will simply fetch it from cloud storage (Lines 4-7). Otherwise, it identifies an

Algorithm 1: CacheServer Prefetch

```
Input: Cache set C, Maximum cache size S, Query history H
1  while true do
2      L = monitor(coordinator)
3      if L ≠ ∅ then
4          for d_i ∉ C, d_i ∈ L do
5              if sizeof(d_i) ≤ S − sizeof(C) then
6                  fetch(d_i)
7                  C = C ∪ d_i
8                  continue
9              if evaluate(d_i, E = findEvict(C)) then
10                 fetch(d_i)
11                 C = (C ∪ d_i) \ E
12             else
13                 break
14     else
15         L = predict(H)
16         goto 4
```

eviction candidate set E and evaluates it against d_i to decide whether to fetch d_i or not. That is, if the score of d_i is higher than the total score of the segments in E, CacheServer evicts E and fetches d_i; otherwise, it continues to process the next prefetch candidate in L.

Next, we discuss the function that scores a segment and the algorithm that identifies the set of eviction candidates.

4.2 Scoring and Identifying Eviction Candidates

4.2.1 *Score Function*

CacheServer scores a data segment based on two factors: (1) the *distance for the next access* and (2) the *size*. The first factor ($distance(d_j)$) reflects *how soon* a data segment d_j will be accessed in the future, while the second factor ($sizeof(d_j)$) represents how long it will take CacheServer to fetch it from cloud storage. Next, we define the score function $s(d_i)$ in Eq. 1.

$$s(d_i) = sizeof(d_i)/distance(d_i) \tag{1}$$

Intuitively, this score reflects the value of caching a segment d_i: the larger it is and the sooner it will be accessed, the higher its value. This value also represents the loss of evicting a segment from cache.

By Eq. 1, CacheServer prefers to evict cache small segments that are distant to be used in the future over large segments that will be used soon. We measure the size of a segment d_i as its relative size to the smallest segment in the database. We measure the distance of d_i as the number of queries in front of the query that will access it. For example, as shown in Fig. 2 Step 1, $s(A)$= 1/1 = 1, while the scores of $s(B)$, $s(C)$ and $s(E)$ are 0.5, 0, and 0.2, respectively.

4.2.2 *Identifying Eviction Candidates*

Next, we first formally define the problem of *selecting an optimal eviction candidate set* in Def. 4.1. We then show that it is an NP-hard problem and offer a solution with a 2-approximation guarantee.

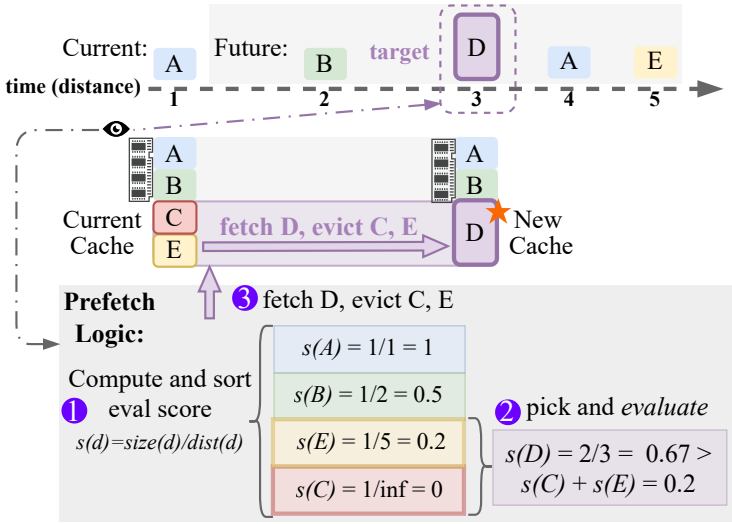


Fig. 2. An Example of CacheServer Prefetch

Problem Definition. Intuitively, we want to find a set of segments from the current cache C that has the minimal overall score but is large enough to accommodate a prefetch candidate d_i.

Definition 4.1. The goal is to find an eviction candidate set $E \in C$ such that:

$$\begin{aligned} &\text{minimize} \sum_{d_j \in E} sizeof(d_j)/distance(d_j) \\ &\text{subject to} \sum_{d_j \in E} sizeof(d_j) \geq sizeof(d_i) \end{aligned} \tag{2}$$

Here, using $v_j = sizeof(d_j)/distance(d_j)$, $w_j = sizeof(d_j)$, and $W = sizeof(d_i)$, we could rewrite Eq. 2 as:

$$\begin{aligned} &\text{minimize} \sum_{d_j \in E} v_j x_j \\ &\text{subject to} \sum_{d_j \in E} w_j x_j \geq W \text{ and } x_j \in \{0, 1\} \end{aligned} \tag{3}$$

Then solving Eq. 3 is equivalent to finding the complement set $E' = C \setminus E$ where:

$$\begin{aligned} &\text{maximize} \sum_{d_j \in E'} v_j x_j \\ &\text{subject to} \sum_{d_j \in E'} w_j x_j \leq S - W \text{ and } x_j \in \{0, 1\} \end{aligned} \tag{4}$$

The Selection Algorithm. Eq. 4 matches exactly the classic NP-hard 0-1 Knapsack Problem [50]. The Knapsack Problem is well studied in literature. CacheServer adopts a greedy algorithm [20] with a 2-approximation guarantee [20]. Specifically, CacheServer sorts the eviction candidates by their scores in non-decreasing order, such as $A \rightarrow B \rightarrow E \rightarrow C$, as shown in Fig. 2 Step 1. Next, CacheServer select the segments with the lowest scores until the total size of these segments is sufficient to accommodate the prefetch target D, as shown in Fig. 2 Step 2. Because the score of D is larger than the total score of C and E, CacheServer evicts C and E to fetch D (Fig. 2, Step 3).

4.3 Further Discussion

Prefetching Multiple Data Segments. Although CacheServer prefetches at individual data segment level, it will make all data segments required by a query available on cache nodes which are then processed by the execution nodes in parallel. That is, prefetching multiple data segments is handled as multiple requests that prefetch single-segment. This is possible because CacheServer divides a user query into multiple sub-queries, each processing a specific data segment, as discussed in Sec. 3.1.

Cost of Scan Planning. The effectiveness of CacheServer prefetch is not directly associated with the scan planner, which can be computationally expensive. Depending on the cloud database's architecture, the planner interacts with the queue in one of two ways: (1) the scanner identifies required segments iteratively and pushes them into the execution queue one by one; (2) the scanner identifies the complete set of required segments first, and then passes them to the queue together. CacheServer prefetch is effective in both scenarios, as long as there are queries and their paired segments waiting in the queue such that CacheServer has sufficient time to make the data segments requested by a query available on cache node before the execution node completes the other queries in front of it. In heavily loaded cloud systems with burst workloads, it is typical for queries to experience queuing, as confirmed by our experiments with the Redset [55] cloud DB workload.

Augmenting Prefetch. Our prefetch can be augmented with other prefetch strategies to handle different scenarios where queries might not be accumulated in the coordinator. CacheServer supports this by adopting a hybrid prefetch process. For example, as shown in Alg. 1 Line 3, when there are queries accumulated in the queue, it will call our prefetch method as discussed. Otherwise, it uses historical query access patterns to predict future data access and prefetch accordingly [32] (Alg. 1 Lines 14-16).

5 Resource Allocation

Next, we present our resource allocation method that minimizes query latency subject to a resource budget.

Problem Definition. CacheServer has to reserve a subset of nodes to cache data. Therefore, subject to a resource budget, CacheServer has fewer nodes to use in query execution. For example, given a budget of 20 nodes, CacheServer might use 10 nodes to cache data and 10 nodes to execute queries. However, it still tends to outperform a cloud database that uses all 20 nodes to run queries in end-to-end query latency. This is because the dedicated cache nodes and the prefetching strategy minimize data transmission cost, while not introducing load imbalance on execution nodes. This gain outweighs the loss of query execution capacity. However, if allocating too many resources to cache nodes, e.g., 15 nodes, then queries might pile up on execution nodes due to the weak computing power of query execution engine, thus degrading the performance of a cloud database instead.

Therefore, how to allocate resources appropriately between cache nodes and execution nodes is critical to minimize query latency. Next, we formally define this resource allocation problem.

Definition 5.1. Given a budget of k nodes, the resource allocation problem is to find an m and an n such that :

$$\begin{aligned} &\text{minimize } l(\mathbb{Q}) \\ &\text{subject to: } m + n = k \end{aligned} \tag{5}$$

where $l(\mathbb{Q})$ represents the latency of a query workload $\mathbb{Q}$, m represents the number of cache nodes, and n represents the number of execution nodes.

The High Level Idea. The idea is to approximate $l(\mathbb{Q})$ as a function that takes m and n as input variables. This is possible because as discussed, the latency of a query workload is governed by (1) the data transmission cost and (2) the query execution capacity, which are controlled by the number of cache nodes m and the number of execution nodes n, respectively. Then we compute the m and n values that minimize this function.

The Expected Latency Per Sub-Query. As discussed in Sec. 3.1, each query is processed in parallel as sub-queries on different execution nodes. We start with approximating the expected latency per sub-query q on an execution node, i.e., $l(q)$. $l(q)$ relies on two factors: (1) query execution time and (2) data transmission time. Although CacheServer's data prefetching effectively overlaps the time fetching a data segment accessed by a sub-query with the execution of other sub-queries, the end-to-end latency of one sub-query amounts to the time of fetching and processing a single segment.

Suppose a sub-query q accesses a data segment d. Let D denote the total size of data segments accessed by the query workload $\mathbb{Q}$, and S denote the cache capacity of each cache node:

Definition 5.2. The *latency* of a sub-query $l(q)$ is estimated as:

$$l(q) = exec(q) + \frac{d}{B_c \frac{m}{n}} + \left(1 - \frac{mS}{D}\right) \frac{d}{B_s} \tag{6}$$

Here, the first term $exec(q)$ represents the execution time of q. The second term represents the time of transmitting a data segment d from cache nodes to execution nodes, determined by the bandwidth between them as well as the size of d. Note that the averaged bandwidth w.r.t. each execution node is $m/n \times B_c$ because n execution nodes share the bandwidth of m cache nodes. The last term represents the time of transmitting d from cloud storage to cache nodes when there is a cache miss. We denote its corresponding bandwidth as B_s, while $1 - mS/D$ is the probability of cache miss based on the geometric distribution.

The Latency of A Query Workload. Based on the expected per sub-query latency on an execution node, we further estimate the latency of a query workload $\mathbb{Q}$, i.e., $l(\mathbb{Q})$. Suppose $\mathbb{Q}$ has W queries. The latency of $\mathbb{Q}$ is determined by the last sub-query to finish. Assume that all sub-queries are randomly scheduled to n execution nodes. Given the number of execution nodes n, the *expected time* for the last sub-query to finish $l(\mathbb{Q}) = l(q) \times W/n$:

Definition 5.3. The latency $l(\mathbb{Q})$ of a query workload $\mathbb{Q}$ is:

$$l(\mathbb{Q}) = \frac{W}{n} \times l(q) \Rightarrow l = \frac{Wexec(q)}{n} + \frac{Wd}{B_c m} + \left(1 - \frac{mS}{D}\right) \frac{Wd}{B_s n} \tag{7}$$

Here, the first term $Wexec(p)/n$ represents the total execution time of all sub-queries in $\mathbb{Q}$. It decreases as the number of execution nodes n increases. The second term $Wd/B_c m$ represents the total time of transmitting data from cache nodes to compute nodes, which decreases as the number of cache nodes m increases. The third term represents the total time of transmitting data from cloud storage to cache nodes due to cache misses. It decreases as the number of cache nodes m increases, resulting in a smaller number of execution nodes n. Now $l(\mathbb{Q})$ is expressed as a function with m and n as variables. Clearly, this reflects the trade-off between n and m in terms of the impact on the latency of $\mathbb{Q}$.

Reformulate the Resource Allocation Problem. Next, based on Eq. 7, we reformulate the resource allocation problem as follows:

Definition 5.4. Given a budget of k nodes:

$$\begin{aligned} &\text{minimize } l(\mathbb{Q}) = \frac{Wexec(q)}{n} + \frac{Wd}{B_c m} + \left(1 - \frac{mS}{D}\right)\frac{Wd}{B_s n} \\ &\text{subject to: } m + n = k \end{aligned} \tag{8}$$

Calculate Optimal m and n. Replacing n with $k - n$ reduces the problem in Def. 5.4 to a single-variable optimization problem. Theorem 5.1 shows how to calculate the optimal allocation.

Theorem 5.1. Given a budget of k nodes, we denote $X = Wexec(q)$, $Y_1 = Wd/B_c$, $Y_2 = Wd/B_s$, $\alpha = S/D$, the optimal number of cache nodes m^* and execution nodes n^* to minimize latency $l(\mathbb{Q})$ is:

$$m^* = \frac{k\sqrt{Y_1}}{\sqrt{X + Y_2(1 - k\alpha)} + \sqrt{Y_1}}, n^* = \frac{k\sqrt{X + Y_2(1 - k\alpha)}}{\sqrt{X + Y_2(1 - k\alpha)} + \sqrt{Y_1}} \tag{9}$$

Proof. After replacing n with $k - m$, the latency $l(\mathbb{Q})$ can be rewritten as:

$$l(\mathbb{Q}) = \frac{X + Y_2 - Y_2\alpha m}{k - m} + \frac{Y_1}{m}$$

Taking the derivative of $l(m)$ w.r.t. m and set it to 0 will find the critical point:

$$l^{'}(m) = \frac{X + Y_2(1 - k\alpha)}{(k - m)^2} - \frac{Y_1}{m^2} = 0$$

Solving this equation yields Eq. 9. We show that this critical point is a minimum by checking the second derivative of $l(\mathbb{Q})$:

$$l^{''}(m) = \frac{2(X + Y_2(1 - k\alpha))}{(k - m)^3} + \frac{2Y_1}{m^3} > 0$$

Here, we assume that the upper bound of cache size kS cannot accommodate all data segments D, i.e., there will be cache misses. Thus, $k\alpha < 1$. Because all system-related parameters are always positive, the critical point corresponds to minimal latency $l(\mathbb{Q})$. □

Example 5.1. Suppose we have a budget of $k = 20$ nodes. Each has 14GB cache capacity if used as a cache node and the dataset is 1TB. By profiling the cloud environment with $W = 500$ test queries, we observe that each node has a bandwidth of $B_c = 10$Gbit/s, and the network bandwidth of the cloud storage is $B_s = 3$Gbit/s. The average execution time per query is $exec(q) = 50ms$. Now, we can compute the optimal m^* and n^* using Eq. 9 as follows:

$$m^* = \frac{20\sqrt{25.6}}{\sqrt{25 + 85.3(1 - 20 * 0.015)} + \sqrt{25.6}} \approx 7, n^* = 20 - 7 = 13$$

Therefore, the optimal resource allocation is to configure approximately 7 cache nodes and 13 execution nodes to minimize the latency of the query workload. In Sec. 6.4, our experiment that varies the number of cache nodes validates its effectiveness.

6 Experiments

The experiments focus on answering the following questions:

- How does CacheServer work under real cloud DB workloads?
- How much does CacheServer gain under various workloads that have different number of queries, sizes of datasets, data segment sizes, and skewness factors?
- How much does CacheServer gain with respect to different scheduling strategies?
- How do the cache disaggregated architecture, prefetch, cache eviction, and resource allocation techniques we propose work?
- How do cache size and query processing speed affect the performance of CacheServer?

6.1 Experiment Setup

Experiment Environment. We implement a prototype system to evaluate CacheServer on the Google Cloud Platform (GCP). It consists of a coordinator, a set of cache nodes, and a set of query execution nodes. The coordinator distributes analytical queries to execution nodes using different scheduling methods evaluated in the experiments. For approaches that do not use CacheServer, we enable local cache by default, i.e., caching data segments on execution nodes for reuse.

Query Workloads and Data. Each data segment by default is 64MB, same as distributed file systems such as GFS/HDFS that are commonly used in cloud databases. We control the size of the datasets by varying the number of data segments.

Redset. We conduct experiments with the Redset [55] workload trace, a real-world analytical workload collected from the Amazon Redshift cloud data warehouse. It is complex and skewed and contains complex queries that access different numbers (2-15) of data segments and multiple types of analytical operations. We produce query loads based on two key metrics of Redset: (1) Data Access Patterns: it samples from the trace and maps the read data ID of each query to our synthetic data segments. This keeps the skewness and the data access patterns the same as the original trace; (2) Query Complexity: it makes sure that the generated queries have the same number of aggregation and join operators to the original trace. We also use the number of queries per minute in Redset to control the query arrival rate. Redset shows that the workload varies over time and frequently experiences peaks where queries queue up allowing CacheServer to fully use queued queries for prefetching.

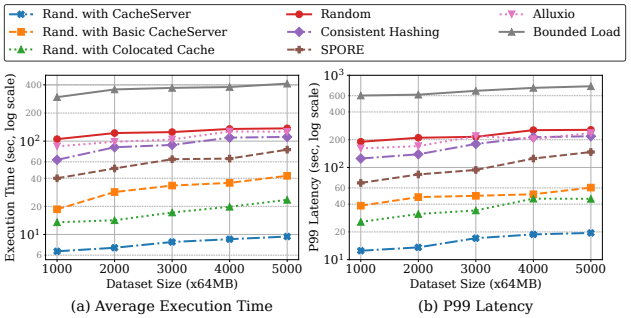
YCSB. We generate diverse workloads to evaluate the performance of CacheServer under different scenarios. Similar to other cloud DB works [21, 58, 61], we generate queries according to Yahoo! Cloud Serving Benchmark (YCSB) [19] to evaluate the performance of the `SCAN` operator under various data popularities. To increase complexity, we apply different analytical operations to the scanned data segments, such as `SUM`, `MIN/MAX`, `AVG`, and `SORT`. The query generator imposes different levels of skewness on the IDs of data segments, and thus controls their hotness. We use the Zipfian [27] distribution to control the distribution of the query workload and vary the skewness with a parameter θ. A larger θ indicates higher skewness and a θ close to 1 means low skewness [1].

Query Engine. We implement a prototype query execution engine, similar to FlexPushdownDB [61] and various analytic operators, while modifying the `SCAN` operator to fetch data from cache nodes. The query engine uses up all available vCPUs on execution nodes and processes queries in parallel.

Hardware Resources. We run experiments on the GCP with 20 `e2-highmem-4` compute nodes that could be configured as query execution nodes or cache nodes. Each node has 4 vCPUs and 32GB memory. Data segments are stored in a standard Google Cloud storage bucket. Each vCPU has around 1.2Gbps network bandwidth. When used as execution nodes, each node uses 4GB memory to cache data locally. This is because caching data locally on query execution nodes shares memory with the query execution engine, and the query engine potentially consumes a huge amount of memory when processing complex queries. When used as CacheServer nodes, each node uses 14GB memory for caching, since query execution is disaggregated from data caching.

Baselines. We compare CacheServer against *Alluxio-based* [41] implementation, *Colocated Cache* that colocate and share distributed caching service on execution nodes similar to Gopher [14], and *local cache* under various query workloads. We also implement a baseline called *Basic CacheServer* that only implements the partitioning to avoid network congestion on hot segments but does not use the prefetch and resource allocation techniques described in Sec. 4. We experiment these baselines under both random scheduling and affinity scheduling.

- *Random Scheduling with CacheServer*: Assign a query to the first available execution node and use CacheServer to cache data.



(a) Average Execution Time (b) P99 Latency

Fig. 3. Varying Dataset Sizes (64MB per Data Block)

• *Random Scheduling with Basic CacheServer*: Assign a query to the first available node and use *Basic CacheServer* to cache data.

• *Random Scheduling with Colocated Cache*: uses consistent hashing to distribute data segments to query execution nodes and cache them locally, similar to Gopher [14]. The query scheduler randomly distributes a query to an execution node and fetches the data segments that are potentially cached on other execution nodes through the fast intra-data center network. It thus leverages our high level idea that decouples affinity scheduling with cloud database to avoid load imbalance, yet reduce data transmission cost.

• *Random Scheduling*: Assign a query to the first available execution node and cache data locally on it.

• *Consistent Hashing* [35]: A typical affinity scheduling implementation in cloud DBs that use local cache.

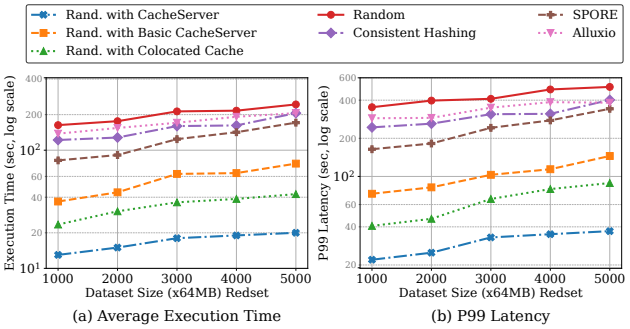
• *SPORE* [29]: An improved affinity scheduling that replicates hot data segments to mitigate load imbalance due to skewed workloads.

• *Alluxio*: Assign a query to the first available execution node and use Alluxio-based [41] implementation to cache data.

• *Bounded Load* [43]: Another improvement of consistent hashing that mitigate load imbalance by forwarding from an overloaded node to the next node. However, its performance is poor due to overflowing a sequence of nodes and replicating data segments all over nodes. Therefore, we only include it as a baseline in Fig. 3.

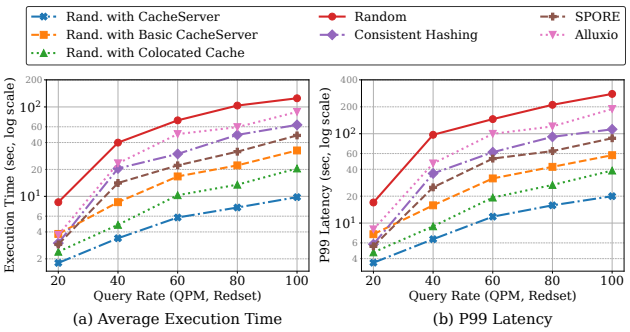
Moreover, besides random and affinity scheduling, we experiment *fairness* and *capacity* scheduling strategies w.r.t. CacheServer, basic CacheServer, Co-located Cache, and Alluxio, as further discussed in Sec. 6.3.

Configurations. By default, we use a dataset with 3,000 data segments. The default skewness parameter θ is set to 1.3. This results in approximately 80% of queries that request 20% of data (hot data). For SPORE, we use the configurations recommended in [29] with a hotness threshold of 2,000 and a replication factor of 1. This means that queries accessing the hottest data will be distributed to two nodes for load balancing. For the CacheServer partitioning discussed in Sec. 3.2, we divide a data segment into 8 partitions. CacheServer uses 13 execution nodes and 7 cache nodes which is founded as optimal by our resource allocation algorithm discussed in Sec. 5. For *Bounded Load*, we use a default balancing parameter $\epsilon = 0.3$ according to [43]. This allows most loaded node to have 1.3× query load than the average. For fairness scheduling, according to [25], we provision 4 users to share compute nodes. For capacity scheduling, similarly, we provision 4 users, allowing each to consume 20% to 60% compute nodes.



(a) Average Execution Time (b) P99 Latency

Fig. 4. Varying Dataset Sizes (64MB per Data Block, Redset)



(a) Average Execution Time (b) P99 Latency

Fig. 5. Varying the Query Arrival Rate

Metrics. Every 20 seconds we roughly submit 500 queries to the system, similar to the average number of queries per minute in Reset burst period. The experiments run 20k queries in total. We report the average query execution time, measures the average end-to-end latency of each query from the moment it is submitted to the coordinator until its completion, including queuing time, data transmission, and query processing. P99 latency represents the 99th percentile of these end-to-end query latencies, reflecting the slowest queries. We also report the cache hit rate and data transmission costs in the ablation study evaluating the prefetch efficiency and data transmission cost of CacheServer.

6.2 Evaluation with Various Workloads

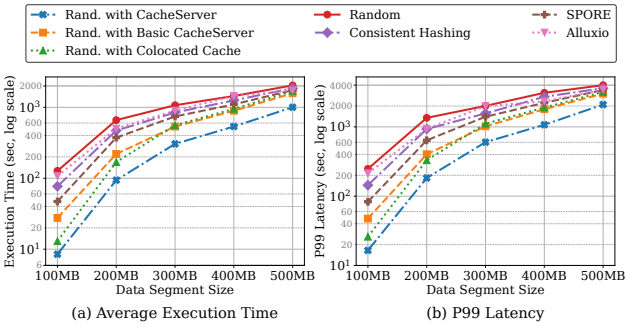
We evaluate the performance of *CacheServer* by varying the data scales and the properties of query workloads.

6.2.1 Varying dataset size

We vary the number of data segments, but keep the skewness factor of the workload fixed. This evaluates how the prefetch and cache hit rate of CacheServer scales under large datasets.

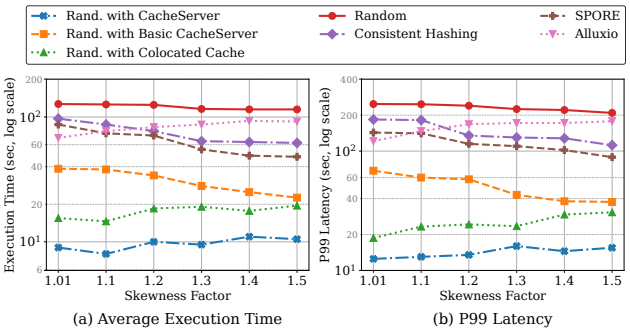
Fig. 3 shows the average query execution time and the tail latency under YCSB with default configuration. Moreover, we experiment with the real-world Redset workload and report the results in Fig. 4.

Random with CacheServer consistently outperforms all baselines by a large margin, from 3× up to 10×. For the YCSB workload with default configuration (Fig. 3), it outperforms *Random with*



(a) Average Execution Time (b) P99 Latency

Fig. 6. Varying Data Segment Sizes



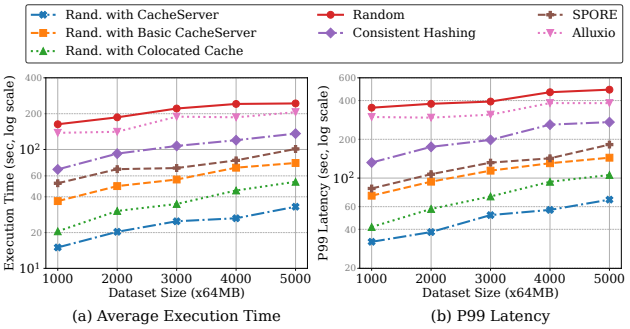
(a) Average Execution Time (b) P99 Latency

Fig. 7. Varying Workload Skewness

Colocated Cache by 2×, *Random with Basic CacheServer* by 3×, *SPORE*, *Consistent Hashing*, by 5×, *Random with Local Caching* and *Alluxio* by 10× in average execution time. The tail latency shows a similar trend in Fig. 3 (b). *Bounded Load*, which uses local cache, performs the worst: it is 50× slower than *Random with CacheServer* in average execution time and 30× slower in tail latency. Therefore, in the remainder of the experiments, we do not report its results to better visualize the trend of each method.

Under the Redset workload, Fig. 4 shows that *Random with CacheServer* is 50% faster than *Colocated Cache*, 2× faster than *Random with Basic CacheServer*, 6× faster than *SPORE*, *Consistent Hashing*, and 10× faster than *Alluxio* and *Random with Local Caching* in both average execution time and tail latency.

Random with Local Caching performs the worst. *Alluxio* is close to *Random with Local Caching* and much worse than other baselines. Although *SPORE* and *Consistent Hashing* outperform *Random with Loacal Caching*, it does not show advantages over *Random with CacheServer*. When distributing queries, *Random with Local Caching* does not consider data locality and thus has the lowest cache hit rate. While *Alluxio* provides a basic cache service similar to CacheServer, it suffers from the lack of optimizations of partition-based load balancing, query-aware prefetching, and resource allocation. Although *SPORE* and *Consistent Hashing* achieves high cache hit rate with affinity scheduling, they suffer from skewed workloads. *SPORE* slightly outperforms *Consistent Hashing* since it is able to replicate hot data segments and mitigate workload imbalance to some degree. However, it is still much worse than *Random with Basic CacheServer*, because using a fixed number of nodes to host hot data segments cannot perfectly balance load across nodes.



(a) Average Execution Time (b) P99 Latency

Fig. 8. Varing Dataset Size under Workload with Different Query Complexities

Although *Random with Basic CacheServer* and *Colocated Cache* outperforms *Consistent Hashing*, *Random with Local Caching*, and *SPORE*, they are slower than *Random with CacheServer*. This shows: (1) the advantage of the CacheServer architecture in achieving data locality while not introducing load imbalance and (2) the effectiveness of our data prefetch strategy that maximize cache hit rate.

Cache Hit Rate. We report the cache hit rate on the Redset workload: *Random with CacheServer*: 0.93, *Random with Colocated Cache*: 0.84, *Random with Basic CacheServer*: 0.84, *Consistent Hashing*: 0.86, *SPORE*: 0.82, *Random with Alluxio*: 0.89, *Random with Local Caching*: 0. *Random with CacheServer* achieves the best cache hit rate using the dedicated cache space and predictive prefetch. Similarly, *Random with Basic CacheServer* and *Random with Alluxio* benefit from dedicated cache space but has lower cache hit rate due to the lack of prefetch. *Random with Colocated Cache* is close to CacheServer because we design it to use a prefetch similar to CacheServer. For *Consistent Hashing* and *SPORE*, the local cache is not large enough on hot node and the cache hit rate is thus lower. The *Random with Local Caching* has the lowest cache hit rate close to 0 as it provides no data locality.

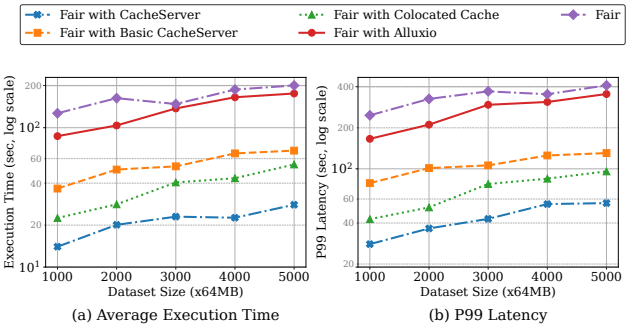
6.2.2 *Varying the Query Arrival Rate*

We vary the size of the query workloads by tuning the number of queries per minute derived from Redset and use the default setup otherwise. Since the CacheServer prefetch technique is designed to take advantage of queued queries, this experiment evaluates how the performance of CacheServer scales with the number of queries and how much the augmented prefetch technique contributes to the performance gain of CacheServer.

As shown in Fig. 5 (a) and Fig. 5 (b), *Random with CacheServer* outperforms all baselines. Under light workload with small queue, the augmented *Random with CacheServer* is still 30% faster than *Colocated Cache*, *Consistent Hashing*, and *SPORE*, 2× faster than *Basic CacheServer* and *Alluxio*, and 5× faster than *Random* in both average execution time and tail latency. Moreover, the more the queries, the larger CacheServer wins, confirming its scalability to query workload as well as the effectiveness of the augmented prefetch technique. Under heavy workload, the augmented CacheServer shows a similar advantage as our other experiments, and is 2× – 10× faster than baselines.

6.2.3 *Varying Data Segment Sizes*

We evaluate how *CacheServer* performs when processing queries accessing multiple data segments using variable-sized data segments. We vary the size from 100MB to 500MB according to the size of micro-partition in Snowflake [21] and depict the results in Fig. 6. When the size is 100MB, similar to the above experiments, *Random with CacheServer* is 50% faster than *Colocated Cache*, 3× faster than *Random with Basic CacheServer*, 5× faster than *SPORE* and *Consistent Hashing*, and 10× faster



(a) Average Execution Time (b) P99 Latency

Fig. 9. Fairness Scheduling with Redset

than *Alluxio* and *Random with Local Caching* in average execution time and tail latency. We observe that the size variance negatively impacts all baselines, including those scheduling with *CacheServer*. This is because the increase of segment size tends to reduce cache hit rate, thus causing larger data transmission cost. Nevertheless, *Random with CacheServer* consistently outperforms *Colocated Cache* and *Random with Basic CacheServer* by at least 1.5× and the rest of baselines by 2×.

6.2.4 *Varying Workload Skewness*

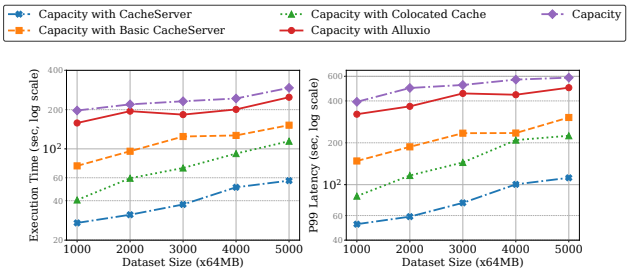
We evaluate how all methods perform under different levels of workload skewness. We vary the skewness parameter θ, but use the default data and query configurations for the rest.

Fig. 7 shows the results on the YCSB workloads. In all cases, *Random with CacheServer* outperforms other methods from 60% up to 10×, even if the workload is close to uniform. When the level of skewness gets larger, the execution time and the tail latency of *SPORE*, *Consistent Hashing* and *Random with Basic CacheServer* decrease. This is because when the skewness is low, the workload is more likely to access the large portion of less frequent data. For *SPORE* and *Consistent Hashing*, this leads to a higher cache miss rate on each compute node. Similarly, *Random with Basic CacheServer* incurs more cache misses due to the lack of the prefetch technique. As the level of skewness increases, the workload concentrates more on hot data segments. This leads to high cache hit rate for *SPORE* and *Consistent Hashing*. *SPORE* outperforms *Consistent Hashing* because it mitigates load imbalance of affinity scheduling to some degree. *Random with Basic CacheServer* outperform *SPORE* and *Consistent Hashing* due to the remote cache architecture that achives high cache hit rate without introducing load imbalance. However, it does not perform as good as *Random with CacheServer* due to the lack of prefetch that maximizes cache availability. *Colocated Cache* performs better than other baselines, since it leverages our idea that decouples affinity scheduling with cloud database to avoid load imbalance.

6.2.5 *Varing Dataset Size under Workload with Different Query Complexities*

This experiment evaluates CacheServer's performance under complex queries. To achieve this, we generate a new workload composed of queries that access the same data segments as the YCSB workload but involves operators with various complexities. We then experiment with this workload on different dataset sizes. As shown in Fig. 8, *Random with CacheServer* still outperforms all baselines at least 2×, showing the superiority of CacheServer in handling queries with high computation complexity.

Summary. The results confirm the advantage of the CacheServer architecture and the effectiveness of our data prefetch techniques under different scales of datasets and various types of workloads.



(a) Average Execution Time (b) P99 Latency

Fig. 10. Capacity Scheduling with Redset

6.3 Resource Capacity-based Scheduling

In this section, we evaluate the performance of *CacheServer* with fairness scheduling and capacity scheduling which are popular in a multi-tenant environment. We vary the scales of datasets Redset workloads.

6.3.1 *Fairness Scheduling*

Fig. 9 shows the average query execution time and the tail latency under the default dataset configuration. *Fairness with CacheServer* is 50% faster than *Fairness with Colocated Cache*, 3× faster than *Fairness with Basic CacheServer*, and is 6× faster than *Fairness with Alluxio* and *Fairness with Local Caching*.

Fairness with Local Caching performs the worst, because fairness scheduling distributes queries based on the current resource usage of each user, rather than ensuring data locality. This leads to low cache hit rate, thus large data transmission cost. *Fairness with Alluxio* performs close to *Fairness with Local Caching* due to the lack of partition-based load balanceing. *Fairness with Colocated Cache* and *Fairness with Basic CacheServer* outperforms *Fairness with Local Caching* and improves cache hit rate, because execution nodes now access data segments through distributed cache. *Fainess with CacheServer* performs even better due to the data prefetch and eviction techniques that further improve cache hit rate.

6.3.2 *Capacity Scheduling*

we evaluate how *CacheServer* improves the performance of capacity scheduling under Redset workloads. As shown in Fig. 10, all methods demonstrate a similar trend in execution time and tail latency. In all cases, *Capacity with CacheServer* outperforms other methods from 50%× to 10× for the similar reason discussed in Sec. 6.3.1

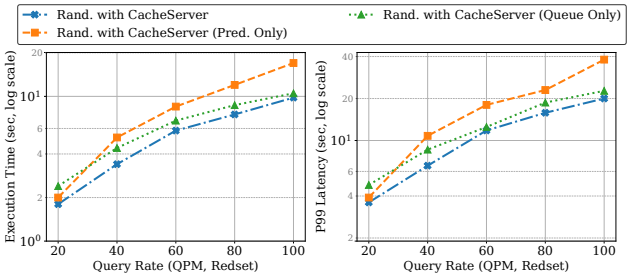
Summary. The experiment results confirm that *CacheServer* consistently and significantly improves the performance of fairness and capacity scheduling in a multi-tenant environment.

6.4 Ablation Study

In this section, we conduct ablation studies to evaluate the benefits of different components of *CacheServer*.

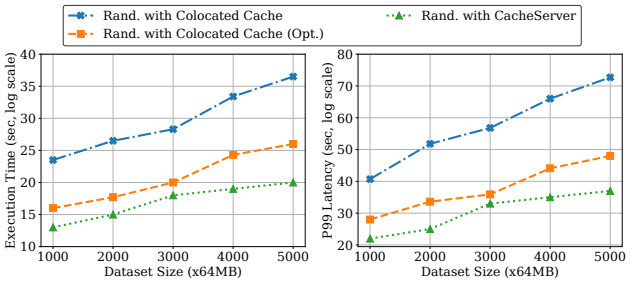
6.4.1 *CacheServer Predictive Prefetch*

We compare the augmented CacheServer against the prediction-only CacheServer and our original queue-only CacheServer. As shown in Fig. 11, when the query arrival rate is low, the augmented CacheServer and the prediction-only CacheServer are 30% faster than the queue-only CacheServer. However, as the query arrival rate increases, the performance of the prediction-only CacheServer degrades. It is 2× and 80% slower than the augmented CacheServer and the queue-only CacheServer



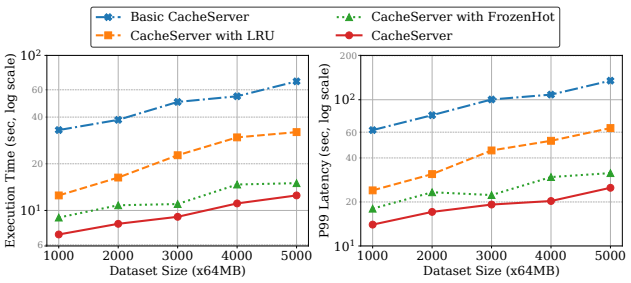
(a) Average Execution Time (b) P99 Latency

Fig. 11. Comparing with Predictive Prefetching



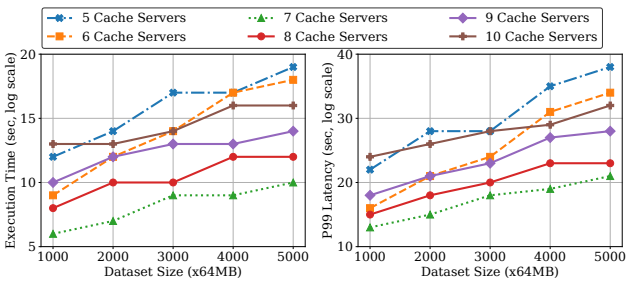
(a) Average Execution Time (b) P99 Latency

Fig. 12. Comparing with Colocated Caching



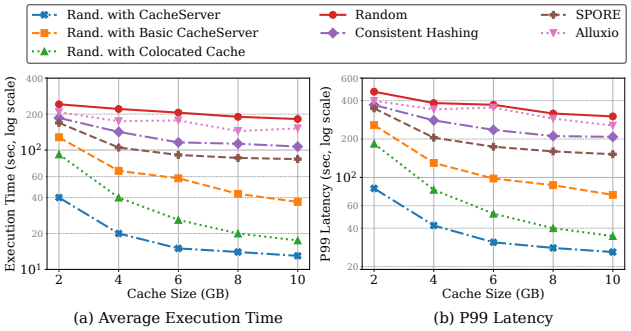
(a) Average Execution Time (b) P99 Latency

Fig. 13. CacheServer with Different Caching Techniques



(a) Average Execution Time (b) P99 Latency

Fig. 14. Varying the Number of CacheServer Nodes



(a) Average Execution Time (b) P99 Latency

Fig. 15. Varying Cache Size

because more queries queue up and thus the prefetch mechanism can leverage more information of upcoming queries to make better prefetch decisions.

6.4.2 *CacheServer Architecture*

We implement a new baseline called *Colocated Cache (Opt)*, a *local cache version* of our CacheServer. It improves *Colocated Cache* with the key technical ideas proposed in our work to mange the cache with a "global view". In particular, it partitions data segments to balance network traffic across nodes (Sec. 3.2). It also uses our score-based strategy (Sec. 4) to prefetch/evict the most/least valuable segments to maximize cache hit rate.

As shown in Fig. 12, *Colocated Cache (Opt)* is still 20% slower than *CacheServer*, because fetching data from other execution nodes interferes with query execution. In particular, in our experiment, we observe that *Colocated Cache (Opt)* incurs a 80% data transmission speed drop compared with CacheServer, i.e., dropping from 210MB/s to 117MB/s. This shows one of the fundamental advantages of CacheServer that disaggregates data cache and query execution.

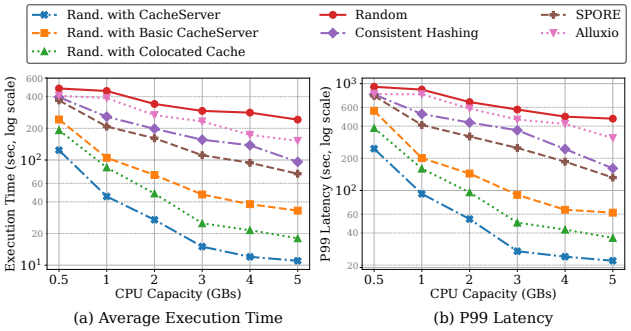
6.4.3 *CacheServer Caching Techniques*

We evaluate our prefetch and cache eviction techniques. We compare *CacheServer* with three variants: (1) the *Basic CacheServer* which only implements the basic function of dedicated cache nodes; (2) the *CacheServer with FrozenHot* [48] that identifies and retains hot data segments in cache to reduce promotion and locking overhead; and (3) *CacheServer with LRU* that implements the basic function and support prefetch, but uses LRU to evict cache. We use random scheduling and YCSB in this experiment.

As shown in Fig. 13, *CacheServer* consistently outperforms *Basic CacheServer*, *CacheServer with ForzenHot* and *CacheServer with LRU*. In particular, *CacheServer* is about 3× faster than *Basic CacheServer*, 1.7× faster than *CacheServer with LRU*, and 20% faster than *CacheServer with ForzenHot* in average execution time. In terms of tail latency, *CacheServer* outperforms *Basic CacheServer* by 5×, *CacheServer with LRU* by 2×, and *CacheServer with ForzenHot* by 25%. *Basic CacheServer* performs the worst due to the lack of prefetch, Although *CacheServer with LRU* and *CacheServer with FrozenHot* implements prefetching, they use the simple LRU to evict cache, often incorrectly pruning the segments that are less hot but will be accessed in the near future. *CacheServer*, on the other hand, is able to accurately prefetch and keep data segments to be accessed in the near future, leading to the best performance.

6.4.4 *CacheServer Resource Allocation*

Next, we evaluate how CacheServer resource allocation affects query latency. We use in total 20 nodes and vary the number of cache nodes from 5 to 10. We use random scheduling and the YCSB default workload.



(a) Average Execution Time (b) P99 Latency

Fig. 16. Varying the Query Processing Capacity

As shown in Fig. 14, configuring 7 nodes as cache nodes achieves the best performance. It is around 1.5× faster than using 5 or 10 cache nodes in average execution time and tail latency. This is because when using too few cache nodes, the limited cache capacity bandwidth reduces cache hit rate, increasing data transmission cost. On the other hand, using too many caches nodes weakens the computation power of query execution engine dramatically, leading to long query execution time.

6.4.5 *CacheServer Cache Size*

In this experiment, we evaluate how the cache capacity of each node affects the query latency. We use the same cache size for the nodes in local cache compute and the cache nodes in CacheServer and vary it from 2GB to 10GB. We use random scheduling and the YCSB default workload.

As shown in Fig. 15, when the cache size is under 6GB, *Random with CacheServer* is 2× faster than *Random with Colocated Cache* and *Random with Basic CacheServer*, 3× faster than *SPORE* and *Consistent Hashing*, and 4× faster than *Random with Alluxio* and *Random with Local Caching* in average execution time; and is 50% faster than *Random with Colocated Cache*, 2.5× faster than and *Random with Basic CacheServer*, 3× faster than *SPORE* and *Consistent Hashing*, and 5× faster than *Random with Local Caching* in tail latency.

When the cache size increases, the performance of all methods improves, as a larger cache size reduces cache misses. However, *Random with Local Caching* benefits little due to the lack of data locality guarantee. Although the affinity scheduling-based *SPORE* and *Consistent Hashing* benefits from larger cache sizes, they suffer from load imbalance due to workload skewness. Similarly, *Random with Alluxio* suffers from network congestion due to the lack of partition-based balance. *Random with CacheServer* outperforms them by at least 5× in average execution time and tail latency. *Random with Colocated Cache* performs worse than *Random with CacheServer*, but significantly outperforms other approaches, confirming the superiority of the CacheServer architecture and its prefetch and cache eviction techniques.

6.4.6 *Query Processing Capacity*

We study how the query processing capacity of execution nodes impacts query latency by controlling the computation time required to process a fixed-size data segment. We vary the query processing capacity from 0.5GB/s to 5GB/s.

As shown in Fig. 16, *Random with CacheServer* consistently outperforms all baselines, similar to our previous experiments. With the lowest CPU capacity, *Random with Local Caching* performs the worst due to its high data transmission cost. The performance of *Consistent Hashing*, *SPORE* and *Random with Alluxio* is close to that of *Random with Local Caching*, because low query processing capacity worsens load imbalance caused by skewed workload.

All approaches benefit from higher query processing capacity thus lower query execution time. Because *Random with CacheServer* balances the load, it benefits the most from higher CPU capacity, resulting in a larger gain compared to other baselines. *Random with Colocated Cache* and *Random with Basic CacheServer* outperforms *Consistent Hashing* and *SPORE* due to the inherent load balance merit. However, *Random with Basic CacheServer* is slower than *Random with CacheServer* due to lack of data prefetch and *Random with Colocated Cache* is slower than *Random with CacheServer* due to the lower cache hit rate. As expected, higher query processing capacity improves the performance of *Consistent Hashing* and *SPORE*, because this mitigates their load imbalance disadvantage. However, they are still 10× slower than *Random with CacheServer*.

7 Related Work

Caching in Cloud Data Systems. Caching data locally on compute nodes is popular in storage-disaggregated cloud databases. However, most of these systems couple local caching with affinity scheduling, including Snowflake [21], Aurora [56, 57], Amazon Redshift [11, 28], Amazon Athena [2], Google BigQuery [3], and Microsoft Azure Synapse [4], suffering from load imbalance due to skewed query workloads.

Distributed data storage systems such as Memcached [23] and Memcached-based systems [17, 46, 64], Apache Cassandra [39], and Linked-in Voldemort [38] caches data in memory to reduce the disk I/O cost. They also use affinity scheduling, i.e., consistent hashing, to distribute data items to storage nodes and look up them. However, these storage systems are agnostic to query execution, thus not handling load imbalance due to workload skewness.

ClickHouse [6] provides a shared caching layer for object storage to accelerate data transfer via parallel reading. However, unlike CacheServer, as a generic caching layer for object storage, it is not optimized for query execution in cloud DBs, i.e., addressing load imbalance and query-aware data prefetch.

Alluxio [41, 54] is a distributed file system that provides a unified namespace and caching layer for users to easily access data stored in different storage systems. Therefore, it focus on user friendly programming interfaces instead of the efficiency of data analysis. Although it offers caching functionality, it is limited to maintaining intermediate results through lineage to avoid redundant computation, rather than reducing the data transmission cost of raw data.

Specialized hardware such as Remote Direct Memory Access (RDMA) and Compute Express Link (CXL) has been used in systems that require frequent data exchanges between compute nodes. In cloud databases context, some works such as DSM-DB [59], LegoBase [65], Active-Memory Replication [63] SMP RDBMS [40] use RDMA to increase the memory capacity of compute nodes, and reduce the concurrency control and fault tolerance overheads for OLTP workloads. PolarDB[13] targets CPU-memory disaggregation by separating hardware resources into dedicated CPU, memory, and storage pool. Memory-centric computing via memory pooling[18] similarly uses CXL over PCIe to share memory across nodes. In contrast, CacheServer disaggregates data caching from query execution, thereby decoupling cloud DBs from affinity scheduling to avoid load imbalance. However, CacheServer readily benefits from such new hardware to reduce the data transmission cost between cache nodes and execution nodes.

S3-FIFO [60] demonstrates that a simple FIFO-based cache replacement policy can outperform LRU, particularly in workloads where objects are accessed once and rarely accessed again. HPCache[45] actively monitors the system and introduces a dynamic cache management strategy that allocates cache space based on its potential to speed up queries. CacheServer, in contrast, focuses on the architectural design of cloud DBs that decouples data caching from query scheduling.

Storage for Serverless Platforms. Pocket [37] is a distributed cloud storage layer that caches metadata and temporal intermediate results for serverless functions. Pocket focuses on automatically

rightsizing resources such as CPU cores and storage capacity to minimize cost. To optimize stateless compute architecture, Polaris [8] and Skyrise [12] externalize state information such as meta data, transaction logs, and intermediate results to remote object storage. Unlike these works, CacheServer focuses on optimizing data caching in cloud DBs, achieving high cache hits while not suffering from load imbalance under skewed workload. Crystal [22] focuses on semantic caching with filters, i.e., caching intermediate results. Unlike CacheServer, it caches data on execution nodes.

Scheduling in Cloud Data Systems. Several methods have been proposed to mitigate the load imbalance issue in cloud databases caused by affinity scheduling under skewed workloads. Sparrow [47] randomly chooses several nodes from all the nodes and schedule queries to one of the chosen node. Bounded Load [43] replicates hot data segments to multiple successor nodes and steers hot queries to these nodes. Slicer [7] is a sharding service that hashes service requests to a new key space and balances server load by manipulating the server's key range and distributes hot requests to multiple servers. Snowflake [21] mitigates the impact of workload skewness through file stealing. When a node finishes scanning its data, it requests from remote storage additional data that is supposed to be processed by other slower nodes. Rather than paying extra efforts to clean the mess caused affinity scheduling, CacheServer instead decouples affinity scheduling from cloud databases, thus eliminating the root cause of load imbalance while still taking the advantage of data cache.

In addition many works have focused on resource fairness and capacity-based scheduling. Dominant Resource Fairness (DRF) [25, 26] identifies the resource demands of jobs and allocates resources to jobs based on their dominant resource usage. Quincy [31] uses a centralized graph representation to transfer job fairness into a min-cost flow problem. Mercury [34] extends distributed scheduling to consider resource fairness. Cheng et al. [16] and Zaharia et al. [62] propose different capacity scheduling mechanisms to make sure each job has a guaranteed number of node capacity. Since nodes could have different CPU, memory and disk [49], the capacity scheduling [42, 66] takes this into consideration to produce better scheduling plans. Although these works improve resource fairness and capacity utilization, they cannot fully benefit from caching data locally on compute nodes due to the weak data locality guarantee, leading to low cache hit rate. CacheServer, decoupling caching and affinity scheduling, makes it easy to integrate these scheduling mechanisms. Therefore, cloud databases are able to meet different scheduling objectives while still enjoying the benefit of caching.

8 Conclusion

We present CacheServer, a system that decouples data caching from query execution with dedicated cache nodes to manage data caching. This architecture allows cache nodes to be shared by execution nodes and brings the benefit of data locality to scheduling strategies beyond affinity scheduling. We further optimize CacheServer with *predictive prefetching* that maximizes the availability of data segments and *latency-aware resource allocation* that allocates resources between execution and cache nodes to minimize the query latency. Our evaluation on various workloads and different scheduling strategies shows that CacheServer outperforms alternatives from 2× to 10× in execution time and tail latency.

Acknowledgments

This research was supported in part by NSF under grants DBI-2327954 and CCF-2106621 and OAC-2608717.

References

[1] 2023. NumPy. https://numpy.org/doc/stable.

[2] 2025. Amazon Athena. https://aws.amazon.com/athena/.

[3] 2025. Google BigQuery. https://docs.cloud.google.com/bigquery/docs/introduction.

[4] 2025. Microsoft Azure Synapse Analytics. https://azure.microsoft.com/en-us/products/synapse-analytics/.

[5] 2025. Scylla DB. https://java-driver.docs.scylladb.com/stable/manual/core/load_balancing/.

[6] 2026. ClickHouse. https://clickhouse.com/blog/building-a-distributed-cache-for-s3#stage-3-the-distributed-cache.

[7] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer:{Auto-Sharding} for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 739–753.

[8] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, et al. 2020. POLARIS: the distributed SQL engine in azure synapse. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3204–3216.

[9] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. 2011. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*. 287–300.

[10] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[11] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[12] Thomas Bodner, Daniel Ritter, Martin Boissier, and Tilmann Rabl. 2025. Skyrise: Exploiting Serverless Cloud Infrastructure for Elastic Data Processing. *Datenbank-Spektrum* 25, 1 (2025), 29–38.

[13] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.

[14] Guohai Zhang Xin Tang Qingchen Chang, Huanchen Zhang Kai Hwang Yuesen Li, Runhuai Huang, Teng Wang Wusheng Zhang Ming Zhang, and Qingchun Chen Xiaodong Hou Qian Wang. [n. d.]. A Low Latency Cache for Cloud RDBMs. *Proceedings of the VLDB Endowment. ISSN* 2150 ([n. d.]), 8097.

[15] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, et al. 2019. Procella: Unifying serving and analytical data at YouTube. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2022–2034.

[16] Jilan Chen, Dan Wang, and Wenbing Zhao. 2013. A task scheduling algorithm for Hadoop platform. *Journal of Computers* 8, 4 (2013), 929–936.

[17] Yue Cheng, Aayush Gupta, and Ali R Butt. 2015. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[18] Yannis Chronis, Anastasia Ailamaki, Lawrence Benson, Helena Caminal, Jana Gičeva, Dave Patterson, Eric Sedlar, and Lisa Wu Wills. 2025. Databases in the era of memory-centric computing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[19] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[20] János Csirik, Johannes Bartholomeus Gerardus Frenk, Martine Labbé, and Shuzhong Zhang. 1991. Heuristics for the 0-1 min-knapsack problem. *Acta Cybernetica* 10, 1-2 (1991), 15–20.

[21] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[22] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: a unified cache storage system for analytical databases. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2432–2444.

[23] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[24] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the thirteenth EuroSys conference*. 1–13.

[25] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX symposium on networked systems design and implementation (NSDI 11)*.

[26] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2013. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 365–378.

[27] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. 243–252.

[28] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[29] Yu Ju Hong and Mithuna Thottethodi. 2013. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–17.

[30] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. 2014. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. 1–7.

[31] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 261–276.

[32] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph Seffi Naor, and Sriram Rao. 2018. Netco: Cache and i/o management for analytics over disaggregated stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 186–198.

[33] Sultana Kalid, Ali Syed, Azeem Mohammad, and Malka N Halgamuge. 2017. Big-data NoSQL databases: A comparison and analysis of "Big-Table","DynamoDB", and "Cassandra". In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*. IEEE, 89–93.

[34] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 485–497.

[35] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[36] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. 1999. Web caching with consistent hashing. *Computer Networks* 31, 11-16 (1999), 1203–1213.

[37] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 427–444.

[38] Jay Kreps. 2012. Project Voldemort.

[39] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[40] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R Narasayya. 2016. Accelerating relational databases by leveraging remote memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data*. 355–370.

[41] Haoyuan Li. 2018. *Alluxio: A virtual distributed file system*. University of California, Berkeley.

[42] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 450–462.

[43] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. 2018. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 587–604.

[44] Michael Mitzenmacher. 2002. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2002), 1094–1104.

[45] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. Hpcache: memory-efficient OLAP through proportional caching. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 1–9.

[46] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.

[47] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 69–84.

[48] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. 2023. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 557–573.

[49] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*.

1–13.

[50] Sartaj Sahni. 1975. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)* 22, 1 (1975), 115–124.

[51] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, et al. 2018. F1 query: Declarative querying at scale. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1835–1848.

[52] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[53] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing a cloud DBMS: architectures and tradeoffs. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2170–2182.

[54] Chunxu Tang, Bin Fan, Jing Zhao, Chen Liang, Yi Wang, Beinan Wang, Ziyue Qiu, Lu Qiu, Bowen Ding, Shouzhuo Sun, et al. 2024. Data Caching for {Enterprise-Grade}{Petabyte-Scale}{OLAP}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 901–915.

[55] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3694–3706.

[56] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[57] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, et al. 2018. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*. 789–796.

[58] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 449–462.

[59] 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. *Proc. VLDB Endow.* 16, 1 (2022), 15–22.

[60] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 130–149.

[61] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms. *Proceedings of the VLDB Endowment* 14, 11 (2021).

[62] Matei Zaharia. 2009. Job scheduling with the fair and capacity schedulers. *Hadoop Summit* 9 (2009), 592.

[63] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking database high availability with RDMA networks. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1637–1650.

[64] Xiaoyi Zhang, Dan Feng, Yu Hua, Jianxi Chen, and Mandi Fu. 2018. A write-efficient and consistent hashing scheme for non-volatile memory. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[65] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, et al. 2021. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1900–1912.

[66] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of the VLDB Endowment*, Vol. 7. VLDB Endowment Inc., 1393–1404.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026