HotHash: Hotness-Aware Consistent Hashing for Cloud Databases

Junyong Zhao[+], Zui Chen[*], Jia Yuan[+], Yi Lu[†], Lei Cao[+], Samuel Madden[*]
University of Arizona[+], MIT[*], Google[†]
(junyong,jiayuan)@arizona.edu,magolorcz@gmail.com,yilux@google.com,(lcao,madden)@csail.mit.edu

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 [18, 19] is widely used for *affinity scheduling* in cloud databases [10, 17, 30, 36] due to its robust handling of node additions and removals. 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. 8, 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 completely eliminates the load imbalance issue introduced by affinity scheduling when handling skewed workloads. HotHash targets two objectives: (1) ensuring data locality and load balancing with *theoretical guarantees* and (2) ease of adoption.

Challenges. To achieve these objectives, HotHash has to address the following challenges.

- **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 [6, 23] 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 the pre-defined upper bound, Bounded Load will forward a service request to another node. Although Bounded Load balances the load on nodes,

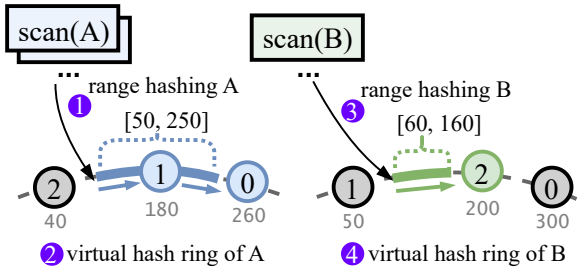


Figure 1: Overview of HotHash

it overlooks data I/O cost. This is because, as further discussed in Sec. 4.2, 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.

Range hashing addresses the node group size and node identification challenges. Unlike consistent hashing which maps each data segment to *one single location* on the ring, range hashing maps one data segment to a *range* of the hash ring, where its start point is decided by consistent hashing and its length is *proportional to* the hotness of the data. The queries accessing this data segment will only be distributed among the nodes that own a part of this range on the hash ring, thus preserving data locality. Range hashing also 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 effectively 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 the 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 consistent hashing, increasing the *collision probability* of queries that access different data segments. This could lead to a less balanced distribution of queries, ultimately resulting in worsened 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 the 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 the 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.

Fusing range hashing into 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. 6. Moreover, because HotHash still uses the idea of a hash ring to map data segments to compute nodes, it is compatible with consistent hashing, robust to node changes, and thus is easy to adopt, as further shown in Sec. 5.2.

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. Furthermore, our simulation experiments (Sec. 7.3) show its robustness to hardware configurations.

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 [31], F1 Query [30], Aurora [34, 35], Snowflake [10, 36] and SparkSQL [4], adopt a storage-disaggregation architecture. This design separates the computation and storage, bringing unique advantages such as higher availability and better scalability. Moreover, it could scale the computation and data storage differently by adding and removing nodes on demand.

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 [33].

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* [18, 19] to distribute queries to compute nodes, ensuring that subsequent or concurrent

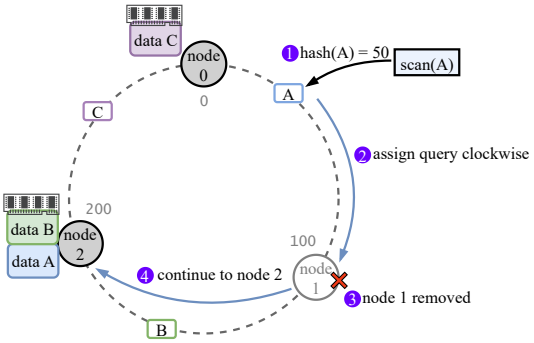


Figure 2: Consistent Hashing: Robust to Change

queries accessing the same data segment will be assigned to the same node [10, 16, 30], 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 can be placed at random locations on this ring. The requests, which are analogous to keys in the classic hashing approach, are also placed on the same 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, only the queries that fall into the range this node owns will have to be re-assigned. For example, 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

Load imbalance caused by skewed query workloads is one of the main causes of stragglers in cloud databases which significantly slows down the query execution [9]. Targeting minimizing query latency under skewed query workloads, we design a new hashing-based query scheduling mechanism, called *HotHash*, which meets the following objectives:

- **Load Balance**. An ideal mechanism should automatically balance the load across compute nodes.
- **Data Locality.** Query latency mainly constitutes the I/O time and computation time. Load balance reduces computation time, while data locality reduces I/O time by minimizing data transmission. To minimize query latency, any mechanism should balance load balance while preserving data locality; these objectives often compete with each other.
- **Compatible with Consistent Hashing.** An ideal mechanism should preserve the key properties of consistent hashing, e.g., simplicity and robustness to node changes.

Next, in Sec. 3, we overview the key ideas and the high-level design choices of HotHash. In Sec. 4 we present a basic solution BalancedHash that adapts Bounded Load [6, 23] into cloud databases. Then in Sec. 5 we present the details of our HotHash which overcomes the limit of BalancedHash. Sec. 6 establishes the theoretical guarantees of HotHash in both data locality and load balance.

3 HOTHASH OVERVIEW

3.1 HotHash Key Ideas

HotHash achieves the objectives listed in Sec. 2.3 with the following two principles: (1) introducing random routing into consistent hashing; (2) taking into consideration the hotness of the data.

Our key ideas are inspired by the pros and cons of the two fundamental query scheduling strategies, namely *affinity scheduling* and *random scheduling*. As the most popular affinity scheduling strategy, consistent hashing assigns queries that access the same data segment to the same node. Although it achieves perfect data locality and hence minimal I/O costs, it *overlooks load balance* across the compute nodes, where hot, overloaded nodes become stragglers and significantly slow down query execution. On the other hand, random scheduling such as round-robin distributes different queries equally to different nodes. This ensures load balance with a price of *poor data locality*, thus suffering from high I/O costs.

In cloud databases, minimizing query latency must consider both data locality and workload balance. Therefore, we propose HotHash, which, by *introducing randomness into consistent hashing in a controlled way*, unifies the merits of affinity scheduling and random scheduling. Unlike conventional consistent hashing that routinely distributes queries based on data locality, in HotHash, queries that access the same data segment can be distributed to different nodes, conditional on the workload of the nodes.

Conceptually, given a query, HotHash *probabilistically* decides on a group of nodes that the query can be distributed to and randomly picks one from this group of nodes to host it. The number of nodes in the group is dependent on the *hotness* of the data segment that the query accesses. 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.

Note although databases such as DynamoDB [11] also use consistent hashing to map a data item to multiple nodes, unlike HotHash,

DynamoDB always replicates data to *successive nodes* on the ring with a *constant replication factor*, e.g, 3.

3.2 Design Choices

Next, we first discuss HotHash's two key design decisions about caching, namely what to cache and at which granularity. We then discuss the queries that HotHash supports.

Caching Raw Input Data. Same as other works in cloud databases [10, 36, 38], HotHash focuses on caching the raw input tables and optimizing the performance of scan operators, as scan operators involve fetching data from remote storage through networks. Accordingly, when estimating the cost of queries and the load on nodes, HotHash ignores the complexity of the queries and only takes into account the size of the data segments. However, the problem of effectively estimating the cost of queries and the load on nodes is orthogonal to the key ideas of HotHash, and thus any such technique can be adopted here.

Storage and Caching Granularity. HotHash assumes tables are stored in a distributed cloud storage service. HotHash horizontally partitions the tables into *blocks* and stores each block as a file in Parquet or other open file formats. The basic caching unit in HotHash is a *data segment*, which contains data for a particular column in a block, because in cloud databases queries typically only download the columns they need. Each data segment has a unique ID, used as the hash key by consistent hashing.

Queries. For ease of presentation, in this paper, we assume each query only accesses one data segment. However, HotHash naturally supports queries accessing multiple data segments, e.g., by hashing different segments independently to the nodes. HotHash supports join operation as well which involves data segments from two different tables. This could be done by hashing each pair of segments that potentially produces join results with a composite key.

We particularly focus on analytical query workloads where objects are immutable. Mutable objects can be implemented as immutable objects with versions (e.g., in Amazon S3, an object can have multiple versions, each treated as a separate immutable object). This approach allows HotHash to leverage replication for load balancing without consistency concerns.

4 BALANCEDHASH: THE FIRST ATTEMPT

In this section, we first discuss *BalancedHash*, which enhances *Bounded Load* [6, 23] with the randomness idea, making it a better match to cloud databases. We then show its limitations in Sec. 4.2, which motivate the design of HotHash.

4.1 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.

There are different ways to decide the next hop of a query q. Bounded Load [23] routes the query to the next node in the hash ring along the clockwise direction.

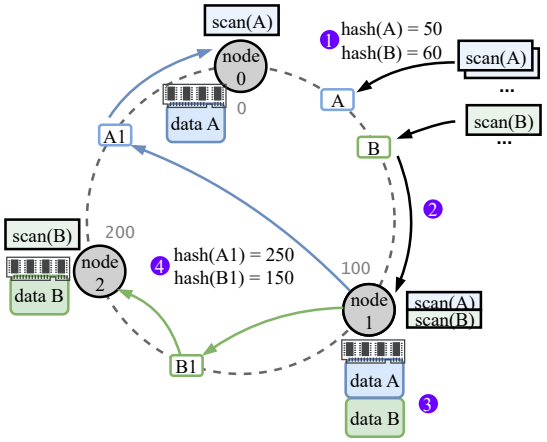


Figure 3: BalancedHash

However, although intuitive, Bounded Load increases the *collision probability* [6] of queries that access different data segments, hence more likely overloading the nodes. For example, assume query q_1 that scans data segment A and query q_2 that scans data segment B are both mapped to $node_1$. Because $node_1$ is overloaded, Bounded Load will route both queries to $node_2$. This could easily overflow $node_2$. Therefore, Bounded Load tends to produce a less balanced query distribution and potentially needs more hops to find a non-overloaded node.

Routing with Randomness. BalancedHash avoids this problem by adopting one of our key ideas, namely introducing *randomness* [6] 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. For example, as shown in Fig. 3, because $node_1$ has already reached the workload bound (3 in this example), BalancedHash modifies the hash key A by adding a suffix to denote routing attempts. It then rehashes this query with the new key $A1$ and maps it to $node_0$. If $node_0$ is also overloaded, BalancedHash will continue to modify the hash key (i.e. $A2$, $A3$, ...) and route the query to another node in the same way.

Now given the two example queries q_1 and q_2 accessing different data segments (i.e., A and B), when routing them to the next hop, BalancedHash will hash them with different hash keys (i.e. $A1$ and $B1$). Compared to Bounded Load, this largely reduces the collision probability, resulting in a more uniformed query distribution. For example, in the next hop, now BalancedHash will route $q_1 : scan(A)$ to $node_0$ and $q_2 : scan(B)$ to $node_2$.

The Workload Bound. BalancedHash defines the workload bound as follows [23]:

Definition 4.1. 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.

Defining the load bound B in this way, BalancedHash requires the load of a node to be not much larger than the average workload per node, while the parameter ϵ defines the *overload factor* that a system can tolerate. The larger the ϵ is, the more the system allows

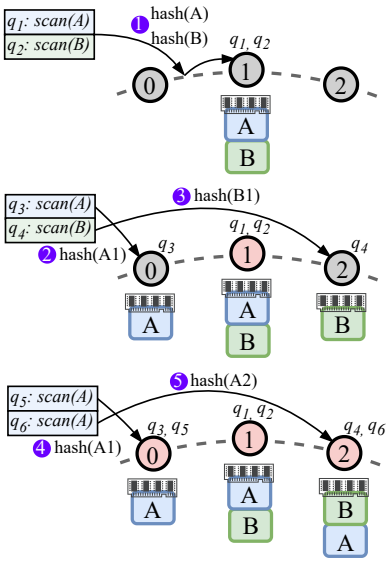


Figure 4: Limitation of BalancedHash

overloading a node. When ϵ is set to 0, the system is expected to achieve perfect load balance.

In this work, we estimate the current workload L_c as $\sum_{i=1}^{m} l(q_i)$, where $l(q_i)$ denotes the estimated load (cost) of a query q. We assume the system has m queries to process.

4.2 Limitations of BalancedHash

Although BalancedHash ensures load balance by introducing a load bound on each node, potentially a data segment, even if it is not hot, could be spread all over the nodes, leading to high data transmission cost and thus poor data locality.

Example 4.1. For example, as shown in Fig. 4, 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. Therefore, A is hotter than B. The load bound of each node is 2. BalancedHash schedules these queries one by one. Because $hash(A)$ and $hash(B)$ fall into the range that $node_1$ owns, it will assign $q_1 : scan(A)$, $q_2 : scan(B)$ to $node_1$. Now $node_1$ reaches the load bound. Therefore, BalancedHash will rehash $q_3 : scan(A)$ and $q_4 : scan(B)$ with $hash(A1)$ and $hash(B1)$, assigning them to $node_0$ and $node_2$, respectively. Similarly, BalancedHash will assign $q_5 : scan(A)$ to $node_0$. Now $node_0$ is full as well. Therefore, BalancedHash has to assign $q_6 : scan(A)$ to $node_2$. Clearly, this schedule is not ideal, because BalancedHash unnecessarily replicates B to 2 nodes and A to 3 nodes. Ideally, B should only be on 1 node and A on 2 nodes.

5 OUR PROPOSED APPROACH: HOTHASH

To overcome the issues of BalancedHash, we introduce HotHash. We focus on the key ideas of *range hashing* and the *virtual hash ring*, which together balance load and preserve data locality.

Basic Ideas of HotHash. The key insight of HotHash is that it fuses data hotness into a randomized routing scheme. Specifically, HotHash explicitly takes data hotness into consideration, while still preserving the load balancing benefit of random routing. HotHash achieves this with two key techniques:

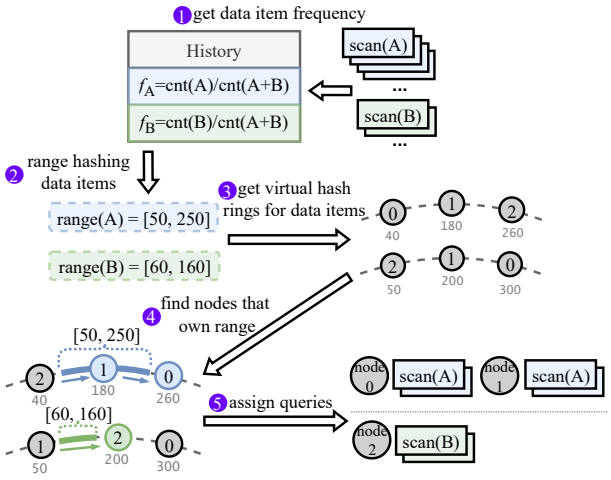


Figure 5: HotHash

Range Hashing. Rather than setting a hard workload constraint on each node and routing a query all over the hash ring, *HotHash* routes a query within a group of nodes that *fall into a certain range of the hash ring*, where the range of the ring is proportional to the hotness of the data segment. The hotter the data segment is, the larger the range is. This reduces load imbalance and increases the availability of hot data, while at the same time preventing the spreading of data segments all over the nodes in the ring.

Virtual Hash Ring. Although this ensures data locality, each data segment using a fixed range of the hash ring increases the collision probability among the queries that access different data segments, similar to the problem that Bounded Load [23] suffers (Sec. 4). HotHash solves this issue by preserving the *random nature of BalancedHash*. In HotHash, each data segment sees a specific (virtual) hash ring corresponding to a random permutation of the nodes in the cloud databases. This is equivalent to routing a query within a randomly selected set of nodes, thus effectively reducing the conflict probability. On the other hand, because these nodes fall into a fixed range on the ring designated to a data segment, HotHash is still able to route queries that access the same data segment to the same set of nodes, thus preserving data locality.

5.1 The HotHash Technique

Next, we describe the details of HotHash, composed 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) it then generates a hash ring for d_i; and (3) routes query q_i among the nodes that fall into the corresponding range of this hash ring.

Range Computation. Range computation in HotHash relies on the hotness of the data. Therefore, we start with designing a lightweight mechanism to measure this metric.

Sliding Window-based Data Hotness. To estimate the hotness of the data over time, HotHash keeps track of the query histories falling into a time window and the data segments these queries use. Note although we use the data segment frequency as the metric to measure hotness, any other metrics can be equally plugged into HotHash. For each historical query, 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 the queries in a window.

More formally, we define data frequency as follows:

Definition 5.1. 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. 5 step 1 shows an example of frequency computation. Given the query workload described in Example 4.1, 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 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:

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

Fig. 5 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 $range(B) = [60, 160]$.

Virtual Node Group Selection. In order to more evenly distribute queries, HotHash decides a set of nodes a query should be distributed to. 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. 5 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. 5 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. Finally, 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 over the same group of nodes. As depicted in Fig. 5 step 5, $scan(A)$ will be uniformly distributed to $node_0$ or $node_1$, while $scan(B)$ always goes to $node_2$.

Algorithm 1: HotHash

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

Alg. 1 shows how HotHash works. For each query q_i that accesses data segment d_i, HotHash first computes the hash value $loc = hash(d_i)$ and the data frequency f_{d_i} (Lines 2-3). It 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 $hash(d_i)$ as seed and store it for future use (Lines 5-7). HotHash then produces the node group *nodes* by finding any node where the range it owns on the 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 to find a node to serve q_i.

5.2 Handling Node Removals and Additions

We show that HotHash is able to keep the data-node mapping at a minimum cost as 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 the 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. 5 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.

6 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. 6.1. We establish the bounds on load imbalance and data locality in Sec. 6.2.

6.1 Notation

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

Definition 6.1. 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 6.2. 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 6.3. 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 6.4. 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.

6.2 Bound on Workload Imbalance and Cache Hit rate

In this section, we first establish HotHash's upper bound on load imbalance in Theorem 6.1. We then show a lower bound on its cache hit rate in Theorem 6.2.

We introduce a parameter α ($\geq$ 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 6.1. 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 6.2. 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. □

Here 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. As for the data transmission cost $\mathcal{T}$ (Eq. 5), we will show in Sec. 6.3 that HotHash is guaranteed to be better than BalancedHash.

6.3 HotHash Is Better In Theory

Next, we theoretically show that HotHash is superior to BalancedHash in both data transmission cost and cache hit rate. We do not compare HotHash to *Bounded Load*, as *Bounded Load* is worse than BalancedHash in cache hit rate.

We start by establishing 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 $\left(\frac{n}{1+\varepsilon} - 1\right) \times L = \left(\frac{n}{1+\varepsilon} - 1\right)\left(1 + \frac{1}{\varepsilon}\right)$. Therefore, the worst-case data transmission cost for BalancedHash corresponds to $\mathcal{T}_{BH} = \Omega\left(D + \left(\frac{n}{1+\varepsilon} - 1\right)\left(1 + \frac{1}{\varepsilon}\right)\right) = \Omega\left(D + n/\varepsilon\right)$. To ensure load balance, ε has to be small (< 1) [6], 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.

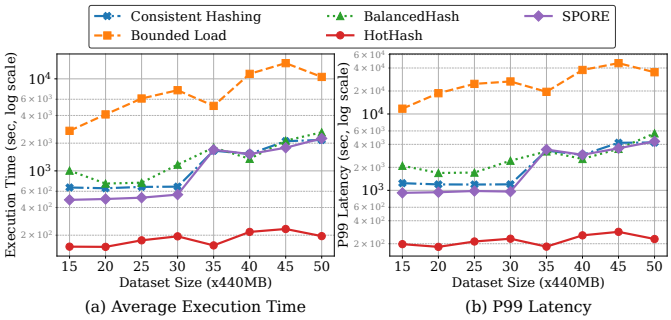
7 EXPERIMENTS

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

- How does *HotHash* perform compared to baselines under different query workloads with different skewness factors?
- Is *HotHash* scalable to large datasets?
- How do hardware resources affect the performance of *HotHash* compared to baselines?

7.1 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*.



(a) Average Execution Time (b) P99 Latency

Figure 6: Varying Dataset Sizes (440MB per Data Block)

Data and Query Workload. As discussed in Sec. 3.2, HotHash partitions tables into blocks and caches data at the *data segment* level. Each data segment in a block corresponds to a column that a query requests. 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 150MB and 10MB, similar to FlexpushdownDB [38], 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 SQL queries according to Yahoo! Cloud Serving Benchmark (YCSB) [8]. 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) [25]. Similar to FlexpushdownDB [38], we introduce skewness to queries accessing the *lineorder* table, and queries are generated to favor data within certain dates.

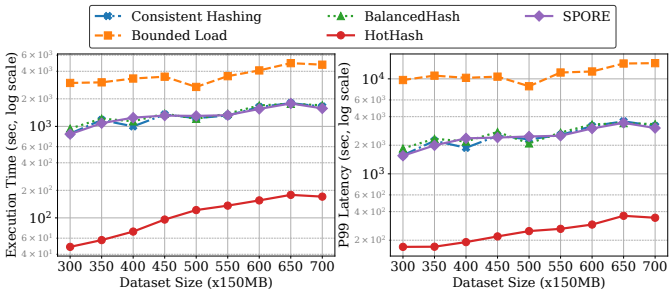
We use the Zipfian [14] distribution to control the distribution of the query workload and vary the skewness with a parameter θ. A larger θ indicates higher skewness and a θ that is close to 1 means low skewness [1]. We also generate data according to a uniform distribution such that all data segments are equally likely to be accessed. 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.

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:

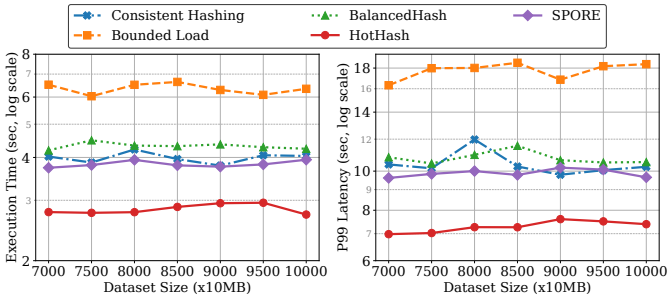
- *Consistent Hashing*: The original consistent hashing [18] that assigns a query based on the data it requires.
- *Bounded Load*: Assign a query based on the data it requests and the workload of the node [23]. Distribute queries to the next node if a node is considered overloaded (Sec. 4).
- *BalancedHash*: Assign a query based on the data it requests and the workload of the node. Distribute queries via re-hashing to avoid cascade overflow [6] (Sec. 4).
- *SPORE*: Assign a query based on the data it requests and replicate a hot data with a fixed data frequency threshold and replication factor [15] (Sec. 8).

Configurations. By default, we use a dataset with 15 data blocks. To evaluate *HotHash*'s performance on datasets with more data



(a) Average Execution Time (b) P99 Latency

Figure 7: Varying Dataset Sizes (150MB per Data Block)



(a) Average Execution Time (b) P99 Latency

Figure 8: Varying Dataset Sizes (10MB per Data Block)

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 (hot queries) that request 20% of data (hot data). For *Bounded Load* and *BalancedHash*, we use a default balancing parameter $\epsilon = 0.3$ according to [6, 23]. 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 [15] 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 [15], *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.

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).

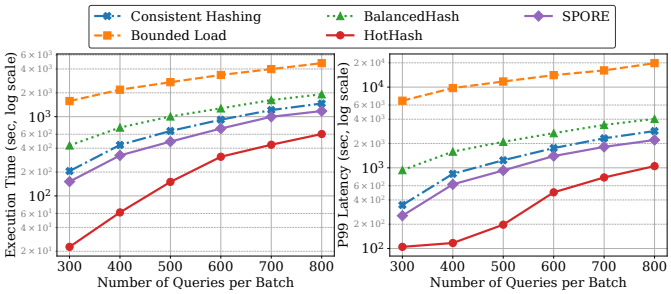
7.2 Evaluation in Real Cloud Environment

We first evaluate the performance of *HotHash* by varying the scales of data and the properties of query workloads. We then study the impacts of the ϵ and α parameters.

7.2.1 *Varying Data Sizes and Query Workloads*

Dataset size. In this set of experiments, we vary the size of the dataset by varying the number of data blocks and the block sizes, but keep the skewness factor of the workload fixed.

Fig. 6 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 more data blocks with different sizes and report the results in Fig. 7 and Fig. 8.



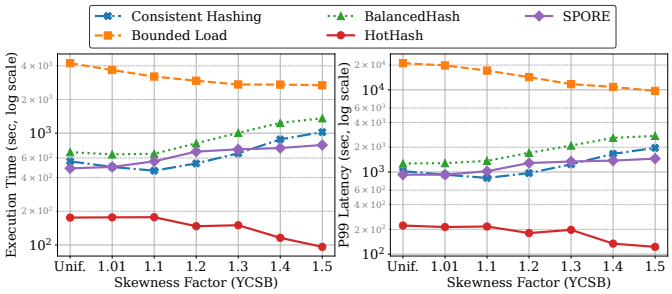
(a) Average Execution Time (b) P99 Latency

Figure 9: Varying Workloads

Our *HotHash* consistently outperforms all baselines by a large margin, from 3× up to 150×. For the default dataset configuration (Fig. 6), when the dataset has fewer than 30 data blocks, *HotHash* outperforms *SPORE* by 3×, *Consistent Hashing* and *BalancedHash* by 5×, and *Bounded Load* by 25× in average execution time. The tail latency shows a similar trend in Fig. 6 (b). *HotHash* constantly outperforms *SPORE* by 5×, *Consistent Hashing* and *BalancedHash* by 8×, and *Bounded Load* by 100×. When the dataset contains more than 35 data blocks, *HotHash* wins more: 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*.

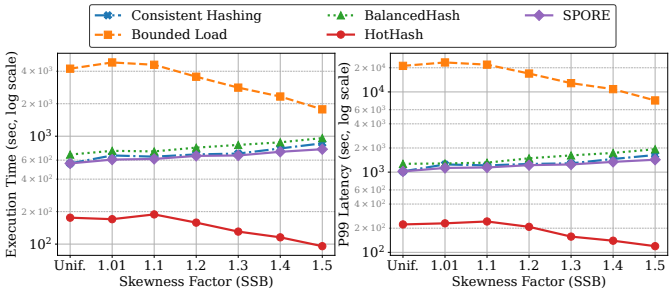
In Fig. 7 and Fig. 8, *HotHash* shows a similar performance gain on datasets with many more data blocks. When the size of each data block is 150MB, Fig. 7 (a) shows that *HotHash* outperforms *SPORE*, *Consistent Hashing* and *BalancedHash* by 10×, and outperforms *Bounded Load* by 25× in average execution time. Fig. 7 (b) shows that *HotHash* is 15× faster than *SPORE*, *Consistent Hashing* and *BalancedHash*, and is 50× faster than *Bounded Load* in tail latency. When the size of the data block is 10MB, Fig. 8 shows that *HotHash* is consistently 40% faster than *SPORE*, *Consistent Hashing* and *BalancedHash*, 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. 4.2, 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 balance the workload to some extent, it uses a fixed number of nodes to host hot data segments, leading to insufficient load balance. On the other hand, *HotHash*'s range hashing dynamically determines the size of the node group hosting hot data segments. This achieves a good load balance while avoiding unnecessarily replicating data. This advantage becomes more apparent as the size of the datasets increases.



(a) Average Execution Time (b) P99 Latency

Figure 10: Varying Workload Skewness with YCSB



(a) Average Execution Time (b) P99 Latency

Figure 11: Varying Workload Skewness with SSB

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

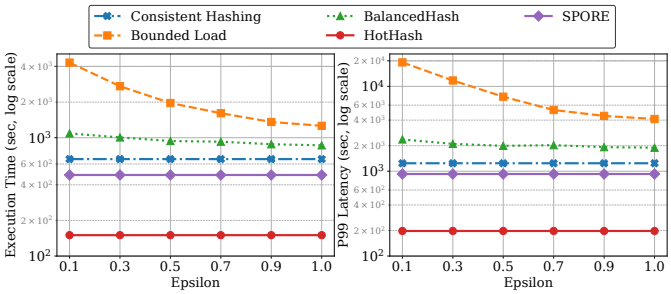
Our *HotHash* significantly outperforms all baselines. As shown in Fig. 9 (a) and Fig. 9 (b), *HotHash* is about 7× faster than *SPORE*, 10× faster than *Consistent Hashing* and *BalancedHash*, and 70× faster than *Bounded Load* in average execution time. In terms of tail latency, *HotHash* outperforms *SPORE* by 2.5×, *Consistent Hashing* and *BalancedHash* by 8×, and *Bounded Load* by 60×.

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. 10 and Fig. 11 show the results on both the YCSB and SSB workloads. On both benchmarks, 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×. In particular, when the workload is close to uniform, *HotHash* is still 3.5× faster than *SPORE*, *Consistent Hashing*, and *BalancedHash*, and 25× faster than *Bounded Load* in execution time, while *HotHash* is 5× faster than *SPORE*, *Consistent Hashing*, and *BalancedHash*, and 100× faster than *Bounded Load* in tail latency.

This is because all methods experience workload imbalance even if the query workload is near uniform due to hash collisions. Therefore, *Bounded Load* and *BalancedHash* still incur a large data transmission cost because of re-distributing queries, while *Consistent Hashing* accumulates many queries on some nodes and overloads these nodes. Because uniform workload does not trigger *SPORE*'s re-hashing mechanism, its performance is close to *Consistent Hashing*. On the other hand, the virtual hash ring in *HotHash* distributes queries accessing different data segments on different hash rings, effectively reducing collision probability.



(a) Average Execution Time (b) P99 Latency

Figure 12: Varying Epsilon

Note 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 the 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. When θ is 1.5, these three baselines are 10× and 16× slower than *HotHash* in execution time and tail latency.

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

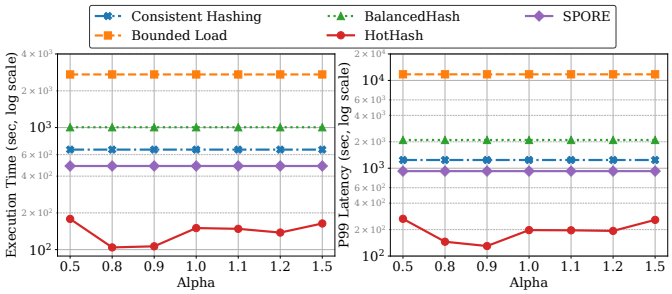
7.2.2 *Varying Parameters*

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. 12, *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*.

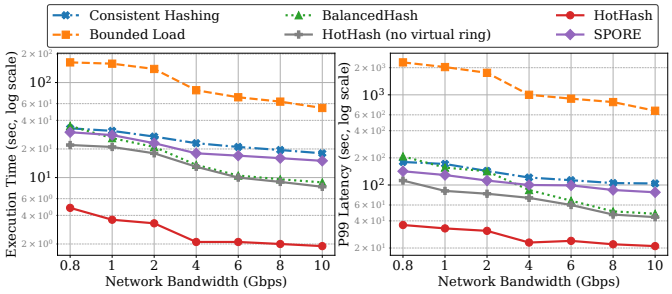
When ε increases, the performance of *Bounded Load* and *BalancedHash* improves, although both methods are still slower than *HotHash* by at least 5×. This is because a large ε trade-off load balance with cache hit rate, while a better cache hit rate reduces data transmission cost.

Varying Alpha. We evaluate how the parameter α affects *HotHash*. We vary the value of α and compare to other four baselines, which are not affected by this parameter. As shown in Fig. 13, *HotHash* consistently outperforms *Consistent Hashing*, *SPORE* and *BalancedHash* by 10× and *Bounded Load* by 50×, indicating that it is not a parameter that is hard to set.



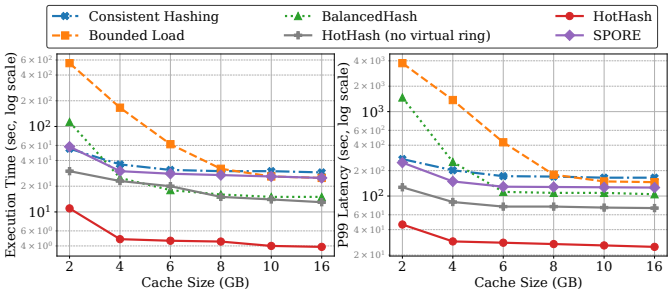
(a) Average Execution Time (b) P99 Latency

Figure 13: Varying Alpha



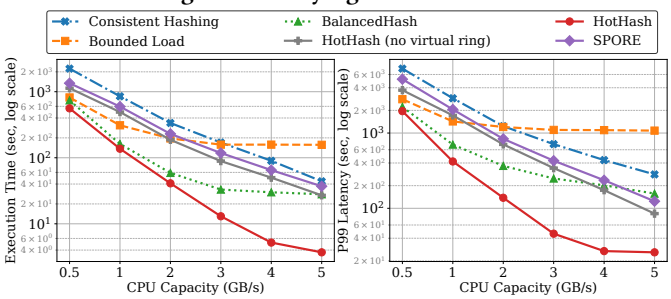
(a) Average Execution Time (b) P99 Latency

Figure 14: Varying Network Bandwidth



(a) Average Execution Time (b) P99 Latency

Figure 15: Varying Cache Size



(a) Average Execution Time (b) P99 Latency

Figure 16: Varying CPU Capacity

7.3 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. 7.1).

As reported in Fig. 14, *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. 7.2; 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. 15, *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. 16, *HotHash* consistently outperforms all baselines, similar to the real cloud experiment in Sec. 7.2. *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 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.

Summary. The simulation experiments conclude that *HotHash* achieves much better performance than all baselines with low-cost hardware, while readily benefiting from advanced hardware.

8 RELATED WORK

Consistent Hashing. In addition to cloud databases [10, 11], consistent hashing is widely adopted in multiple areas, including web caching, distributed storage systems, and peer-to-peer networks.

Consistent hashing was initially introduced in [18, 19] for Web caching. This algorithm has inspired many subsequent research in related fields. For in-memory caching systems, Memcached [12] and Memcached-based systems [7, 24, 40] use consistent hashing to map hash keys to cache servers. Many distributed key-value storage systems, including Apache Cassandra [21] and Linked-in Voldemort [20] use consistent hashing to partition and distribute data items to storage servers. In addition, many distributed hash table (DHT) systems and peer-to-peer networks are designed based on consistent hashing [5]. For example, Chord [32], Pastry [29], and CAN [27] 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 [6, 23], 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, thus incurring large data transmission costs. As shown in our experiments (Sec. 7), HotHash, which balances the loads on nodes based on the data hotness, outperforms Bounded Load-based methods by a large margin.

Similar to [23], the idea of Bounded Load is also used in the content delivery networks (CDN) [22] where a hot object is replicated to multiple successor nodes. This increases the collision probability when mapping hot objects to nodes, as we have analyzed in Sec. 4.

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 [15] 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. 7).

EC-Cache [26] 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 [10] 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 [13] 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 [39]. LATE performs speculative detection on each node to detect stragglers and launches copied tasks on faster nodes to accelerate the job. In addition, Hopper [28] 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 [37] 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.

9 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] Numpy, 2023.

[2] Adya, A., Myers, D., Howell, J., Elson, J., Meek, C., Khemani, V., Fulger, S., Gu, P., Bhuvanagiri, L., Hunter, J., et al. Slicer:{Auto-Sharding} for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 739–753.

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

[4] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (2015), pp. 1383–1394.

[5] Awerbuch, B., and Scheideler, C. Towards a scalable and robust dht. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures* (2006), pp. 318–327.

[6] Chen, J., Coleman, B., and Shrivastava, A. Revisiting consistent hashing with bounded loads. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2021), vol. 35, pp. 3976–3983.

[7] Cheng, Y., Gupta, A., and Butt, A. R. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), pp. 1–16.

[8] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154.

[9] Coppa, E., and Finocchi, I. On data skewness, stragglers, and mapreduce progress indicators. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), pp. 139–152.

[10] Dageville, B., Cruanes, T., Zukowski, M., Antonov, V., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 215–226.

[11] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review 41*, 6 (2007), 205–220.

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

[13] Gill, S. S., Ouyang, X., and Garraghan, P. 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.

[14] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data* (1994), pp. 243–252.

[15] Hong, Y. J., and Thottethodi, M. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), pp. 1–17.

[16] Huang, Q., Gudmundsdottir, H., Vigfusson, Y., Freedman, D. A., Birman, K., and van Renesse, R. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks* (2014), pp. 1–7.

[17] Kalid, S., Syed, A., Mohammad, A., and Halgamuge, M. N. 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)* (2017), IEEE, pp. 89–93.

[18] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 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* (1997), pp. 654–663.

[19] Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., and Yerushalmi, Y. Web caching with consistent hashing. *Computer Networks 31*, 11-16 (1999), 1203–1213.

[20] Kreps, J. Project voldemort, 2012.

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

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

[23] Mirrokni, V., Thorup, M., and Zadimoghaddam, M. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (2018), SIAM, pp. 587–604.

[24] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 385–398.

[25] O'Neil, P. E., O'Neil, E. J., and Chen, X. The star schema benchmark (ssb). *Pat 200*, 0 (2007), 50.

[26] Rashmi, K., Chowdhury, M., Kosaian, J., Stoica, I., and Ramchandran, K. {EC-Cache}:{Load-Balanced},{Low-Latency} cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 401–417.

[27] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (2001), pp. 161–172.

[28] Ren, X., Ananthanarayanan, G., Wierman, A., and Yu, M. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 379–392.

[29] Rowstron, A., and Druschel, P. 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* (2001), Springer, pp. 329–350.

[30] Samwel, B., Cieslewicz, J., Handy, B., Govig, J., Venetis, P., Yang, C., Peters, K., Shute, J., Tenedorio, D., Apte, H., et al. F1 query: Declarative querying at scale. *Proceedings of the VLDB Endowment 11*, 12 (2018), 1835–1848.

[31] Sethi, R., Traverso, M., Sundstrom, D., Phillips, D., Xie, W., Sun, Y., Yegitbasi, N., Jin, H., Hwang, E., Shingte, N., et al. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), IEEE, pp. 1802–1813.

[32] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review 31*, 4 (2001), 149–160.

[33] Tan, J., Ghanem, T., Perron, M., Yu, X., Stonebraker, M., DeWitt, D., Serafini, M., Aboulnaga, A., and Kraska, T. Choosing a cloud dbms: architectures and tradeoffs. *Proceedings of the VLDB Endowment 12*, 12 (2019), 2170–2182.

[34] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., and Bao, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 1041–1052.

[35] Verbitski, A., Gupta, A., Saha, D., Corey, J., Gupta, K., Brahmadesam, M., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvilli, T., et al. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 789–796.

[36] Vuppalapati, M., Miron, J., Agarwal, R., Truong, D., Motivala, A., and Cruanes, T. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 449–462.

[37] Yadwadkar, N. J., Ananthanarayanan, G., and Katz, R. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), pp. 1–14.

[38] Yang, Y., Youill, M., Woicik, M., Liu, Y., Yu, X., Serafini, M., Aboulnaga, A., and Stonebraker, M. Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms. *Proceedings of the VLDB Endowment 14*, 11 (2021).

[39] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. Improving mapreduce performance in heterogeneous environments. In *Osdi* (2008), vol. 8, p. 7.

[40] Zhang, X., Feng, D., Hua, Y., Chen, J., and Fu, M. A write-efficient and consistent hashing scheme for non-volatile memory. In *Proceedings of the 47th International Conference on Parallel Processing* (2018), pp. 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 $Y \sim \mathcal{N}\left(1, \frac{n}{m} f_d^{\alpha-1}\left(1 - \frac{1}{nf_d^\alpha}\right)\right)$. 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. 6.3, we have:

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

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

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

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