HotHash: Hotness-Aware Consistent Hashing for Cloud Databases

ABSTRACT

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.

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 same data segment to the same node, ensuring *data locality*. Consistent hashing [19, 20] is widely used for *affinity scheduling* in cloud databases [11, 18, 31, 38] 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, 24] 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 distributing.

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

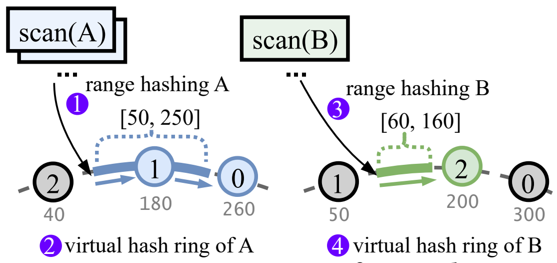


Figure 1: Overview of HotHash

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 [32], F1 Query [31], Aurora [36, 37], Snowflake [11, 38] 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 storage-disaggregation 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 [35].

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* [19, 20] 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, 17, 31], 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.

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,

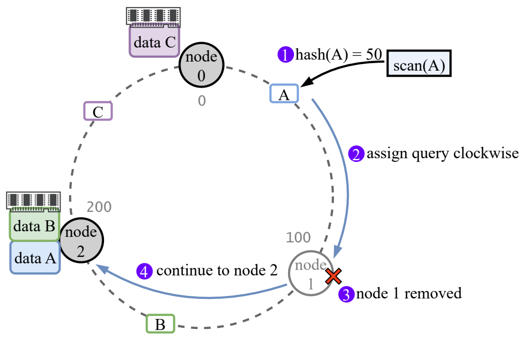


Figure 2: Consistent Hashing: Robust to Change

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 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., simplicity and robustness to node changes.

3 HOTHASH OVERVIEW

3.1 HotHash Key Ideas

HotHash achieves the above objectives with the following two ideas: (1) *range hashing* that considers the hotness of the data; (2) *virtual hashing ring* that introduce randomness into query distributing.

Range Hashing. Unlike consistent hashing that always hashes a data segment to one node, *HotHash*'s range hashing maps a data segment to a group of nodes that *fall into a certain range of the hash ring*, where the range is proportional to the hotness of this data segment. The hotter the data segment, the more replicas of the data segment will be created, while multiple nodes handling the same hot data segment naturally *balance the load* on nodes. Because the same data segments always go to a fixed set of nodes, range hashing still *preserves data locality*, thus reducing data transmission cost.

HotHash continuously computes and updates the hotness of data segments with a sliding window-based mechanism such that it can adapt to the drift of query workloads over time. HotHash is flexible in using different metrics to evaluate data hotness, e.g., query access frequencies or actual query execution time over data segments. In our experiments (Sec. 6.4), we show that maintaining data hotness incur negligible CPU and memory cost.

Virtual Hash Ring. One problem with range hashing, which maps a data segment to a continuous range of a ring, is that it increases collision probability among queries, while collisions result in load imbalance. This is because even if two data segments are hashed to different locations owned by two different nodes, collisions can still occur in range hashing due to the overlap of their arcs. In this case, two hot data segments might be assigned to the same nodes.

We thus introduce *virtual hash ring* to address this issue. Each data segment sees a specific (virtual) hash ring corresponding to a random permutation of nodes in the cloud database. Now, the same arc covers different nodes in different rings. Two data segments with overlapping arcs thus do not necessarily go to the same nodes.

More theoretically, because 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.

Combining range hashing and virtual hash ring, HotHash unifies the merits of *affinity scheduling* and *random scheduling*, two fundamental query scheduling strategies. Affinity scheduling achieves perfect data locality and hence minimal I/O costs, at the cost of load imbalance. Random scheduling, which distributes queries evenly to different nodes, ensures load balance with a price of *poor data locality*, suffering from high I/O costs. In HotHash, queries that access the same data segment always go to a fixed group of compute nodes, achieving the data locality merit of affinity scheduling, while HotHash randomly selects this group of nodes and randomly picks a node from the group to serve a query, concurrently obtaining the load balance strength of random scheduling. The details of range hashing and virtual hash ring are presented in Sec. 4.2 and 4.3.

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.

3.2 HotHash In Action in Cloud Databases

We introduce how HotHash schedule queries in a cloud DB and our two design decisions on handling data change and caching.

Query Scheduling. 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 as well.

The *query planner* in the database determines the data segments that each query accesses. Metadata such as the minimum and maximum values within each data segment could be used to filter out the segments that do not satisfy the query predicates. The cloud database then uses HotHash to hash the selected data segments to compute nodes with the ID of a data segment as the hash key, ensuring data locality and load balance.

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. For join queries, HotHash works in two ways, depending on the join algorithm selected by the query optimizer. First, if it selects a co-partitioned join, HotHash co-locates segments by treating them as a single unit and uses 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. The other option is to map each data segment involved in the join separately, similar to the case of a single-segment query. Once the scan and filter operations on each data segment involved in a join are completed, the system shuffles the intermediate results to perform join, i.e., co-locating the data that may produce join results. This supports join algorithms like hash join, merger sort join, broad cast join, etc.

Data Change. This work focuses on analytical query workloads where objects (data segments) are immutable. 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, 22]. In addition, this mechanism may cause the deletion of existing data segments when compaction occurs. Handling new or deleted segments in HotHash or consistent hashing 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.

Caching Raw Data. Same as other works in cloud databases [11, 38, 40], HotHash focuses on caching raw input tables and optimizing the performance of scan operators, as scan operators involve fetching data from remote storage through networks.

4 THE HOTHASH TECHNIQUES

In this section, we first introduce the overall process of 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, (1) HotHash first 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.

Algorithm 1: HotHash

```
Input: $Queries, Nodes, timestamp$
Output: $Assignment$
1  for $q_i \in Queries$ 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 $\neg vring$ then
7          $vring = permute(hash(d_i), Nodes)$
8      $nodes = \emptyset$
9      for $node \in vring$ do
10         if $range(node) \cap range(d_i)$ then
11             $nodes.add(node)$
12     $node = assign(q_i, nodes)$
13     $Assignment[q_i] = node$
14     $update(d_i, timestamp)$
15 return $Assignment$
```

As shown in Alg. 1, given a query q_i accessing data segment d_i, HotHash start 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 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).

Finally, HotHash 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 *assign* projects q_i 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.

The whole process 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 HotHash 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

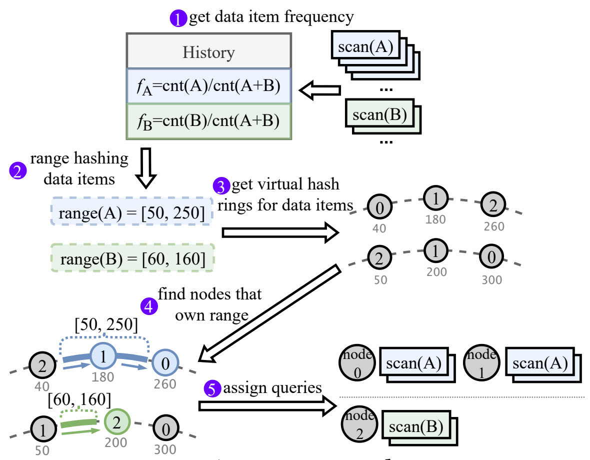


Figure 3: HotHash

correlation becomes small [33]. 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.

Within the sliding window, HotHash increments the count of each data segment it accesses. The frequency of a data segment is then computed as the count of the data segment over the total counts of all data segments accessed by queries in a window. More 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 $f_{d_i} = \frac{1}{m}\sum_{j=1}^{m}[d_j = d_i]$

Fig. 3 step 1 shows an example of frequency computation. The system has 3 nodes and sequentially gets 6 queries $\{q_1, \cdots, q_6\}$, accessing two data segments A and B. 4 queries access data segment A, while only 2 queries access data segment B. 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:

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

Fig. 3 step 2 shows an example of computing the range for data segments A and B. The length of the whole hash ring is 300 which bounds the possible hash value to [0, 300). Items A and B are hashed to locations 50 and 60 respectively ($hash(A) = 50$, $hash(B) = 60$). We have computed $f_A = 2/3$ and $f_B = 1/3$ in step 1. Therefore, $range(A) = [50, 250]$ and $range(B) = [60, 160]$.

4.3 Virtual Hash Ring

In order to more evenly distribute queries, HotHash proposes *virtual hash ring*. For each data segment, HotHash generates a random permutation of the nodes, which is generated once and stored for later usage. We store the node permutation in a data structure similar to the hash ring in consistent hashing and refer this data structure as a *virtual hash ring*. Specifically, for each data segment d_i, we modify the hash function to take a seed as an extra argument, where the seed corresponds to 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.

As shown in Fig. 3 step 3, data segments A and B have different virtual hash rings: it is $[node_0, node_1, node_2]$ for A and is $[node_2, node_1, node_0]$ for B. The queries that access the same data segment will always see the same virtual hash ring. 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. That is, any node can host q_i if it owns a part of $range(d_i)$ in the virtual hash ring w.r.t. d_i. For example, as shown 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. Thus, a sequence of queries that access the same data segment will be distributed across the same group of nodes. As depicted in Fig. 3 step 5, $scan(A)$ will be randomly distributed to $node_0$ or $node_1$, while $scan(B)$ always goes to $node_2$.

Virtual Hash Ring: No Additional Data Movement. Although a node exists on multiple hash rings in HotHash, 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. Thus, HotHash does not introduce additional data movement under node churns, because data movement depends on the number of data segments assigned to each node.

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 will only impact queries (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

In order 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. Please refer to Appendix A for the proof. □

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

$$C \geq 1 - \frac{n}{m}O\left(\frac{D}{n} + 1\right) \tag{4}$$

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

$$\mathcal{T} \leq O(D + n) \tag{5}$$

Proof. To show that Eq. 4 holds, we first establish Eq. 6, which is proven in Appendix B.

$$C \geq \frac{1}{m}\sum_{d=1}^{D}\left(mf_d - \underbrace{\lceil nf_d^{\alpha}\rceil}_{\leq nf_d^{\alpha}+1}\right) \geq 1 - \frac{n}{m}O\left(\frac{D}{n} + \sum_{d=1}^{D}f_d^{\alpha}\right) \tag{6}$$

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. 4 holds. Accordingly, in the **worst case**, the transmission cost is $\mathcal{T} \leq O(D + n)$. Thus Eq. 5 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. 4, 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. 5), we will show in Sec. 5.3 that HotHash is guaranteed to be better than the baseline.
Segments of Different Sizes. The above analysis assumes that data segments have the same size. Next, we give bounds in the scenario where segments vary in size. Let $s_i < S$ denote the size of data segment i relative to a unit segment d_u, where d_u corresponds to the smallest segment. Using frequency scaling $f_i = s_i \cdot f_i^{\text{unit}}$, and based on Theorem 5.1 and Theorem 5.2, the load imbalance is bounded by $O(S^2 \cdot D^{\alpha-2})$, and the total data transmission is bounded by $O(D + n \cdot S^{\alpha})$. Please refer to Appendix C for the proof.

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, 24] 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 $m/(\frac{n}{1+\varepsilon})$. 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 $p = 1 - \frac{1}{1+\varepsilon}$ 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 $L = 1/(1-p) = 1 + \frac{1}{\varepsilon}$. 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 $(\frac{n}{1+\varepsilon} - 1) \times L = (\frac{n}{1+\varepsilon} - 1)(1 + \frac{1}{\varepsilon})$. Therefore, the worst-case data transmission cost for BalancedHash corresponds to $\mathcal{T}_{BH} = \Omega\left(D + (\frac{n}{1+\varepsilon} - 1)(1 + \frac{1}{\varepsilon})\right) = \Omega(D + n/\varepsilon)$. 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.

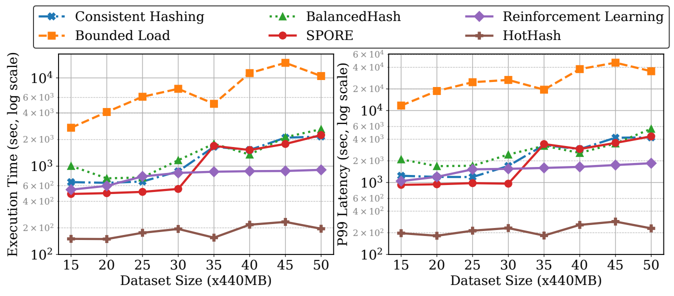
6 EXPERIMENTS

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

- How does *HotHash* perform compared to baselines under various query workloads that have different sizes of datasets, different sizes of data segments, different skewness factors, different number of queries, different analytical operations with varying complexity?
- 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 *HotHash* introduces?
- Ablation study: How do the randomness of virtual hash ring and the parameter α affect HotHash?
- How do hardware resources affect the performance of *HotHash*?

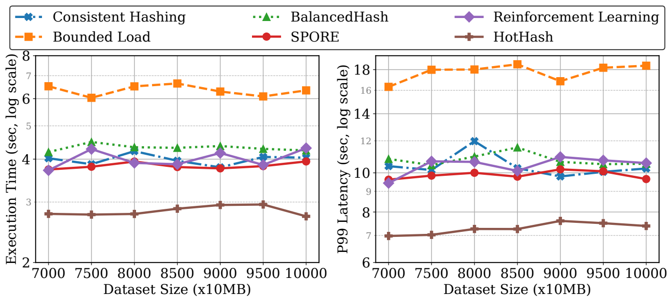
6.1 Experiment Setup

Experiment Environment. We implement a prototype system to evaluate HotHash and the 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**. Moreover, we conduct **simulation experiments** to show if HotHash is robust to *varying hardware configurations*. Due to space limit, we include the results of the simulation experiments in Appendix D.1. In summary, this set of experiments shows that *HotHash* achieves much better performance than all baselines with low-cost hardware, while readily benefiting from advanced hardware.
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. We also use smaller data block sizes of 10MB, similar to FlexpushdownDB [40], 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!



(a) Average Execution Time (b) P99 Latency

Figure 4: Varying Dataset Sizes (440MB per Data Block)



(a) Average Execution Time (b) P99 Latency

Figure 5: Varying Dataset Sizes (10MB per Data Block)

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. In addition to YCSB, we conduct experiments with Star Schema Benchmark (SSB) [26]. We introduce skewness to queries accessing the *lineorder* table, and queries are generated to favor data within certain dates.

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 SSB results on 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) workloads where queries use different analytic operations with varying complexity to access the same data segments, (3) hot data drifts over time, (4) workloads with data changes, and (5) node removal/addition during query execution.

Hardware Resources. We run our experiments on the GCP with 20 `e2-highmem-4` compute nodes. 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. In order 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 [19] 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 [24]. 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 [16] (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 700 and 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). For *Bounded Load* and *BalancedHash*, we use a default balancing parameter $\epsilon = 0.3$ according to [7, 24]. 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 [16] 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 [16], *SPORE* is not sensitive to the hotness threshold and replication factor. For *HotHash*, we set the default $\alpha = 1$. This corresponds to a scheduling 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 10 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

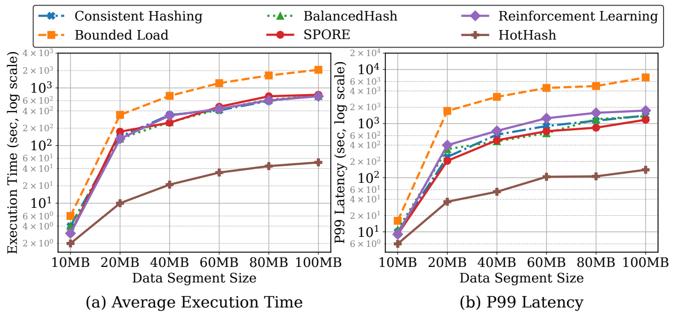
In this section, we evaluate the performance of *HotHash* by varying the data scales and the properties of query workloads.

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.

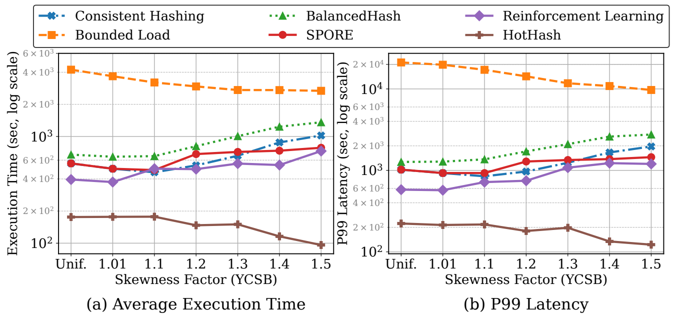
Fig. 4 shows the average query execution time and the tail latency under the default dataset configuration. In addition, we increase the total size of the dataset by using much more (10k) but much smaller (10M) data blocks and report the results in Fig. 5.

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 *Bounded Load* by 25× in average execution time. The tail latency shows a similar trend in Fig. 4 (b). *HotHash* constantly outperforms *SPORE* by 5×, *Consistent Hashing*, *BalancedHash* and *RL-based scheduler* by 8×, and *Bounded Load* by



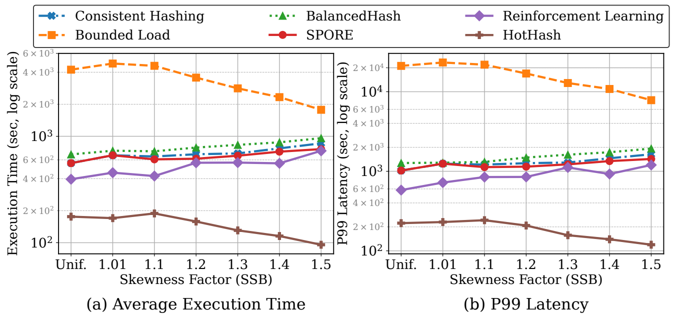
(a) Average Execution Time (b) P99 Latency

Figure 6: Varying Data Segment Sizes



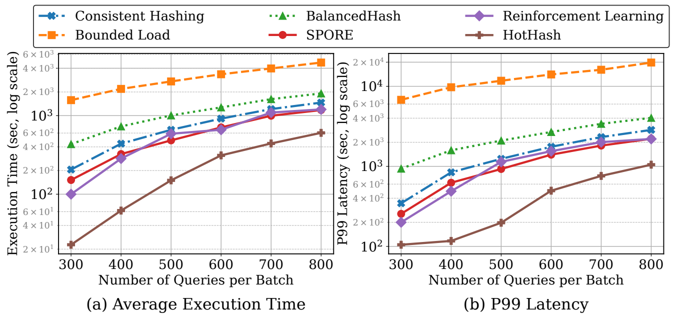
(a) Average Execution Time (b) P99 Latency

Figure 7: Varying Workload Skewness with YCSB



(a) Average Execution Time (b) P99 Latency

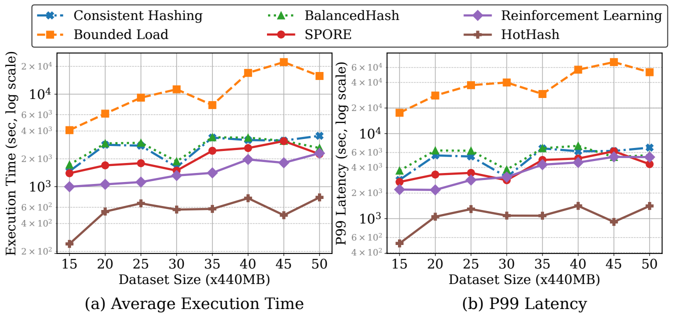
Figure 8: Varying Workload Skewness with SSB



(a) Average Execution Time (b) P99 Latency

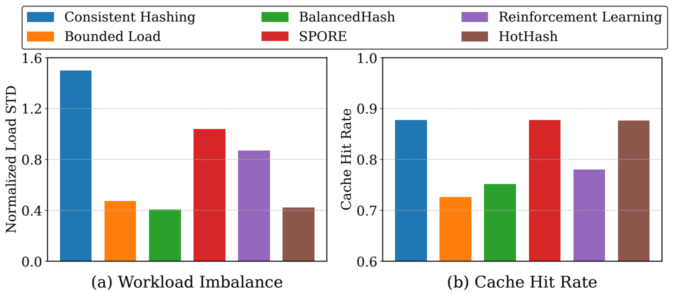
Figure 9: Varying the Number of Queries in Query Workload

100×. 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*.



(a) Average Execution Time (b) P99 Latency

Figure 10: Different Operators on the Same Data Segments



(a) Workload Imbalance (b) Cache Hit Rate

Figure 11: Load Imbalance & Cache Hit Rate

When the size of the data block is 10MB, Fig. 5 shows that *HotHash* is 40% faster than *SPORE*, *Consistent Hashing*, *BalancedHash*, and the *RL-based scheduler*, and 2.5× faster than *Bounded Load* 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 the cascade overflow problem by introducing randomness, it still suffers from spreading cold data segments, as analyzed in Sec. 5.3, leading to a data transmission cost much higher than *HotHash*. *SPORE* slightly outperforms *Consistent Hashing* but is still much worse than *HotHash*. This is because although *SPORE* distributes hot data to other nodes and thus balances the workload to some extent, it uses a fixed number of nodes to host hot data segments, leading to insufficient load balance. The RL-based scheduler slightly outperforms Consistent Hashing, BalancedHash, and SPORE, but it is still slower than HotHash in average execution time and tail latency. 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. 11, HotHash achieves a cache hit rate of 0.88 and a normalized load imbalance of 0.42. The cache hit rate is close to Consistent

Hashing (0.89), 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. Nevertheless, HotHash consistently outperforms them by a substantial 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 and Fig. 8 show the results on the YCSB and SSB workloads. All methods demonstrate a similar trend in execution time and tail latency. In all cases, *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* also benefits from a more skewed workload, because its range hashing technique 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. 9 (a) and Fig. 9 (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×.

6.2.5 *Varying The Complexity of Analytical Operators*

In this experiment, we run different analytical operators with different complexities on the same data segments. As shown in Fig. 10, *HotHash* still outperforms all baselines by a large margin. In particular, *HotHash* is 3× faster than *SPORE*, *RL-based scheduler*, *Consistent Hashing* and *BalancedHash*, and 20× faster than *Bounded Load* in average execution time. For tail latency, *HotHash* is also 3× faster than *SPORE*, *RL-based scheduler*, *Consistent Hashing* and *BalancedHash*, and 10× faster than *Bounded Load*.

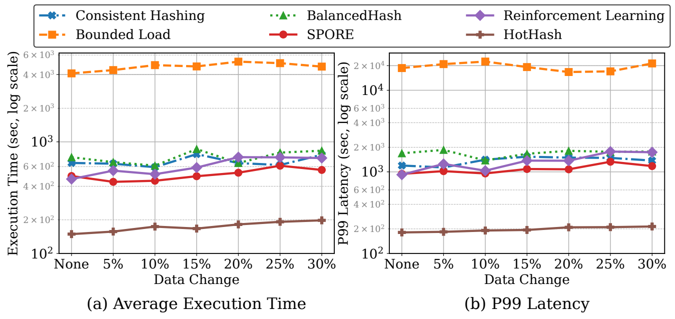


Figure 12: Varying Data Change Percentage

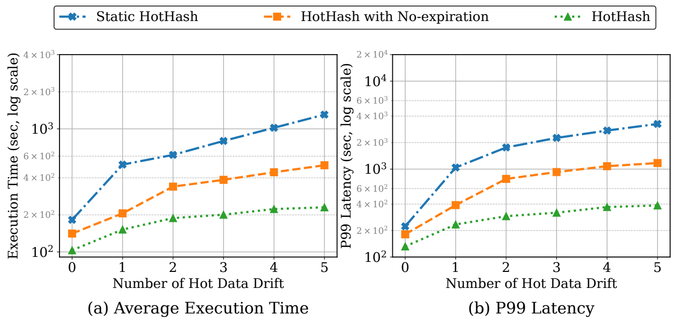


Figure 13: Workload with Concept Drifts

Summary. The experiment results confirm that *HotHash* consistently and significantly outperforms all baselines under different scales of datasets and various types of workloads.

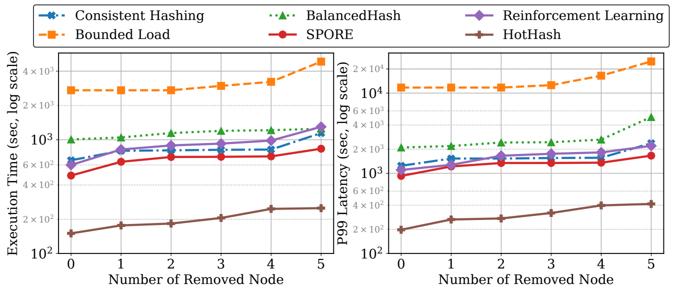
6.3 Evaluation With Dynamic Workloads

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. 12 show a similar trend as in previous experiments that HotHash consistently outperforms all baselines by a large margin, achieving improvements ranging from 5× to 50× in average execution time and tail latency. This is because data changes do not introduce much overhead to HotHash.

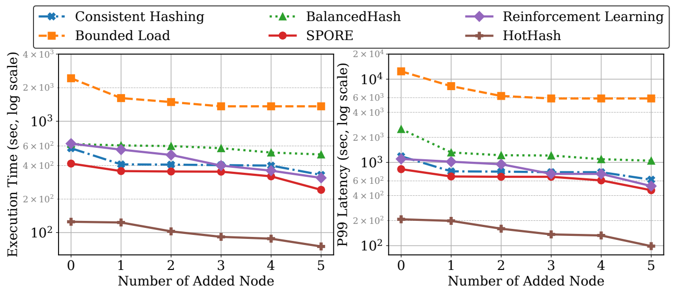
6.3.2 *Concept Drift*

We evaluate how *HotHash* adapts to workload shifts by varying the frequency of hot data drifts. We compare HotHash with two baseline variations: (1) calculating the hotness once based on queries within a time window and never updating the statistics (called *Static*); and (2) continuously updating the hotness statistics as new queries arrive but never expiring old queries (called *No-expiration*). As shown in Fig. 13, HotHash adapts well to real-time workload shifts and produces the best and most stable results. As the frequency of data drifts increases, HotHash outperforms *No-expiration* by 2× in average execution time and tail latency, and *Static* by 5×. *Static* performs the worst, as it does not adapt to data drifts. While



(a) Average Execution Time (b) P99 Latency

Figure 14: Removing Compute Nodes



(a) Average Execution Time (b) P99 Latency

Figure 15: Adding Compute Nodes

No-expiration can adapt to some degree, it does not precisely reflect long-term changes in data hotness, leading to performance degradation as the number of data drifts increases.

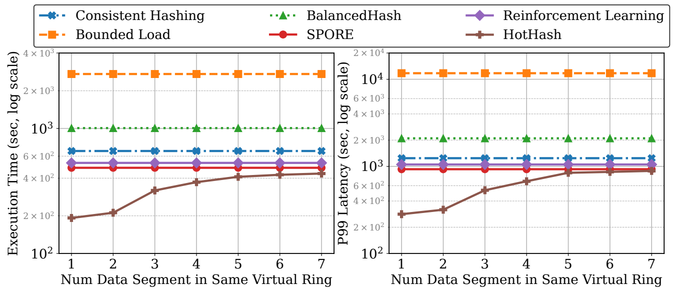
6.3.3 Node Changes

Junyong: We investigate how *HotHash* responses to node changes by adding and removing nodes in the system while the workload is being executed. As shown in Fig. 14 and Fig. 15, *HotHash* consistently outperforms all baselines by a large margin. In particular, HotHash is 5× faster than *Consistent Hashing*, *BalancedHash*, *SPORE*, and the *RL-based scheduler*, while also being 10× faster than *Bounded Load* in bothe average execution time and tail latency. These results demonstrate that HotHash is able to keep the data-node mapping at a minimum cost under node changes just like consistent hashing, and maintain its advantages against all baselines under node changes.

Junyong: **Summary.** The experiment results confirm that *HotHash* also provides strong scalability and performance in dynamic environments.

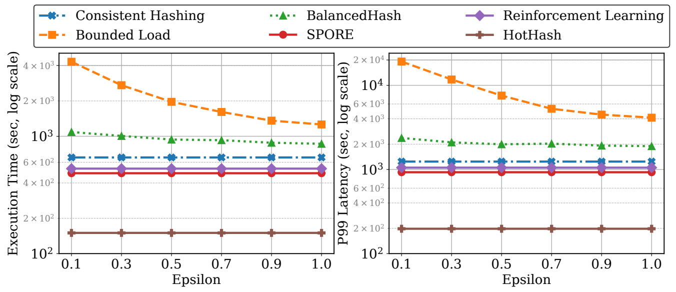
6.4 HotHash Overhead

We measure HotHash's overhead. Compared to *Consistent Hashing*, HotHash introduces two extra components: (1) data hotness tracking in the sliding window and (2) virtual hash ring. 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



(a) Average Execution Time (b) P99 Latency

Figure 16: Varying the Randomness of Virtual Hash Ring



(a) Average Execution Time (b) P99 Latency

Figure 17: Varying Epsilon

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%).

6.5 Varying Parameters

6.5.1 Varying the Randomness of 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.

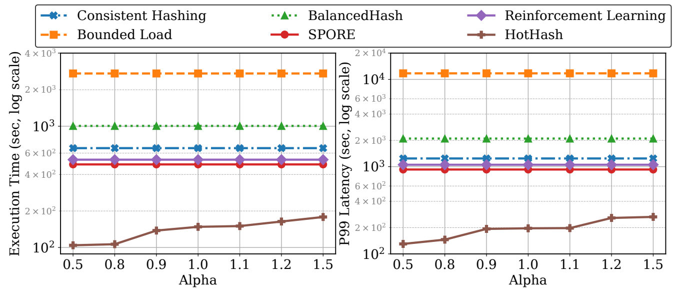
As shown in Fig. 16, at the maximum randomness where each data segment has its own virtual hash ring, *HotHash* demonstrates the same performance gains as shown previously. As randomness decreases, the performance of *HotHash* also degrades. When 5 or more data segments share the same virtual hash ring, *HotHash*'s performance is close to *SPORE*. This shows that the randomness of virtual hash ring helps greatly.

6.5.2 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. 17, *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*.

When ε increases, the performance of *Bounded Load* and *BalancedHash* improves, although both methods are still slower than



(a) Average Execution Time (b) P99 Latency

Figure 18: Varying Alpha

HotHash by at least 5×. This is because a large ε trade-off load balance with cache hit rate, while a better cache hit rate reduces data transmission cost.

6.5.3 *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. 18, *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, 25, 42] use consistent hashing to map hash keys to cache servers. Many distributed key-value storage systems, including Apache Cassandra [22] and Linked-in Voldemort [21] 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 [34], Pastry [30], and CAN [28] 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, 24], 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 [24], the idea of Bounded Load is also used in the content delivery networks (CDN) [23] where a hot object is replicated to multiple successor nodes. This increases the collision probability when mapping hot objects to nodes.

Slicer [2] is a sharding service that hashes service requests to a new key space. It balances server load by manipulating the server's key range and distributes hot requests to multiple servers. Similar to Bounded Load, Slicer introduces load bound to servers and always uses the least loaded server to avoid overloading a server. However, in cloud databases, it suffers the same problem as Bounded Load and BalancedHash due to overlooking the data transmission cost.

SPORE [16] 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 [27] 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 [41]. LATE performs speculative detection on each node to detect stragglers and launches copied tasks on faster nodes to accelerate the job. In addition, Hopper [29] 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. Yadwadkar et al. present Wrangler [39] that uses machine learning models to proactively avoid assigning tasks to slow nodes. 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 the *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.

REFERENCES

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

[2] 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.

[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] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Jay Kreps. 2012. Project Voldemort.

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

[23] Bruce M Maggs and Ramesh K Sitaraman. 2015. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review* 45, 3 (2015), 52–66.

[24] 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.

[25] 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.

[26] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). *Pat* 200, 0 (2007), 50.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. 2014. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14.

[40] 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).

[41] 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.

[42] 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.

A PROOF OF THE LOAD IMBALANCE BOUND

Theorem A.1. The load imbalance $\mathcal{I}$ of HotHash is bounded by $O(D^{\alpha-2})$.

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:

$$\begin{aligned}
&\mathrm{Var}\left[\left|\frac{n}{m}w - 1\right|\right] \\
=&\frac{(D^\alpha - 1)(n - D^\alpha)}{mD^{\alpha+1}} + \frac{mD^{\alpha-1}(D^\alpha - 1)}{mD^{\alpha+1}} + \frac{D^{2\alpha}(n - D^\alpha)}{mD^{\alpha+1}} \\
=&\frac{nD^\alpha - n - D^{2\alpha} + D^\alpha + mD^{2\alpha-1} - mD^{\alpha-1} + nD^{2\alpha} - D^{3\alpha}}{mD^{\alpha+1}} \\
=&O\left(D^{\alpha-2}\right)
\end{aligned} \tag{7}$$

□

B PROOF OF THE CACHE HIT RATIO BOUND

Theorem B.1. The cache hit ratio C of HotHash is bounded by $1 - \frac{n}{m}O\left(\frac{D}{n} + 1\right)$. Accordingly, the transmission cost is $\mathcal{T} = O(D + n)$.

Proof. From the derivation in Def. 5.3, we have:

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

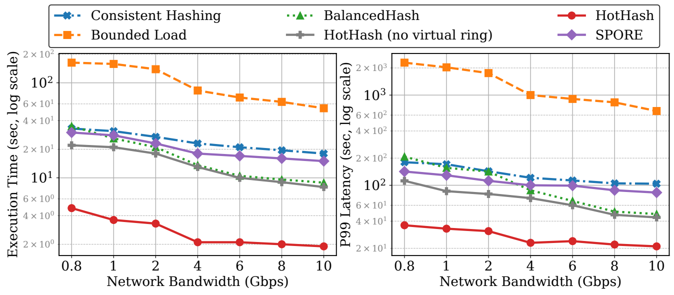
To simplify it, we group the queries first by data items, and then by node:

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

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

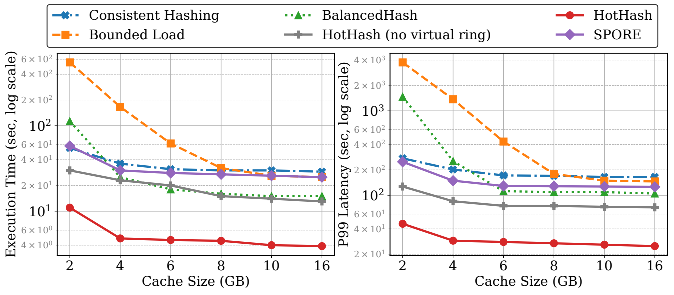
$$C \geq \frac{1}{m}\sum_{d=1}^{D}\left(mf_d - \underbrace{\lceil nf_d^\alpha\rceil}_{\leq nf_d^\alpha+1}\right) \geq 1 - \frac{n}{m}O\left(\frac{D}{n} + \sum_{d=1}^{D} f_d^\alpha\right) \tag{8}$$

Notice 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 $C \geq 1 - \frac{n}{m}O\left(\frac{D}{n} + 1\right)$. Accordingly, in the **worst case**, the transmission cost is $\mathcal{T} = O(D + n)$. □



(a) Average Execution Time (b) P99 Latency

Figure 19: Varying Network Bandwidth



(a) Average Execution Time (b) P99 Latency

Figure 20: Varying Cache Size

C DATA SEGMENT WITH DIFFERENT SIZES

Theorem C.1. Given a set of data segments with different sizes, in HotHash, the load imbalance is bounded by $O(S^2 \cdot D^{\alpha-2})$, and the total data transmission is bounded by $O(D + n \cdot S^{\alpha})$.

Proof. For load imbalance, the workload scales from w to $S \cdot w$, increasing the variance by at most S^2. Applying scaled Chebyshev's inequality to the new imbalance gives:

$$\begin{aligned}&\Pr\left(\left|\frac{n}{m}Sw - 1\right| \geq t\right)\\ \leq&\frac{1}{t^2}\mathrm{Var}\left[\left|\frac{n}{m}Sw - 1\right|\right]\\ \leq&\frac{S^2}{t^2}\mathrm{Var}\left[\left|\frac{n}{m}w - 1\right|\right]\\ \leq&O(S^2 \cdot D^{\alpha-2})\end{aligned}$$

Similarly, for data transmission, using the weighted frequencies, the number of nodes in the node group of i is now $\lceil n f_i^{\alpha} \rceil \leq 1 + n(s_i \cdot f_i^{\text{unit}})^{\alpha} \leq 1 + S^{\alpha} \cdot (f_i^{\text{unit}})^{\alpha}$, which gives the transmission cost $\mathcal{T} = O(D + nS^{\alpha} \sum_{d=1}^{D} (f_d^{\text{unit}})^{\alpha}) = O(D + n \cdot S^{\alpha})$. Here, the last equality holds since $\sum (f_d^{\text{unit}})^{\alpha} \leq 1$. □

D EVALUATION IN SIMULATION ENVIRONMENT

D.1 Evaluation in Simulation Environment

We implement a simulator to evaluate the robustness of *HotHash* to various hardware configurations, including CPU capacities, cache memory sizes, and network bandwidths. The simulation experiments use the same datasets, workloads, and parameter configurations as the real cloud environment experiments. Moreover, we introduce an extra baseline, called *HotHash without virtual hash ring*, to separately evaluate the effectiveness of *range hashing* and *virtual hash ring*. This "range hashing only" HotHash significantly outperforms *Consistent Hashing*, *SPORE*, and *Bounded Load* from 30% up to 60×, confirming the effectiveness of range hashing. However, it is consistently slower than the full-fledged *HotHash* by 2× to 10×, indicating that *virtual hash ring* effectively reduces *hash collision* and thus further improves the performance of HotHash.

Network Bandwidth. First, we evaluate the performance of each method under different network bandwidths. We use the default data and workload configuration (Sec. 6.1).

As reported in Fig. 19, *HotHash* outperforms *Consistent Hashing*, *SPORE*, and *BalancedHash* by 10×, and *Bounded Load* by 50× in both execution time and tail latency. This trend is consistent with our real-world experiment results in Sec. ??; and this validates our simulation implementation.

All six methods benefit from larger network bandwidth since this directly reduces the cost of reading remote storage. However, when the network bandwidth increases to 10Gbps, *HotHash* still outperforms *BalancedHash*, *Consistent Hashing*, *SPORE*, and *Bounded Load* by 5×, 10×, 10×, and 30×, respectively. The gain comes from our innovative *range hashing* and *virtual hash ring*.

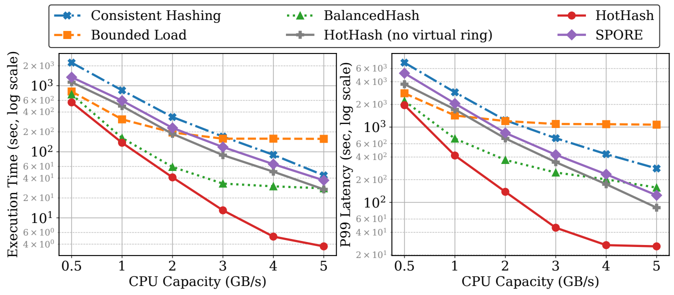
Cache Size. Next, we vary the cache sizes. As shown in Fig. 20, *HotHash* is consistently better than all baselines. When the cache size is under 4GB, *HotHash* is 10× faster than *SPORE* and *BalancedHash*, and 50× faster than *Bounded Load* in execution time; and 25× faster than *SPORE* and *BalancedHash*, and 60× faster than *Bounded Load* in tail latency.

When the cache size increases, the performance of *Bounded Load* and *BalancedHash* improves, as a larger cache size reduces the chance of cache replacement and hence the I/O cost. However, *HotHash* still outperforms all baselines by at least 5× in both execution time and tail latency. This is because *Bounded Load* and *BalancedHash* still need to access remote storage to generate cache replications on many nodes when re-distributing queries for load balance. *Consistent Hashing* benefits less from a larger cache size since it has the best cache hit rate. However, it performs the worst due to severe load imbalance. Similar to *Consistent Hashing*, *SPORE* performs worse than *BalancedHash* and *HotHash*, and benefits less from a larger cache size.

CPU Capacity. Next, we study the impact of CPU capacity on each method. We simulate the CPU capacity as the size of the data that a compute node can process per second. In our real cloud environment, each compute node roughly processes 2.5GB per second on a query workload that mixes analytics operations with different complexities, including `sum`, `min/max`, `mean`, and `sort`. Accordingly, we vary the CPU capacity from 0.5GB/s to 5GB/s.

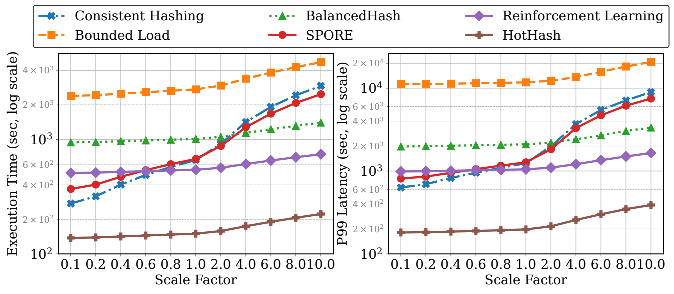
As shown in Fig. 21, *HotHash* consistently outperforms all baselines, similar to the real cloud experiment in Sec. ??. *Consistent Hashing* is 1.5× slower than *SPORE* and at least 3× slower in both execution time and tail latency than *Bounded Load*, *BalancedHash* and *HotHash* under the lowest CPU capacity since it is the worst in balancing the load on nodes.

All six methods benefit from larger CPU capacity. When the capacity is greater than 2GB/s, the I/O time starts to become a



(a) Average Execution Time (b) P99 Latency

Figure 21: Varying CPU Capacity



(a) Average Execution Time (b) P99 Latency

Figure 22: Varying Operator Execution Time

bottleneck for *Bounded Load* and *BalancedHash*. Because *HotHash* effectively reduces data transmission cost while balancing node workload, it benefits more from the increase of the CPU capacity, resulting in a larger gain against *Bounded Load* and *BalancedHash*. Although *SPORE* outperforms *Consistent Hashing*, it still suffers from insufficient load balance, thus performing much worse than *HotHash*. In particular, when the CPU capacity increases to 5GB/s, *HotHash* is 10× faster than *SPORE* and *BalancedHash*, and is 50× faster than *Bounded Load* in both execution time and tail latency.

As expected, the performance of *Consistent Hashing* and *SPORE* improves quickly with higher CPU capacity because a more advanced and thus a more expensive CPU mitigates its load imbalance disadvantage. However, they are still nearly 10× slower than *HotHash*, although their performance eventually approaches *BalancedHash* – the second best.

E THE EFFECTIVENESS OF HOTHASH IN REAL CLOUD DATABASES

Next, we approximate the effectiveness of HotHash in real cloud databases by varying the execution time of the operators. More specifically, the gap between our prototype system and a real cloud database mainly is due to the efficiency of the query engines. Different query engines could incur different query execution costs on the same operator. In cloud databases, the overall query execution time depends on the data transmission cost and the query execution cost. Therefore, if the execution engine is super fast and thus the data transmission cost dominates the overall execution time, then balancing loads on compute nodes might be less critical. This potentially impacts the performance gain of HotHash, as HotHash aims to achieve both data locality to reduce data transmission cost and load balance to reduce query execution time. On the other hand, if a real query engine is slower than our experiment prototype, HotHash could potentially achieve a larger gain.

To address this gap, we have conducted a new experiment in which we use our prototype operators as bases and apply different scale factors to their execution time to simulate query engines with different performance. Specifically, we vary this factor from 0.1 to 10 so that the efficiency of the query engine can be controlled to vary from 10× slower to 10× faster than that of our prototype. We report the results in Fig. 22 of our technical report [], showing that HotHash is still consistently faster than all baselines from 2× to 50× in average execution time and tail latency. When the scale factor increases, as expected, the performance gain of HotHash is slightly smaller but still significant. This shows that HotHash is generally effective when working with query engines with different performance. This approximately validates the effectiveness of HotHash in real cloud databases.