HotHash: Hotness-Aware Consistent Hashing for Cloud Databases

JUNYONG ZHAO, University of Arizona, USA
JIA YUAN, University of Arizona, USA
ZUI CHEN, MIT, USA
SAMUEL MADDEN, MIT, USA
LEI CAO, University of Arizona, USA

Cloud databases often use *consistent hashing* to schedule queries because of its data locality guarantee – scheduling the queries accessing the same data segment to the same node. This makes optimizations such as *caching* effective in reducing data I/O costs. However, consistent hashing causes load imbalance when handling skewed workloads and can lead to large query latencies. In this paper, to address this limitation, we propose HotHash, a technique that offers strong data locality and load balancing guarantees, while still preserving the key properties of consistent hashing, e.g., robustness to node changes. HotHash achieves these objectives with two key ideas: (1) range hashing that takes data hotness into consideration and (2) virtual hash ring that introduces randomness into query scheduling. More specifically, rather than mapping one data segment to one single node, *range hashing* maps it to a range of the hash ring where its length is proportional to the hotness of this data item; and it achieves so with *one single hash*. Furthermore, HotHash uses a *virtual hash ring* where the locations of nodes in the hash ring are randomized for each data segment, which randomizes nodes caching each data segment while preserving data locality for a given item. We show that HotHash is robust to node changes in that it still uses the same principles of consistent hashing to map data to the nodes. Our experimental evaluation on various workloads shows that HotHash is 1.4× to 150× faster than the state-of-the-art in average execution time and tail latency.

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

Additional Key Words and Phrases: Cloud Databases, Consistent Hashing, Load Balance

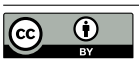
ACM Reference Format:
Junyong Zhao, Jia Yuan, Zui Chen, Samuel Madden, and Lei Cao. 2026. HotHash: Hotness-Aware Consistent Hashing for Cloud Databases. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 196 (June 2026), 27 pages. https://doi.org/10.1145/3802073

1 Introduction

Motivation. Modern cloud databases broadly adopt a *disaggregated storage architecture* that separates compute nodes from data storage nodes. This offers the flexibility to scale the computation and data storage separately. However, when running queries, cloud databases have to transmit data from remote storage to compute nodes through the network, which typically has lower bandwidth than local disks. This tends to substantially increase query latency.

A common solution to address this data transmission bottleneck is *caching*, where compute nodes keep data segments for subsequent queries to reuse. To achieve a high cache hit rate, cloud databases are typically coupled with *affinity scheduling* which distributes queries accessing the

Authors' Contact Information: Junyong Zhao, University of Arizona, USA, junyong@arizona.edu; Jia Yuan, University of Arizona, USA, jiayuan@arizona.edu; Zui Chen, MIT, USA, magolorcz@gmail.com; Samuel Madden, MIT, USA, madden@csail.mit.edu; Lei Cao, University of Arizona, USA, lcao@csail.mit.edu.



ACM 2836-6573/2026/6-ART196
https://doi.org/10.1145/3802073

same data segment to the same node, ensuring *data locality*. Consistent hashing [21, 22] is widely used for *affinity scheduling* in cloud databases [11, 20, 34, 44] due to its robust handling of node changes. This elasticity is crucial for dynamic environments where the number of nodes changes frequently.

However, problems arise when using consistent hashing or other affinity scheduling schemes with skewed workloads, where certain data segments are disproportionately popular. In real-world applications, such imbalanced workloads are common, often due to temporal or spatial factors. For example, viral social media posts can quickly attract massive numbers of views. Such workloads are challenging with consistent hashing as many queries for the same hot data segment are directed to a single node, causing it to become overloaded and straggle. This *load imbalance* on compute nodes results in increased *tail latency* and degraded system performance.

Objectives. As further discussed in Sec. 7, there are works handling stragglers caused by load imbalance or other reasons, such as file stealing. Orthogonal to these efforts that target *mitigating the consequence* of load imbalance, in this work, we propose HotHash, an enhancement to consistent hashing that *eliminates load imbalance* introduced by affinity scheduling under skewed workloads. HotHash targets two objectives: (1) ensuring data locality and load balancing with *theoretical guarantees* and (2) ease of adoption.

Challenges. Challenges to achieve these objectives include:

- **The Conflict Requirements of Load Balance and Data Locality.** Intuitively, *random scheduling*, another fundamental query scheduling strategy that randomly distributes queries to different compute nodes, will ensure load balance. However, it is very likely to distribute the queries accessing the same data segment to different nodes, leading to poor data locality and in turn low cache hit rate. This, nevertheless, is exactly what cloud databases try to avoid by adopting affinity scheduling, e.g., consistent hashing, which on the contrary, sacrifices load balance to guarantee data locality. It is thus challenging to simultaneously achieve these two goals that seem conflicting with each other.
- **Compatible with Consistent Hashing.** Second, to effectively balance the two conflicting requirements will inevitably introduce extra complexity to consistent hashing. However, for ease of adoption, the new strategy should be compatible with existing consistent hashing-based strategies. That is, it should perform in a similar way to consistent hashing and not require major changes to existing cloud databases, e.g., using a hash function to map queries (data segments) to nodes; and it has to be robust to node additions or removals. That is, when a node joins or leaves, the system only has to move around a small amount of data, ideally only the data segments on one single node, just as in consistent hashing, while still guaranteeing data locality and load balance. This is challenging.

To the best of our knowledge, no work has addressed the above problems to make consistent hashing a better match to cloud databases. In other areas such as networking and web services, some works [7, 27] use the idea of *Bounded Load* to address the load imbalance problem in consistent hashing. Such designs first set an upper bound on the load of any compute nodes. When consistent hashing picks a node where its load is higher than this bound, Bounded Load will forward a service request to another node. Although Bounded Load balances the load on nodes, it overlooks data I/O cost. This is because distributing queries based on a hard load constraint on the nodes regardless of the hotness of their data tends to replicate cold data segments, e.g., when a cold data segment is mapped to an overloaded node, thus leading to high I/O costs.

Proposed approach. HotHash addresses these challenges with two key techniques: (1) *range hashing* that leverages the hotness of the data segments and (2) *virtual hash rings* that introduce randomness into query distribution.

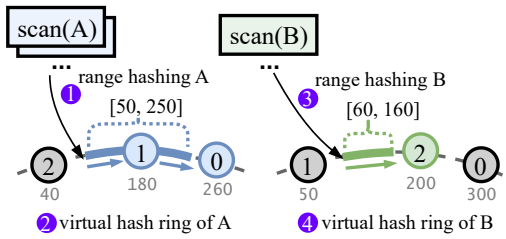


Fig. 1. Overview of HotHash

Unlike consistent hashing which maps each data segment to *one single location* on the ring, *range hashing* maps one data segment to a *range* of the hash ring, where its start point is decided by consistent hashing and its length is *proportional to* the hotness of the data. The queries accessing this data segment will only be distributed among the nodes that own a part of this range on the hash ring, thus preserving data locality. Range hashing improves load balancing by distributing queries that access hotter data segments to more nodes. Furthermore, because the hotness of a data segment determines the number of nodes that potentially could host it, range hashing avoids replicating cold data, thus not suffering from the high I/O cost issue of Bounded Load. The example in Fig. 1 shows the results of range hashing two data segments A and B. Because A is hotter than B, it is mapped to a larger range ($[50, 250]$) than B ($[60, 160]$). Thus, queries accessing A will be distributed to a node group of size two, whereas queries accessing B only use a single node.

However, range hashing, which replicates hot data segments to neighboring nodes on the hash ring, introduces correlations between nodes on their workloads. More specifically, if a node hosts two data segments A and B, its neighboring nodes will have a high probability to host A and B as well. This correlation between the workloads on neighboring nodes compromises the inherently random nature of hashing, increasing the *collision probability* of queries that access different data segments. This could lead to a less balanced distribution of queries, thus degrading query latency.

We introduce *virtual hash rings* to address this issue. In HotHash, different data segments see different virtual hash rings, where each ring corresponds to *a random permutation* of the nodes. This randomness introduced by virtual hash rings reduces collision probability when mapping the queries accessing different data segments to *physical nodes*. In this way, even if two data segments are mapped to largely overlapping ranges, the nodes that these two ranges cover do not necessarily overlap. Essentially, this avoids many queries accumulating on certain nodes. Moreover, HotHash still distributes queries accessing the same data segment to nodes that are *logically adjacent*, allowing it to identify nodes that have a copy of a data segment with a single hash. Fig. 1 shows the two different virtual hash rings w.r.t. data segments A and B. Although the hash values of A and B are close (50 and 60), the nodes hosting queries accessing A ($node_1$ and $node_0$) and B ($node_2$) do not overlap.

Combining range hashing and virtual hash ring, HotHash seamlessly unifies the merits of affinity scheduling and random scheduling, offering strong theoretical guarantees for both load balancing and data locality, as we describe in Sec. 5. Because HotHash still uses the idea of hash ring to map data segments to compute nodes, it is compatible with consistent hashing, thus robust to node changes and easy to adopt, as further shown in Sec. 4.4.

In summary, this paper makes the following contributions:

- We propose *range hashing* that for each data segment, identifies an appropriately sized group of nodes *with one single hash* that effectively balances the load on nodes while ensuring data locality.

• We propose the idea of the *virtual hash ring* that reduces the *collision probability* of queries that access different data segments, thus producing a balanced query distribution on nodes.
• With range hashing and the virtual hash ring, HotHash offers a strong theoretical guarantee on both data locality and load balance while being compatible with consistent hashing and robust to node changes. Moreover, we theoretically show that HotHash is superior to Bounded Load-based methods in data transmission costs.
• Our experiments with various workloads confirm that HotHash outperforms the state-of-the-art from 1.4× to 150× in average execution time and tail latency.

2 Background

This section overviews affinity scheduling in cloud databases (Sec. 2.1), consistent hashing for affinity scheduling as well as its problem in handling skewed workloads (Sec. 2.2). We then describe the key objectives of this work in Sec. 2.3

2.1 Affinity Scheduling In Cloud Databases

Cloud-oriented databases such as Presto [35], F1 Query [34], Aurora [42, 43], Snowflake [11, 44] and SparkSQL [4], adopt a storage-disaggregation architecture. This design separates the computation and storage, bringing unique advantages such as better scalability and higher elasticity.

Affinity Scheduling. For a disaggregated storage architecture, the network connecting the computation and data storage often constitutes the performance bottleneck. If a query is assigned to a node where the required data segments are missing, the node will have to fetch these data segments from the distributed storage through a network. This often leads to performance degradation compared with a shared-nothing database [40].

Caching addresses this problem where compute nodes cache a data segment locally and reuse it on other queries rather than repeatedly fetching it through the network. To improve the cache hit rate, cloud databases use *consistent hashing* [21, 22] to distribute queries to compute nodes, ensuring that subsequent or concurrent queries accessing the same data segment will be assigned to the same node [11, 19, 34], i.e., *affinity scheduling*. This guarantees *data locality*, greatly improving the system's performance.

2.2 Consistent Hashing for Affinity Scheduling

Consistent hashing represents the resource requestors (e.g., queries) and the compute nodes in a ring structure known as the hash ring. The compute nodes and the requests can be placed at random locations on this ring using a *hash function*. Each request is served by the node that first appears in a clockwise traversal of the ring. Intuitively, each node "owns" a range of the hash ring, and any requests coming in at this range will be served by the same node.

As shown in Fig. 2, there are three nodes on the ring. $Node_1$ owns range [0,100] and any request falling into this range will be sent to it. For example, given a query q requesting data segment A, q is hashed to location 50 based on its ID A. Then it is sent to and cached on $node_1$. Subsequent queries that process data segment A will be assigned to $node_1$ and thus are able to reuse the cached data segment A.

Consistent hashing is popular in cloud databases for affinity scheduling because it is able to maintain the data-node mapping at a minimum cost when facing node additions or removals, which occurs commonly in the cloud. When a node is added or removed, only the queries that fall into the range this node owns will have to be re-assigned. As shown in Fig. 2, removing $node_1$ will only affect data segment A of query q which now should be owned by $node_2$.

Issues in Handling Skewed Workloads. Despite its robustness to node changes, consistent hashing causes load imbalance when handling skewed workloads. More specifically, consistent hashing distributes queries based on the data segments they access. However, in reality, the query

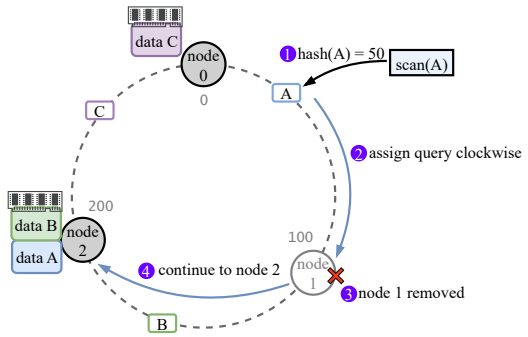


Fig. 2. Consistent Hashing: Robust to Change

workloads are typically very skewed, where some data segments (hot data) are accessed much more frequently than others. This can lead to load imbalance and large query latencies on hot nodes.

For example, in Fig. 2, if data segment A is much hotter than B and C, then an overwhelming portion of queries will be assigned to $node_1$, leaving only few queries processed on $node_0$ and $node_2$. Eventually, this will overload $node_1$.

2.3 HotHash Objectives

In cloud databases, query latency mainly constitutes I/O time and computation time. Load imbalance caused by skewed query workloads is one of the main causes of stragglers that slows down query execution [10]. Targeting minimizing query latency under skewed query workloads, we design a new hashing-based query scheduling mechanism, called *HotHash*, which meets three objectives: (1) *balancing the load* across compute nodes to reduce computation time; (2) *preserving data locality* to optimize I/O time; (3) *being compatible with consistent hashing* to retain the key properties of consistent hashing, i.e., systems like Snowflake can simply replace consistent hashing with HotHash to distribute micro-partitions to nodes, while preserve the simplicity and robustness to node changes.

3 HotHash Overview

In this section, we introduce how HotHash schedules queries in a cloud DB under analytical workloads and show how to integrate it into a cloud database. We then introduce some key design considerations of HotHash.

3.1 Query Scheduling with HotHash

This work assumes that the cloud DB has already *partitioned* tables into data segments using any partitioning method such as range partitioning or hash partitioning. Each data segment is stored as a file in open file formats. Data segment is the basic caching unit which corresponds to the micro-partition in Snowflake [11], data block in Redshift [16] and Capacitor in Google F1 Query [2, 34]. For example, as shown in Fig. 3 step 1, the query planner determines that there are four queries accessing segment A and two queries accessing segment B. It then uses HotHash to distribute data segments to compute nodes.

More specifically, HotHash uses **range hashing** to map a data segment to a group of nodes that *fall into a certain range of the hash ring*. This range is proportional to the hotness of this data segment. As shown in the example in Fig. 3 step 2, HotHash constantly tracks data hotness information and determines that data segment A is hotter than B. It thus HotHash maps A to a

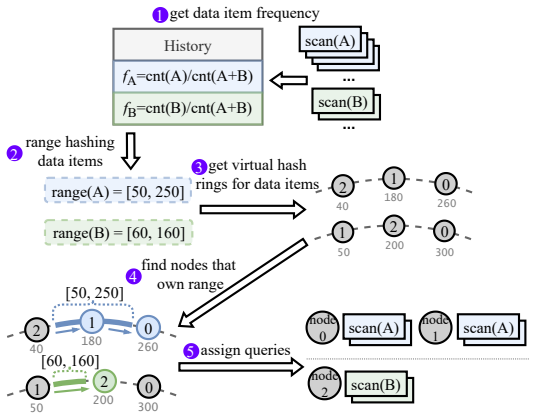


Fig. 3. HotHash

longer range than B on the hash ring and assign A to two nodes and B to one node. Since multiple nodes handles A, the load is *naturally balance* on nodes. Because the same data segments still go to a fixed set of nodes, range hashing *preserves data locality*, thus reducing data transmission cost.

However, range hashing increases the collision probability among queries and eventually leads to load imbalance. This is because collisions is more likely to occur due to the overlap range of different data segments. In this case, two hot data segments might be assigned to the same nodes. For example, as shown in Fig. 3 step 3, using a single hash ring of $[node_2, node_1, node_0]$ for both A and B will result in data segments A assigned to $node_0$ and $node_1$, and B also assigned to $node_1$, leaving $node_2$ unused and $node_1$ still overloaded as in consistent hashing.

HotHash thus introduce **virtual hash ring** to address this issue. That is, each data segment sees a specific (virtual) hash ring corresponding to a random permutation of nodes in the cloud database. Now, the same range covers different nodes in different rings. Therefore, two data segments with overlapping range do not necessarily go to the same nodes. As shown in Fig. 3 step 3-4, data segment A sees the virtual hash ring of $[node_2, node_1, node_0]$, while data segment B sees another virtual hash ring of $[node_1, node_2, node_0]$. As a result, A is assigned to $node_0$ and $node_1$, while B is assigned to and $node_2$, avoiding overloading $node_1$ and overloading $node_1$.

Compatible to Consistent Hashing. We illustrate that HotHash is compatible to consistent hash in three perspectives. First, similar to consistent hashing, HotHash (1) uses one hash operation to map a data segment to a group of nodes, (2) maps compute nodes to different virtual hash rings with a hash function that takes the key of a data segment as an argument, and (3) finds a node from the node group to host one particular query with hashing. See Sec. 4 for details. Second, in Sec. 4.4 we show that HotHash maintains the data-node mapping at a minimum cost when facing node changes, like consistent hashing. Furthermore, we show in Sec. 4.3 that HotHash does not introduce additional data movement under node changes, although it assigns one node to multiple hash rings. However, compared to consistent hashing, HotHash requires additional work to monitor data hotness for range hashing, as further discussed in Sec. 4.2.

3.2 Design Considerations

We further discuss some designs of HotHash in supporting real-world cloud databases.

Avoid Single Point Failure. HotHash supports multiple schedulers to avoid single point failure. Specifically, HotHash levarages the consensus protocols well studied in literature such as gossip [23] and Raft [29] to efficiently synchronize the data hotness information between multiple schedulers. For example, as commonly used in distributed databases [18, 38, 39], each scheduler could run a Raft-based protocol to constantly get data hotness information maintained by the leader scheduler to reach consensus. In the case of leader failure, the rest schedulers could perform leader selection to quickly reach consensus again. Since HotHash continuously updates data frequencies with a sliding window, the temporal inconsistency could be promptly mitigated.

Queries Accessing Multiple Segments. For ease of presentation, in this paper, we use queries that access only one data segment as examples. However, HotHash naturally supports queries accessing multiple data segments. Such queries can be divided into two categories: *queries with or without joins*. If a query does not involve join, HotHash hashes the data segments it accesses separately and runs it parallelly on different machines.

When handling joins, a cloud database could use HotHash to map each raw data segment (micro-partition) involved in the join to compute nodes. HotHash offers similar data locality benefit like consistent hashing. Once the scan and filter operations on each data segment involved in a join are completed, the cloud database then shuffles the intermediate results to perform the join with a chosen join algorithm. If co-partitioned join such as hash join is selected, it can use HotHash to co-locate segments by using the combined data segment ID as the hash key. For example, if $query_1$ accesses data segments A and B, HotHash co-locates A and B on the same node using the hash key AB. This allows HotHash to support hash join or merge sort join.

Data Changes. This work focuses on caching raw inputs for scan operators in analytic workloads where objects (data segments) are immutable, similar to other works [11, 44, 45]. However, HotHash readily supports data deletion, insertion, and update. More specifically, when a cloud database inserts new data records or deletes/updates existing data records, the system could use a Log-structured Merge Tree (LSM)-like structure to efficiently generate new immutable segments. This is a typical mechanism to handle data change in cloud databases [6, 12, 25]. In addition, this mechanism may cause the deletion of existing data segments when compaction occurs. Handling new or deleted segments in HotHash is straightforward. When a query accesses a new data segment, HotHash simply hashes it to compute nodes, as described in Sec. 4. After a data segment is deleted, no queries will access this segment anymore. Its corresponding cache on compute nodes will naturally be purged by a cache eviction mechanism such as LRU.

4 The HotHash Techniques

In this section, we first introduce the key steps in HotHash. We then present the details of *range hashing* and *virtual hash ring*. Finally, we show that HotHash incurs the same cost with consistent hashing when handling node changes.

4.1 The Overall Process of HotHash

Next, we describe the details of HotHash, which mainly consists of 3 steps. Given a query q_i and data segment d_i it accesses, HotHash (1) computes a range of the hash ring based on the hotness of d_i; (2) generates a hash ring for d_i; and (3) distributes query q_i among the nodes that fall into the corresponding range of this hash ring.

As shown in Alg. 1, given a query q_i accessing data segment d_i, HotHash starts with conducting *range hashing*: it computes the hash value $loc = hash(d_i)$ and the data frequency f_{d_i} (Lines 2-3), and then computes $range(d_i)$ using loc and f_{d_i} (Line 4).

Next, HotHash uses the virtual hash ring w.r.t. d_i to decide the node group. If the virtual hash ring has not been generated before, HotHash will generate a new permutation of nodes using a

Algorithm 1: HotHash

```
   Input: Queries, Nodes, timestamp
   Output: Assignment
1  for q_i ∈ Queries do
2      loc = hash(d_i)
3      f_{d_i} = frequency(d_i)
4      range(d_i) = [loc, loc + f_{d_i} * len_R]
5      vring = virtualRing(d_i)
6      if ¬vring then
7          vring = permute(hash(d_i), Nodes)
8      nodes = ∅
9      for node ∈ vring do
10         if range(node) ∩ range(d_i) then
11             nodes.add(node)
12     node = assign(q_i, nodes)
13     Assignment[q_i] = node
14     update(d_i, timestamp)
15 return Assignment
```

hash function that has $hash(d_i)$ as seed and store it for future use (Lines 5-7). In this way, even if the node mapping of each hash ring is lost, HotHash can restore it with the hash function.

HotHash then produces the node group *nodes* by finding any node where the range it owns on the corresponding virtual hash ring intersects with $range(d_i)$ (Lines 8-11). It assigns query q_i to one node within the node group and updates the frequencies of the historical queries accordingly (Lines 12-14). Specifically, the node assignment function hashes q_i (with its ID as key) to another hash ring built over the node group *nodes* and uses the rule of consistent hashing (next node on the ring) to find a node to serve q_i.

HotHash uses hash functions and hash rings to identify a node group, map nodes to rings, and choose a node to host a query, ensuring that it is *compatible* with consistent hashing.

Next, in Sec. 4.2 and Sec. 4.3, we present in detail range hashing and virtual hash ring, and use an example to further illustrate HotHash's query scheduling process sketched in Alg. 1.

4.2 Range Hashing

Range hashing is based on the hotness of the data. Therefore, we start with designing a lightweight mechanism to measure this metric. We then show how to use data hotness to compute the range that a data segment occupies on a ring.

Sliding Window-based Data Hotness Evaluation. HotHash continuously updates data hotness based on batches of queries that fall within a sliding window. For query batches arriving in future windows, we first detect concept drift by measuring their correlation with the existing window and update the data hotness if the correlation becomes small [36]. Specifically, we use the Pearson correlation to measure the similarity between two windows and use a threshold of 0.5 in experiments to determine whether the hotness distribution of data segments has changed. For example, given segment IDs [A, B], if their frequencies change from [10, 20] to [100, 200], the correlation will be high, indicating the distribution has little change. However, if their frequencies change from [10, 20] to [200, 100], the Pearson correlation will be low. We set the threshold to be 0.5 to allow some fluctuations in the access patterns while still capturing the major changes. In this way, HotHash keeps the hotness statistics stable to avoid unnecessary data transmission costs caused by frequent

scheduling changes when the characteristics of the query workload fluctuate only slightly, while still ensuring adaptation to new hotness patterns when significant drifts occur. In the future, we will investigate how to dynamically tune this threshold with query latency as feedback.

Within the sliding window, HotHash increments the count of each data segment it accesses. The frequency of a data segment is computed as the count of the data segment over the total counts of all data segments accessed by queries in a window. Formally, we define data frequency as follows:

Definition 4.1. We use $[m]$ to denote historical queries and $[D]$ for the data segments used by the queries. A query q_i is denoted as $q_i = (i, d_i)$ where $i \in [m]$ is the query ID and d_i is the data segment used by this query. The frequency of data $d_i \in [D]$ is computed as $f_{d_i} = \frac{1}{m} \sum_{j=1}^{m} [d_j = d_i]$

For the example shown in Fig 3, the frequency of data segment A f_A is computed as the count of accessing A (4) over the total count (6) of accessing data segments A and B: $f_A = 4/6 \approx 0.67$.

Computing Data Hotness with Other Metrics. Although in the above presentation we use the query access frequency as the metric to measure hotness, any other metrics can be equally plugged into HotHash. For example, we could use the same sliding window mechanism to collect the execution time of historical queries per data segment and compute data hotness based on these statistics. Let $t(q_i)$ be the historical execution time of q_i. The data hotness of a data segment d_i is then computed as $h_{d_i} = \sum_{j=1}^{m} t(q_{d_j=d_i}) / \sum_{k=1}^{m} t(q_k)$. HotHash is thus able to effectively handle scenarios where: (1) different queries accessing the same data segment may incur different computational costs and (2) data segments are in different sizes.

Computing the Range. Next, HotHash computes a *range* of the hash ring that a data segment d_i could fall into. Given a query $q_i = (i, d_i)$, HotHash first uses consistent hashing to hash data d_i to a location on the ring denoted as $loc(d_i) = hash(d_i)$, which is the start point of the range. Then, it computes the length of the range as $len(d_i) = len_R \times f_{d_i}$, where len_R denotes the total length of the whole ring. Formally, the range of d_i is computed as follows:

$$range(d_i) = [hash(d_i), hash(d_i) + len_R \times f_{d_i}] \tag{1}$$

In Fig. 3, the length of the whole hash ring is 300 which bounds the possible hash value to [0, 300). Segments A and B are hashed to locations 50 and 60, respectively. Because we have computed $f_A = 2/3$ and $f_B = 1/3$ in step 1, $range(A) = [50, 250]$ and $range(B) = [60, 160]$.

4.3 Virtual Hash Ring

For each new data segment d_i that a query workload accesses, HotHash generates a random permutation (virtual hash ring) of the nodes and stores it for later usage. Specifically, we modify the hash function to take a seed as an extra argument, where the seed is the hash value of this data segment. We then use this new function to map nodes to a virtual hash ring. The location of a node $node_j$ on a virtual hash ring w.r.t. data segment d_i is computed as $loc(node_j) = PermuteHash(node_j, seed = hash(d_i))$. Hashing with different seeds maps a node to different locations on different virtual hash rings, equivalent to randomly permuting the nodes on each virtual hash ring. In Fig. 3 step 3, the virtual hash rings are $[node_0, node_1, node_2]$ for A and $[node_2, node_1, node_0]$ for B.

In summary, given any query q_i that accesses data segment d_i, the group of nodes to which q_i could be distributed is determined by $range(d_i)$ computed by Eq. 1 and the virtual hash ring w.r.t. d_i. In Fig. 3 step 4, the node group of A is $[node_0, node_1]$, while the node group for B is $[node_2]$.

Assigning a Node From the Node Group. For each query q_i, HotHash assigns one node from its node group to serve the query. As depicted in Fig. 3 step 5, $scan(A)$ will be distributed to $node_0$ or $node_1$, while $scan(B)$ always goes to $node_2$.

Virtual Hash Ring: The Theory. We analyze how virtual hash ring addresses the increased collision probability due to range hashing. Since the n physical nodes are randomly distributed on

each virtual ring, a virtual node on a ring could correspond to any physical node. No matter to which virtual node a query q_i accessing data segment d_i is mapped, it has a collision probability $\frac{1}{n}$ with another query q_j accessing data segment d_j. Therefore, statistically, HotHash has the same collision probability as consistent hashing when processing queries accessing different data segments. However, now two queries that access the same data segment d_i have a probability of $\frac{r-1}{r}$ going to two different nodes, where r denotes the number of nodes to which range hashing replicates d_i. This avoids overloading one node when d_i is hot.

Virtual Hash Ring: No Additional Data Movement. Although a node exists on multiple hash rings in HotHash, this does not introduce additional data movement under node churns. This is because each ring is only responsible for one particular data segment. Therefore, given a ring, a node will host only one data segment that falls into its range. In contrast, in consistent hashing, multiple data segments might fall into the range that a node owns. Because both consistent hashing and HotHash *hash* nodes and data segments to a ring, the random nature of hash functions ensure that the expected number of data segments assigned to a node is equivalent in consistent hashing and HotHash. Because data movement depends on the number of data segments assigned to each node, HotHash does not incur additional data movement when adding or removing nodes.

4.4 Handling Node Removals and Additions

HotHash is able to keep the data-node mapping at a minimum cost when facing node changes, same to consistent hashing.

The range hashing of HotHash maps each data segment to a range of the hash ring. This mapping is invariant to the number of nodes and their permutations. A node $node_i$ could host a query $q_i = (i, d_i)$ if the range that $node_i$ owns on the hash ring intersects with $range(d_i)$, where the range ownership of $node_i$ is determined in a way exactly the same to consistent hashing. Therefore, a node addition or removal only impacts data segments falling into the range this node owns.

Furthermore, once a node $node_i$, which could be a new node or existing node, takes over a data segment d_j from another node $node_j$ due to node removal or addition, $node_i$ is guaranteed to be in the node group of d_j if $node_j$ was in this node group, and vice versa. Therefore, HotHash is able to correctly schedule the queries accessing data segment d_j to $node_i$ and reuse the data cached on it.

For example, as shown in Fig. 3 step 4, if $node_2$ is removed, it does not impact queries accessing data segment A, because $node_2$ is not in the node group of A ($[node_0, node_1]$). On the other hand, the node group of B changes from $[node_2]$ to $[node_0]$, as now $node_0$ owns the range of $node_2$. Consequently, $node_0$ takes over data segment B, which HotHash is still able to find and reuse. On the other hand, suppose a new $node_3$ is inserted into a location between $node_1$ and $node_0$ on A's virtual hash ring and after $node_0$ on B's virtual hash ring. On A's virtual hash ring, because $node_3$ owns a part of the range that $node_1$ owned before, part of $node_1$'s queries (data segments) will move to $node_3$. As with $node_1$, now the new node $node_3$ becomes a member of A's node group ($[node_1, node_3, node_0]$). Therefore, HotHash will begin scheduling queries that access A to $node_3$. However, on B's virtual hash ring, although $node_3$ takes over a part of the range owned by $node_0$, $node_0$ is not in the node group of B and thus neither is $node_3$. Therefore, the queries that access B remain unchanged.

5 Theoretical Analysis

Next, we show that HotHash has strong theoretical guarantees on both load balance and data locality. We first introduce the additional notation used in the analysis in Sec. 5.1. We establish the bounds on load imbalance and data locality in Sec. 5.2.

5.1 Notation

To describe the load on nodes, we introduce the *query assignment* and *node load* formally as follows:

Definition 5.1. We use a to denote the query assignment. Each assignment $a_i \in [m]$ represents the node ID to which a query $q_i = (i, d_i)$ is mapped. The load of a node $k \in [n]$ is computed as $\sum_{i=1}^{m}(a_i = k)$ and we denote this by w_k.

We measure load imbalance as the *variance* of load on nodes.

Definition 5.2. Imbalance $\mathcal{I}$: First, consider the variance of a workload: $\sigma^2 = \frac{1}{n}\sum_{k=1}^{n}(\frac{m}{n} - w_k)^2$. Assume all nodes are symmetrical and thus all w_ks follow the same distribution $\mathcal{D}$. Therefore, the load on each node corresponds to a random variable $w \sim \mathcal{D}$. This gives $\sigma(w) = \left|\frac{m}{n} - w\right|$, simplifying the definition of imbalance as:

$$\mathcal{I} = \frac{n}{m}\mathbb{E}_a\left[\sigma(w)\right] = \mathbb{E}_a\left[\left|\frac{n}{m}w - 1\right|\right] \tag{2}$$

In Eq. 2, w is the random variable representing the load on each node, which is determined by the assignment a, and E_a denotes the expectation over all such assignment a.

Next, we define *cache hit rate* and *data transmission cost* to measure the data locality of HotHash.

Definition 5.3. Cache hit rate C is a measure of the proportion of data segments that are assigned to the same node where they were previously assigned. It is computed as follows:

$$C = \frac{1}{m}\sum_{i=1}^{m}\left[\sum_{j=1}^{i-1}[d_j = d_i][a_j = a_i] > 0\right] \in [0, 1]$$

Here a_i represents the node to which data segment d_i is assigned. An assignment a_i is considered a hit if a previous assignment a_j assigns the same data segment d_i to the same node. Intuitively, C represents the probability that a query could reuse the cached data.

Definition 5.4. Data transmission cost $\mathcal{T}$ represents the number of data segments fetched from remote storage. It is calculated as:

$$\mathcal{T} = m(1 - C)$$

Intuitively, given a query, HotHash will trigger a data transmission if and only if the query misses the cache. This explains why we compute the transmission cost in this way.

5.2 Bound on Workload Imbalance and Cache Hit rate

In this section, we first establish HotHash's upper bound on load imbalance in Theorem 5.1. We then show a lower bound on its cache hit rate in Theorem 5.2.

We introduce a parameter α (≥ 1) into Eq. 1 to trade-off load balancing and data locality. That is, $range(d) = [hash(d), hash(d) + len_R \times f_d^{\alpha}]$. Given a data segment d, a larger α will lead to a smaller $range(d)$. HotHash thus will replicate d to a smaller number of nodes n_d ($n_d \propto nf_d^{\alpha}$), yielding a better data locality but potentially a worse load balance.

Theorem 5.1. The load imbalance $\mathcal{I}$ of HotHash is at most:

$$\mathcal{I} \leq O(D^{\alpha-2}) \tag{3}$$

Here D represents the number of data segments, which is typically a large value. Therefore, if we set α smaller than 2 (1 by default), the load imbalance $\mathcal{I}$ of HotHash will be very small.

Proof. Queries assigned to a specific node follow an accumulation of binomial distributions:

$$w \sim \mathcal{D} = \sum_{d=1}^{D} B(1, f_d^{\alpha})B\left(mf_d, \frac{1}{nf_d^{\alpha}}\right)$$

By normal approximation:

$$B\left(mf_d, \frac{1}{nf_d^\alpha}\right) \sim \mathcal{N}\left(\frac{m}{n}f_d^{1-\alpha}, \frac{m}{n}f_d^{1-\alpha}\left(1-\frac{1}{nf_d^\alpha}\right)\right)$$

Therefore, we have:

$$w \sim \frac{m}{n}\sum_{d=1}^{D} f_d^{1-\alpha} B(1, f_d^\alpha)\mathcal{N}\left(1, \frac{n}{m}f_d^{\alpha-1}\left(1-\frac{1}{nf_d^\alpha}\right)\right)$$

$$\frac{n}{m}w - 1 \sim \sum_{d=1}^{D} f_d^{1-\alpha}\left(B(1, f_d^\alpha)\mathcal{N}\left(1, \frac{n}{m}f_d^{\alpha-1}\left(1-\frac{1}{nf_d^\alpha}\right)\right) - f_d^\alpha\right)$$

For convenience, we denote $X \sim B(1, f_d^\alpha)$ and $Y \sim \mathcal{N}\big(1, \frac{n}{m}f_d^{\alpha-1}\big(1-\frac{1}{nf_d^\alpha}\big)\big)$. We could rewrite the above as:

$$\frac{n}{m}w - 1 \sim \sum_{d=1}^{D} f_d^{1-\alpha}\left(XY - f_d^\alpha\right)$$

Notice that the binomial distribution and the normal distribution are independent. For X and Y, $\mathbb{E}[X] = f_d^\alpha$ and $\mathbb{E}[Y] = 1$, thus:

$$\mathbb{E}_a\left[\frac{n}{m}w - 1\right] = \sum_{d=1}^{D} f_d^{1-\alpha}\left(\mathbb{E}[X]\mathbb{E}[Y] - f_d^\alpha\right) = 0$$

This shows that the load on each node is balanced in expectation. Now, consider $\mathbb{E}_a[|\frac{n}{m}w - 1|]$, by the Chebyshev's inequality:

$$\Pr\left(\left|\frac{n}{m}w - 1\right| \geq t\right) \leq \frac{1}{t^2}\mathrm{Var}\left[\left|\frac{n}{m}w - 1\right|\right]$$

This gives:

$$\mathbb{E}_a\left[\frac{n}{m}w - 1\right] = \int \Pr\left(\left|\frac{n}{m}w - 1\right| \geq t\right)\mathrm{d}t \leq C \cdot \mathrm{Var}\left[\left|\frac{n}{m}w - 1\right|\right]$$

We have shown that $E[XY - f_d^\alpha] = 0$, therefore:

$$\mathrm{Var}[XY - f_d^\alpha] = \mathrm{Var}[X]\mathrm{Var}[Y] + \mathbb{E}^2[Y]\mathrm{Var}[X] + \mathbb{E}^2[X]\mathrm{Var}[Y]$$

and

$$\begin{aligned}
&\mathrm{Var}\left[\left|\frac{n}{m}w - 1\right|\right] \\
=&\sum_{d=1}^{D} f_d^{2(1-\alpha)}\mathrm{Var}[XY - f_d^\alpha] \\
=&\sum_{d=1}^{D} f_d(1-f_d^\alpha)\frac{n}{m}\left(1-\frac{1}{nf_d^\alpha}\right) \\
&+ f_d^{2-\alpha}(1-f_d^\alpha) + \frac{n}{m}f_d^{1-\alpha}\left(1-\frac{1}{nf_d^\alpha}\right)
\end{aligned}$$

By the symmetry and the Lagrangian method, in the worst case, we have $f_d = \frac{1}{D}$ for $d \in [D]$. In this case:

$$\begin{aligned}
&\mathrm{Var}\left[\left|\frac{n}{m}w - 1\right|\right] \\
=&\frac{(D^\alpha - 1)(n - D^\alpha)}{mD^{\alpha+1}} + \frac{mD^{\alpha-1}(D^\alpha - 1)}{mD^{\alpha+1}} + \frac{D^{2\alpha}(n - D^\alpha)}{mD^{\alpha+1}} \\
=&\frac{nD^\alpha - n - D^{2\alpha} + D^\alpha + mD^{2\alpha-1} - mD^{\alpha-1} + nD^{2\alpha} - D^{3\alpha}}{mD^{\alpha+1}} \\
=&O\left(D^{\alpha-2}\right)
\end{aligned} \tag{4}$$

□

Theorem 5.2. The cache hit rate C of HotHash is at least:

$$C \geq 1 - \frac{n}{m} O\left(\frac{D}{n} + 1\right) \tag{5}$$

Accordingly, the data transmission cost $\mathcal{T}$ is bounded by $O(D + n)$:

$$\mathcal{T} \leq O(D + n) \tag{6}$$

Proof. To show that Eq. 5 holds, we first establish Eq. 7.

$$C \geq \frac{1}{m}\sum_{d=1}^{D}\left(mf_d - \underbrace{\lceil nf_d^\alpha \rceil}_{\leq nf_d^\alpha + 1}\right) \geq 1 - \frac{n}{m} O\left(\frac{D}{n} + \sum_{d=1}^{D} f_d^\alpha\right) \tag{7}$$

From the derivation in Def. 5.3, we have:

$$C = \frac{1}{m}\sum_{i=1}^{m}\left[\sum_{j=1}^{i-1}[d_j = d_i][a_j = a_i] > 0\right]$$

To simplify it, we group the queries first by data items, and then by node:

$$\begin{aligned}
C &= \frac{1}{m}\sum_{i=1}^{m}\left[\sum_{j=1}^{i-1}[d_j = d_i][a_j = a_i] > 0\right] \\
&= \frac{1}{m}\sum_{d=1}^{D}\sum_{k=1}^{n}\sum_{i=1}^{m}[d_i = d][a_i = k]\left[\sum_{j=1}^{i-1}[d_j = d][a_j = k] > 0\right] \\
&= \frac{1}{m}\sum_{d=1}^{D}\underbrace{\sum_{k=1}^{n}\max\left(\left(\sum_{i=1}^{m}[d_i = d][a_i = k]\right) - 1, 0\right)}_{\text{since } \sum\max(x-1,0)=\sum(x-1)+\sum[x=0]} \\
&= \frac{1}{m}\sum_{d=1}^{D}\left(mf_d - n + \sum_{k=1}^{n}\left[\sum_{i=1}^{m}[d_i = d][a_i = k] = 0\right]\right) \\
&= \frac{1}{m}\sum_{d=1}^{D}\left(mf_d - \sum_{k=1}^{n}\left[\sum_{i=1}^{m}[d_i = d][a_i = k] > 0\right]\right) \\
&= \frac{1}{m}\sum_{d=1}^{D}\left(mf_d - n\mathbb{E}_k\left[\exists i, d_i = d, a_i = k | d\right]\right)
\end{aligned}$$

By Eq. 1, given a data item d, the number of nodes in its node group is $\lceil nf_d^\alpha \rceil$, which means $\mathbb{E}_k\left[\exists i, d_i = d, a_i = k | d\right] \leq \Pr_k(k \text{ is in group of } d) = \frac{1}{n}\lceil nf_d^\alpha \rceil$. This shows that Eq. 7 holds.

Note that $\sum_{d=1}^{D} f_d = 1$. Because $\alpha \geq 1$, this gives $\sum_{d=1}^{D} f_d^\alpha \leq 1$. Therefore, we have $C \geq 1 - \frac{n}{m}O\left(\frac{D}{n} + 1\right)$. This shows that Eq. 5 holds. Accordingly, in the **worst case**, the transmission cost is $\mathcal{T} \leq O\left(D + n\right)$. Thus Eq. 6 holds. □

D denotes the number of data segments; m represents the number of queries in the workload; and n is the number of nodes. By Eq. 5, the cache hit rate C relies on $\frac{D}{m}$. Because m is typically much larger than D, C tends to be close to 1, indicating very high cache hit rate. Regarding the data transmission cost $\mathcal{T}$ (Eq. 6), we will show in Sec. 5.3 that HotHash is guaranteed to be better than the baseline.

5.3 HotHash Is Better In Theory

To show the theoretical advantage of HotHash, we first introduce a baseline, *BalancedHash*, which enhances *Bounded Load* [7, 27] with our randomness idea introduced in virtual hash ring. We then show that HotHash is superior to BalancedHash in both data transmission cost and cache hit rate.

BalancedHash: a Bounded Load-based Method. Given a query q that accesses a data segment A, BalancedHash uses consistent hashing to map it to a node $node_i$. If the workload of $node_i$ is lower than a load bound B, $node_i$ will host this query, fetching the data segment A from the (remote) storage, caching A here, and then processing the query q. Otherwise, q will be routed to another node. As discussed in Sec. 4, routing the query to the next node in the hash ring along the clockwise direction will increases *collision probability* [7] of queries that access different data segments, hence more likely overloading the nodes.

BalancedHash avoids this problem by adopting one of our key ideas, namely introducing *randomness* [7] into query routing. Rather than routing the queries linearly along the hash ring, it randomly chooses the next hop for each query. More specifically, when routing a query, BalancedHash modifies the IDs (keys) of its data segments and rehashes the query with consistent hashing to a different location on the hash ring until finding a non-overloaded node.

To trade off load balance and data locality, we introduce an overload factor ϵ into the definition of workload bound.

Definition 5.5. Given a system with n nodes, the workload bound $B = (1 + \epsilon)\frac{1}{n}L_c$, where L_c denotes the current workload of the whole system and ϵ is an input parameter that has to be larger than 0.

By Def. 5.5, the larger ϵ is, the more the system allows overloading a node. When ϵ is 0, the system expects perfect load balance.

The Advantage of HotHash Over BalancedHash. First, we establish for BalancedHash an upper bound of its worst-case cache hit rate. Consider a scenario where $D = \frac{n}{1+\varepsilon} - 1$, where the number of queries accessing each data segment, computed as m/D, is slightly higher than the load bound of each node, computed as $m/(\frac{n}{1+\epsilon})$. Consequently, we observe at least $D = \frac{n}{1+\varepsilon} - 1$ query overflows.

Then BalancedHash needs to redistribute these overflowed queries to non-overloaded nodes, which is a fraction $p = 1 - \frac{1}{1+\epsilon}$ of all the nodes. According to the geometric distribution, the expected number of nodes that a query q has to traverse before finding a non-overloaded node is $L = 1/(1-p) = 1 + \frac{1}{\epsilon}$. Although this redistribution process only creates 1 data replication w.r.t. query q, query q potentially touches L nodes, with node $node_1$ storing data segment d_1 of query q_1, and so on.

We then construct a new workload by re-ordering the queries in the above workload. This new workload moves query q before q_1, causing q_1 to overflow instead of q. Consequently, q_1 triggers another round of redistribution and creates a new data replication. The redistribution of q_1 again touches a list of nodes, with $node'_2$ storing q'_2 on data d'_2, etc. This process iterates until the query

lands in a non-overloaded node. This in expectation takes L turns, resulting in a workload with L data replications instead of 1.

Since in the initial workload we created $\frac{n}{1+\varepsilon} - 1$ re-distributed queries, to construct a worst-case query ordering, we adjust the ordering of each of these queries in the way described above. Then, the total number of data replications it causes is $\left(\frac{n}{1+\varepsilon} - 1\right) \times L = \left(\frac{n}{1+\varepsilon} - 1\right)\left(1 + \frac{1}{\varepsilon}\right)$. Therefore, the worst-case data transmission cost for BalancedHash corresponds to $\mathcal{T}_{BH} = \Omega\left(D + \left(\frac{n}{1+\varepsilon} - 1\right)\left(1 + \frac{1}{\varepsilon}\right)\right) = \Omega\left(D + n/\varepsilon\right)$. To ensure load balance, ε has to be small (< 1) [7], leading to $1/\varepsilon > 1$. Therefore, $\mathcal{T}_{BH} = \Omega(D + n/\varepsilon) > O(D + n) = \mathcal{T}_{HotHash}$, proving that HotHash has a lower data transmission cost, consequently a higher cache hit rate.

6 Experiments

In this section, we evaluate the performance of HotHash by focusing on the following questions:

- How does HotHash perform under various query workloads that have different sizes of datasets, different sizes of data segments, different skewness factors, different number of queries?
- Is HotHash robust to dynamic workloads where data keep changing, hot data drifts over time, and compute nodes change during query execution?
- How large is the overhead of HotHash including tracking data hotness and virtual hash ring?
- Ablation study: How do the range hashing, virtual hash ring, and the parameter α affect HotHash?

6.1 Experiment Setup

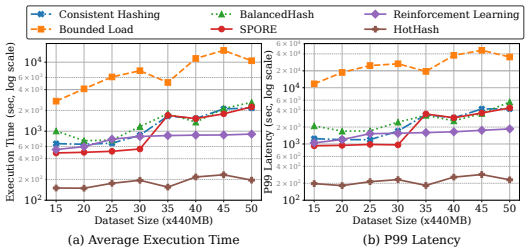
Experiment Environment. We implement a prototype system to evaluate HotHash and baselines in a cloud environment. It consists of a coordinator and a set of worker nodes. The coordinator distributes analytical SQL queries to workers using different methods evaluated in this work. We run experiments on the Google Cloud Platform (GCP) to evaluate HotHash in real cloud environment.

Data and Query Workloads. As discussed in Sec. 3.2, HotHash partitions tables into blocks and caches data at the *data segment* level. Accordingly, we develop a data generator to generate data *block by block*. Each data block by default is 440MB in its original format according to the typical micro-partition size of Snowflake [11]. We also use smaller data block sizes of 10MB, similar to FlexpushdownDB [45], to evaluate HotHash's scalability to a large number of blocks. We control the size of the datasets by varying the number of data blocks. We generate queries according to Yahoo! Cloud Serving Benchmark (YCSB) [9]. 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 [15] 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 that is close to 1 means low skewness [1]. Because all methods show similar trends on the two benchmarks, due to the space limit, we only report the YCSB results except the experiments of varying query skewness.

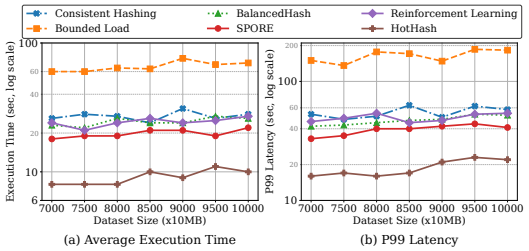
We also generate more complex workloads to evaluate HotHash under a more dynamic and realistic workload including: (1) data segments of different sizes, (2) hot data drifts over time, (3) workloads with data changes, and (4) node removal/addition during query execution.

Moreover, we conduct experiments with Redset [41], which is a real-world analytical workload collected from the Amazon Redshift cloud data warehouse. It is complex and skewed, containing queries that access different numbers (2-15) of data segments and involve multiple types of analytical operations.



(a) Average Execution Time (b) P99 Latency

Fig. 4. Varying Dataset Sizes (440MB per Data Block)



(a) Average Execution Time (b) P99 Latency

Fig. 5. Varying Dataset Sizes (10MB per Data Block, Redset)

Hardware Resources. We run experiments on the GCP with 20 `e2-highmem-4` compute nodes, similar to the X-Large warehouse of Snowflake [11]. Each node has 4 vCPUs and 32GB memory. The data blocks are stored in a standard GCP storage bucket, and each vCPU has around 1.2Gbps network bandwidth. To avoid overflowing compute engines, each node uses 4GB memory to cache data.

Baselines. We compare the following baselines:

- *Consistent Hashing*: The original consistent hashing [21] that assigns a query based on the data it requires.
- *Bounded Load*: Assign a query based on the data it requests and the workload of the node [27]. Distribute queries to the next node if a node is considered overloaded.
- *BalancedHash*: Assign a query based on the data it requests and the workload of the node. Distribute queries via re-hashing to avoid cascade overflow [7] (Sec. 5.3).
- *SPORE*: Assign a query based on the data it requests and replicate a hot data with a fixed data frequency threshold and replication factor [17] (Sec. 7).
- *Reinforcement Learning (RL)-based scheduling*: A learned scheduler designed to optimize load balance and data locality.

Configurations. By default, we use a dataset with 15 data blocks. To evaluate *HotHash*'s performance on datasets with more data blocks, we use two additional datasets with 10,000 blocks, but each block is smaller. The default skewness parameter θ is set to 1.3. This results in approximately 80% of queries that request 20% of data (hot data), similar to the skewness level of the Redset [41] workload. For *Bounded Load* and *BalancedHash*, we use a default balancing parameter $\epsilon = 0.3$ according to [7, 27]. This means allowing the most loaded node to have at most 1.3× of queries than the average. For *SPORE*, we use the experiment configurations recommended in [17] 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. By [17], *SPORE* is not sensitive to the hotness threshold and replication factor. For *HotHash*, we set the default $\alpha = 1$, i.e., a scheduling

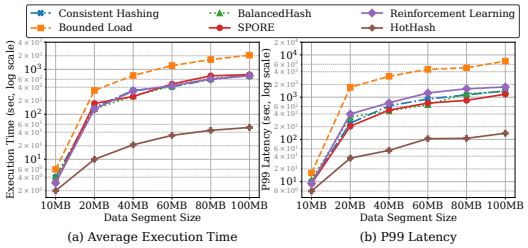


Fig. 6. Varying Data Segment Sizes

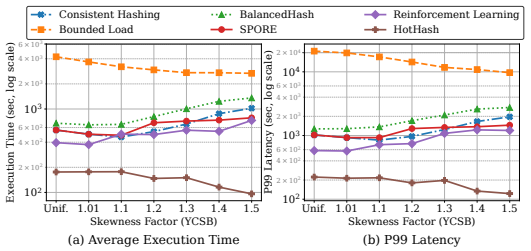


Fig. 7. Varying Workload Skewness with YCSB

strategy that strictly enforces load balance. When computing the hotness of data segments, we set the sliding window to continuously track a batch of 500 queries.

The *RL-based* scheduler is implemented using a Deep Q-Network (DQN), with both the state and multi-objective reward function designed to optimize load balance and data locality. Due to the excessive cost of training on a real cloud environment, we have designed a simulation environment to generate real-time cache and load states. This allows us to repeatedly retrain the scheduler in different experimental setups.

Metrics. Every 30 seconds we roughly submit 500 queries to the system and run 20k queries in total. We then report the average query execution time and the 99th percentile latency (tail latency). We also report the cache hit rate and load imbalance factor, as well as the overhead that HotHash introduces.

6.2 Evaluation With Varying Workloads

In this section, we evaluate the performance of *HotHash* by varying the data scales and the properties of query workloads to show the effectiveness of *HotHash* under different scales of workload.

6.2.1 Varying dataset size

In this set of experiments, we vary the number of data blocks, but keep the skewness factor of the workload fixed, targeting both "many segment per nodes" and "few very hot segments relative to nodes" regimes.

Fig. 4 shows the average query execution time and the tail latency under the default dataset configuration. We also report the results on the real-world Redset workload using much more (10k) but smaller (10M) data blocks in Fig. 5.

HotHash consistently outperforms all baselines by a large margin, from 3× up to 150×. For the default dataset configuration (Fig. 4), when the dataset has fewer than 30 data blocks, *HotHash* outperforms *SPORE* by 3×, *Consistent Hashing*, *BalancedHash*, and *RL-based scheduler* by 5×, and

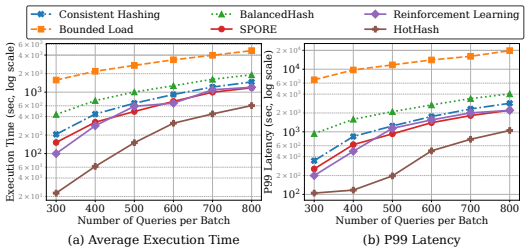


Fig. 8. Varying the Number of Queries in Query Workload

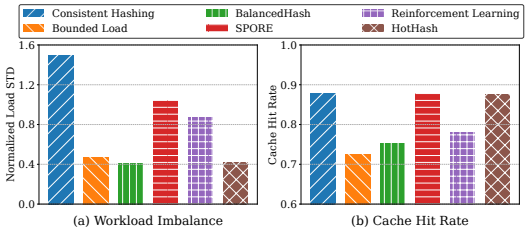


Fig. 9. Load Imbalance & Cache Hit Rate

Bounded Load by 25× in average execution time. The tail latency shows a similar trend in Fig. 4 (b). When the dataset contains more than 35 data blocks, *HotHash* remains the advantage against the *RL-based scheduler* and wins more against other baselines: it is 10× faster than *SPORE*, *Consistent Hashing*, and *BalancedHash*, and is 50× faster than *Bounded Load* in execution time; its advantage is even larger for tail latency, where it is 20× faster than *SPORE*, *Consistent Hashing*, and *BalancedHash*, and is 150× faster than *Bounded Load*.

For the real-world Redset workload (Fig. 5), similarly, *HotHash* is at least 3× faster than *RL-based* scheduler, *BalancedHash*, *BalancedHash* and *SPORE*, and 7× faster than *BoundedLoad* in both average execution time and tail latency.

In particular, *Bounded Load* performs the worst. Although *BalancedHash* performs better than *Bounded Load*, it does not show advantages over *Consistent Hashing* and *SPORE* for the following reasons. When re-distributing queries to non-overloaded nodes, *Bounded Load* increases the collision probability of queries, cascadingly overflowing a sequence of nodes. It thus tends to replicate both hot and cold data segments all over nodes. Although *BalancedHash* mitigates this problem by introducing randomness, it still suffers from spreading cold data segments, as analyzed in Sec. 5.3, leading to high data transmission cost. *SPORE* slightly outperforms *Consistent Hashing* but is much worse than *HotHash*. This is because *SPORE* replicates hot data to a fixed number of nodes, leading to insufficient load balance. *The RL-based scheduler* slightly outperforms *Consistent Hashing*, *BalancedHash*, and *SPORE*, but is slower than *HotHash*. This is because, while our simulation environment can provide real-time feedback on cache locality and load balance, there remains a gap between simulation and a real-world cloud environment, resulting in sub-optimal scheduling decisions.

HotHash's range hashing dynamically determines the size of the node group hosting hot data segments. This achieves load balance while avoiding unnecessarily data replication. This advantage becomes more apparent as the size of the datasets increases.

Cache Hit Rate & Load Imbalance: Near Optimal. As shown in Fig. 9, *HotHash* achieves a cache hit rate of 0.87 and a normalized load imbalance of 0.42. The cache hit rate is close to *Consistent Hashing* (0.88), and the normalized load imbalance is close to *BalancedHash* (0.4), while *Consistent Hashing* and *BalancedHash* are expected to achieve the best possible cache hit rate and load imbalance, respectively. This shows that *HotHash* achieves a query scheduling policy close to the **optimal solution**. Therefore, it is difficult for the baselines to outperform *HotHash*, even if we have careful retrained the *RL-based scheduler* in each experiment setting.

6.2.2 Varying Data Segment Sizes

We evaluate how *HotHash* performs when processing queries accessing multiple data segments using variable-sized data segments. We vary the size of each data segment from 10MB to 100MB. The results are shown in Fig. 6. We observe that the size variance impacts all baselines, including *HotHash*, since the data segment sizes directly determine the data transmission costs and affects the query cost. The near optimal trade-off of data locality and load balance makes *HotHash* consistently outperforms the baselines by a large margin, achieving improvements ranging from 40% to 60×.

6.2.3 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 show the results on the YCSB workloads. *HotHash* outperforms other methods from 3.5× up to 100×, even if the workload is close to uniform. When the level of skewness gets larger ($\theta >$ 1.2), the execution time and the tail latency of *Bounded Load* and *HotHash* decrease. This is because a largely skewed query workload is overwhelmed by hot queries requesting a small portion of data. Therefore, for *Bounded Load*, query rescheduling and data replication are bounded to a small set of nodes, leading to less data transmission. *HotHash* benefits from a more skewed workload, because its range hashing bounds data replication to certain nodes.

For *Consistent Hashing*, as the level of skewness increases, the workload imbalance becomes more severe on nodes, and therefore, its execution time and tail latency increase. *SPORE* does not outperform *Consistent Hashing* on less-skewed workload, because the extra data transmission cost in this case outweighs the performance degradation caused by workload imbalance. A similar trend occurs for *BalancedHash* since it generates a more balanced query distribution by rehashing queries to different places. When the level of skewness increases, queries jump through more hops from their home nodes. This creates more data replications.

6.2.4 Varying The Number of Queries

We vary the size of the query workloads by tuning the number of queries in each batch with a fixed skewness factor. We use the default setup otherwise.

Our *HotHash* significantly outperforms all baselines. As shown in Fig. 8 (a) and Fig. 8 (b), *HotHash* is about 7× faster than *SPORE*, and *RL-based scheduler*, 10× faster than *Consistent Hashing* and *BalancedHash*, and 70× faster than *Bounded Load* in average execution time. In terms of tail latency, *HotHash* outperforms *SPORE*, and *RL-based scheduler* by 2.5×, *Consistent Hashing* and *BalancedHash* by 8×, and *Bounded Load* by 60×.

6.3 Evaluation With Dynamic Workloads

We evaluate *HotHash* under dynamic workloads where the data segments are updated, the data access patterns change rapidly, and compute nodes join or leave the systems.

6.3.1 Data Changes

In this experiment, we evaluate how HotHash performs under data changes. We update the data in the dataset and vary the percentage of data change from 0% to 30%. The results reported in

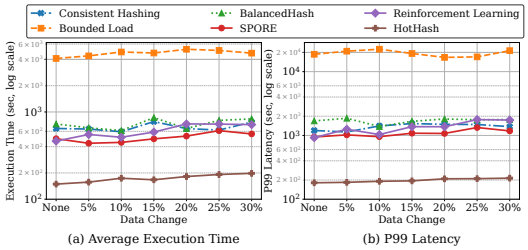


Fig. 10. Varying Data Change Percentage

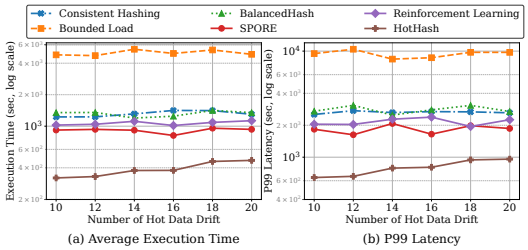


Fig. 11. HotHash under Rapid Concept Drifts

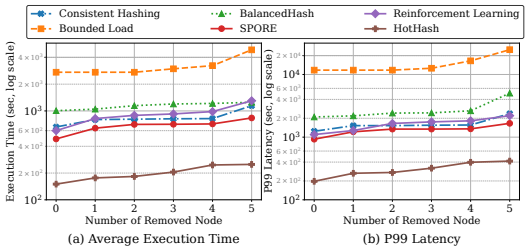


Fig. 12. Removing Compute Nodes

Fig. 10 show a similar trend as in previous experiments, outperforming all baselines from 5× to 50× in average execution time and tail latency. This is because rehashing the newly updated data segments do not introduce much overhead.

6.3.2 *Concept Drift*

We evaluate how *HotHash* adapts to workload shifts by varying the frequency of hot data drifts. We compare HotHash with baselines under a workload with rapid hotness changes from every 2 minutes to every 30 seconds. As shown in Fig. 11, HotHash is still 2× faster than *Rl-based* scheduler, *SPORE*, *BalancedHash* and *Consistent Hashing*, and 10× faster than *BoundedLoad*. This shows that HotHash is able to mitigate the impact of hotness changes with the sliding window and Pearson correlation. Thus, HotHash remains significantly faster than baselines.

6.3.3 *Node Changes*

We investigate how *HotHash* responds to node changes by adding and removing nodes during query execution. As shown in Fig. 12 and Fig. 13, *HotHash* consistently outperforms all baselines by a large margin. In particular, *HotHash* is 5× faster than *Consistent Hashing*, *BalancedHash*, *SPORE*,

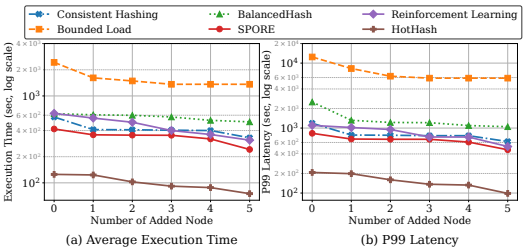


Fig. 13. Adding Compute Nodes

and the *RL-based scheduler*, and is 10× faster than *Bounded Load* in average execution time and tail latency. These results demonstrate that *HotHash* maintains the data-node mapping at a minimum cost under node changes just like *Consistent Hashing*, and remains its advantage against all baselines under node changes.

6.4 HotHash Overhead

In this section, we measure HotHash's overhead: (1) tracking data hotness and (2) storing virtual hash rings.

In our experiments, we set each window to contain 500 queries. The cost of calculating data hotness is around 300ms per batch. When amortized over each query within the batch, this cost is negligible compared to the average query execution time and latency per query. Specifically, the storage size of virtual hash ring is only 38KB, which is less than 0.002% of the dataset size in the query workload, while the total computation cost to maintain virtual hash ring is approximately 0.5s, accounting for less than 0.03% of the overall query execution time. The computational cost to add or remove a node from the virtual hash ring is around 250 μs, which is less than 0.001% of the average query execution time. In addition, we also measure the cost of redistributing data segments caused by node churn. In *HotHash*, it corresponds to 7% of the total cost of data transmission, which is very close to *Consistent Hashing* (6%).

We run additional experiments with 10^8 data segments, each 10MB. This results in 1PB data. Our results show that the memory overhead of storing rings in this setting is only around 6.8MB, while the data frequency statistics take around 310KB. Moreover, we also investigate opportunities to further reduce memory consumption. Specifically, we run experiments that only storing the rings of hot data segments whose frequency is above 0.1, while computing those of cold segments on the fly. The results show that the memory overhead is further reduced to around 1.5MB, while the total cost of computing rings on the fly is only 1.2s. Both are negligible compared to the total size of the dataset and the total query execution time.

6.5 Ablation Study

We study the effectiveness of range hashing and virtual hash ring.

6.5.1 *Range Hashing*

We evaluate range hashing by constraining the maximal length of the range allowed for each range hashing operation, i.e., the maximal replication factor, corresponding to the len_R parameter in Def. 1, Sec. 4. 0% indicates that a data segment is assigned to a single node, while 100% corresponds to our original range hashing that allows replicating data to all nodes. The results (Fig. 14) show that if HotHash assigns each data segment to one node, it performs similarly to *Consistent Hashing*. As the length increases, HotHash performs better due to the more balanced load on nodes. With

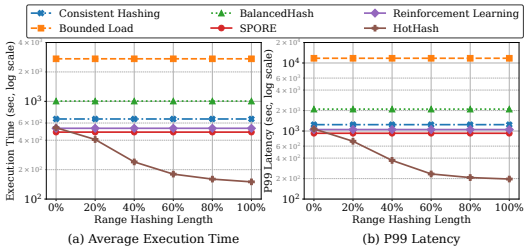


Fig. 14. Varying the Virtual Hash Ring Length

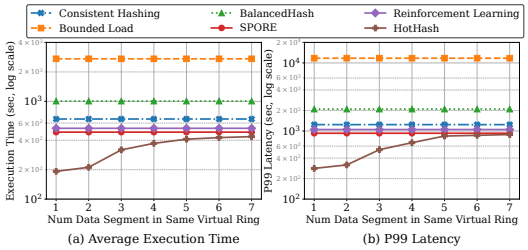


Fig. 15. Varying the Randomness of Virtual Hash Ring

our original range hashing, *HotHash* is 5× faster than *RL-based* scheduler, *SPORE*, *BalancedHash* and *Consistent Hashing*, and 10× faster than *Bounded Load*.

6.5.2 *Virtual Hash Ring*

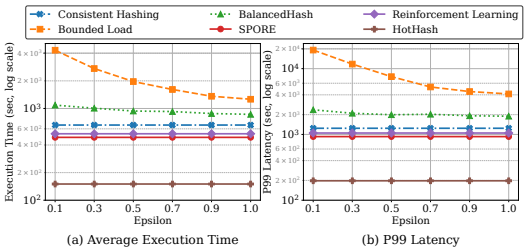
We investigate the trade-off between the benefit and the overhead of the randomness that virtual hash ring introduces. We control the randomness by varying the number of data segments that share the same hash ring. *HotHash* represents the extreme case with maximum randomness, where each data segment has its own hash ring. The other extreme is to let all data segments share a single hash ring, as in *Consistent Hashing*. As shown in Fig. 15, at the maximum randomness where each data segment has its own virtual hash ring, *HotHash* demonstrates the same performance gains as shown previously. As randomness decreases, the performance of *HotHash* also degrades. When 5 or more data segments share the same virtual hash ring, *HotHash*'s performance is close to *SPORE*. This is because if segments share a ring, HotHash tends to replicate hot segments to neighboring nodes, resulting in a high collision probability and hence a less balanced load. This is similar to *SPORE*, which replicates hot segments on the same hash ring, although using a different method than HotHash. The experiment shows that the randomness of virtual hash ring helps greatly.

6.6 Varying Parameters

6.6.1 *Varying Epsilon*

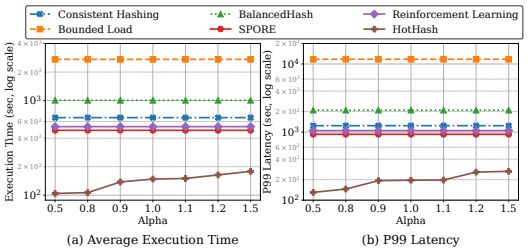
We evaluate how the parameter ε affects the performance of *Bounded Load* and *BalancedHash*. Note *Consistent Hashing*, *SPORE*, and *HotHash* do not use this parameter.

As shown in Fig. 16, *HotHash* consistently outperforms *Bounded Load* and *BalancedHash* as ε varies. When ε is close to 0.1, *HotHash* outperforms *BalancedHash* by 8× in average execution time and 10× in tail latency. In addition, *HotHash* is about 20× faster in execution time and 100× faster in tail latency compared to *Bounded Load*.



(a) Average Execution Time (b) P99 Latency

Fig. 16. Varying Epsilon



(a) Average Execution Time (b) P99 Latency

Fig. 17. Varying Alpha

When ε increases, the performance of *Bounded Load* and *BalancedHash* improves, although both methods are still slower than *HotHash* by at least 5×. This is because a large ε trades off load balance with cache hit rate, while a better cache hit rate reduces data transmission cost.

6.6.2 *Varying Alpha*

We evaluate how the parameter α affects *HotHash*. We vary the value of α and compare HotHash with other four baselines, which are not affected by α. As shown in Fig. 17, *HotHash* consistently outperforms *Consistent Hashing*, *SPORE* and *BalancedHash* by 10× and *Bounded Load* by 50×, indicating that α is not hard to tune.

7 Related Work

Consistent Hashing. In addition to cloud databases [11, 12], consistent hashing is widely adopted in multiple areas, including web caching, distributed storage systems, and peer-to-peer networks. Memcached [13] and Memcached-based systems [8, 28, 47] use consistent hashing to map hash keys to cache servers. Many distributed key-value storage systems, including Apache Cassandra [25] and Linked-in Voldemort [24] use consistent hashing to partition and distribute data items to storage servers. Some distributed hash table (DHT) systems and peer-to-peer networks are designed based on consistent hashing [5]. For example, Chord [37], Pastry [33], and CAN [31] use consistent hashing to locate data stored on specific nodes and provide efficient routing.

Many of these systems use consistent hashing with virtual nodes for load balancing. At a high level, each physical node (or server) is mapped to a set of virtual nodes on the hash ring, and data (cache or keys) mapped to each virtual node is stored on corresponding physical nodes. This reduces hot spots where some nodes store more data than others due to *imbalanced hash key distribution*. However, virtual nodes cannot balance skewed workload, since data with the same key is still mapped to a single node. Thus, hot data could still overload a node. In [7, 27], the authors improve consistent hashing to evenly distribute clients to servers using the Bounded Load idea discussed in

Sec. 1. However, it tends to spread data all over the nodes on the hash ring, incurring large data transmission costs.

Similar to [27], the idea of Bounded Load is also used in the content delivery networks (CDN) [26] where a hot object is replicated to multiple successor nodes. This increases the collision probability when mapping hot objects to nodes.

SPORE [17] is an augmented Memcached variant that replicates hot keys to multiple nodes to mitigate the impact of workload imbalance. In SPORE, each node maintains access counts of the data segments locally. When the access counts on a node exceed a fixed threshold, the server node replicates the hot key to a fixed number of nodes. This method, overlooking the relative popularities among different data segments, tends to be less effective in balancing load. In contrast, HotHash dynamically adjusts the replication rate of the data segments based on their hotness, thus consistently outperforming SPORE as shown in our experiments (Sec. 6).

EC-Cache [30] balances workload on cloud object storage through erasure coding. A data segment is encoded into multiple smaller data units stored on different storage servers. When a data segment is requested, the compute node retrieves a subset of data units randomly from multiple servers. However, although using EC-Cache to simultaneously access multiple storage servers improves read performance, it does not address the load imbalance issue on compute nodes due to skewed query workloads.

Snowflake [11] 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. However, reading additional data from remote storage inevitably increases query latency. Moreover, implementing file stealing requires additional engineering work whereas HotHash only modifies consistent hashing.

Straggler Mitigation. There have been many works focused on mitigating stragglers [14] in distributed systems. While there are many reasons that could cause stragglers, slow nodes due to overload are considered as one of the main sources of stragglers. A widely adopted solution in the Map-Reduce systems is LATE [46]. LATE performs speculative detection on each node to detect stragglers and launches copied tasks on faster nodes to accelerate the job. In addition, Hopper [32] uses speculative straggler detection to proactively reserve time slots on faster nodes to take over tasks on stragglers. Besides speculation, Dolly [3] generates multiple clones of tasks and executes them within their specified resource budget. The earliest finished tasks are used to improve latency. The key difference between our HotHash and these straggler mitigation techniques is that HotHash is designed for cloud databases, where caching data on compute nodes and reusing the cache later are vital to reduce query latency. Furthermore, unlike these works, HotHash does not require a heavy change to the systems.

8 Conclusion

We present HotHash, a query scheduling mechanism for cloud databases that offers a strong guarantee on both load balance and data locality under skewed workloads, while still preserving the robustness to node changes provided by consistent hashing. The key ideas include *range hashing* and *virtual hash ring* that together balance the workload, while at the same time avoiding unnecessary data transmission. Our evaluation on various workloads and hardware configurations shows that HotHash outperforms the state-of-the-art from 1.4× to 150× in execution time and tail latency.

Acknowledgments

This research was supported in part by NSF under grants DBI-2327954 and CCF-2106621 and by Amazon Research Award (2023 Fall).

References

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

[2] 2026. Google Capacitor. https://cloud.google.com/blog/products/bigquery/inside-capacitor-bigquerys-next-generation-columnar-storage-format.

[3] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 185–198.

[4] 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.

[5] Baruch Awerbuch and Christian Scheideler. 2006. Towards a scalable and robust DHT. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*. 318–327.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[7] John Chen, Benjamin Coleman, and Anshumali Shrivastava. 2021. Revisiting consistent hashing with bounded loads. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 3976–3983.

[8] 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.

[9] 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.

[10] Emilio Coppa and Irene Finocchi. 2015. On data skewness, stragglers, and MapReduce progress indicators. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 139–152.

[11] 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.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

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

[14] Sukhpal Singh Gill, Xue Ouyang, and Peter Garraghan. 2020. Tails in the cloud: a survey and taxonomy of straggler management within large-scale cloud data centres. *The Journal of Supercomputing* 76 (2020), 10050–10089.

[15] 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.

[16] 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.

[17] 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.

[18] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] David Kempe, Alin Dobra, and Johannes Gehrke. 2003. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 482–491.

[24] Jay Kreps. 2012. Project Voldemort.

[25] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.
[26] Bruce M Maggs and Ramesh K Sitaraman. 2015. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review* 45, 3 (2015), 52–66.
[27] 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.
[28] 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.
[29] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.
[30] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. {EC-Cache}:{Load-Balanced},{Low-Latency} Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 401–417.
[31] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. 161–172.
[32] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. 2015. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 379–392.
[33] Antony Rowstron and Peter Druschel. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*. Springer, 329–350.
[34] 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.
[35] 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.
[36] Sadia Shakil, Chin-Hui Lee, and Shella Dawn Keilholz. 2016. Evaluation of sliding window correlation performance for characterizing dynamic functional connectivity and brain states. *Neuroimage* 133 (2016), 111–128.
[37] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review* 31, 4 (2001), 149–160.
[38] Nishant Suneja. 2019. Scylladb optimizes database architecture to maximize hardware performance. *IEEE Software* 36, 04 (2019), 96–100.
[39] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.
[40] 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.
[41] 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.
[42] 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.
[43] 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.
[44] 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.
[45] 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).

[46] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments.. In *Osdi*, Vol. 8. 7.

[47] 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.

Received October 2025; revised January 2026; accepted February 2026