VIP Hashing - Adapting to Skew in Popularity of Data on the Fly

Aarati Kakaraparthy
University of Wisconsin, Madison
aaratik@cs.wisc.edu

Jignesh M. Patel
University of Wisconsin, Madison
jignesh@cs.wisc.edu

Brian P. Kroth
Microsoft Gray Systems Lab
bpkroth@microsoft.com

Kwanghyun Park
Microsoft Gray Systems Lab
kwpark@microsoft.com

ABSTRACT

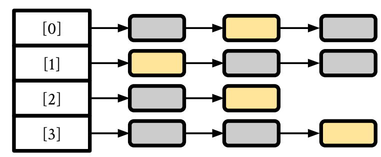
All data is not equally popular. Often, some portion of data is more frequently accessed than the rest, which causes a skew in popularity of the data items. Adapting to this skew can improve performance, and this topic has been studied extensively in the past for disk-based settings. In this work, we consider an in-memory data structure, namely *hash table*, and show how one can leverage the skew in popularity for higher performance.

Hashing is a low-latency operation, sensitive to the effects of caching and code complexity, among other factors. These factors make learning in-the-loop challenging as the overhead of performing additional operations can have significant impact on performance. In this paper, we propose VIP hashing, a hash table method that uses lightweight mechanisms for *learning* the skew in popularity and *adapting* the hash table layout on the fly. These mechanisms are non-blocking, i.e, the hash table is operational at all times. The overhead is controlled by *sensing* changes in the popularity distribution to *dynamically switch-on/off* the mechanisms as needed.

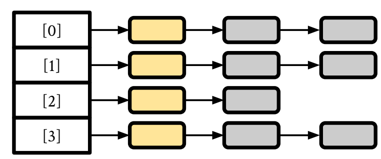
We ran extensive tests against a host of workloads generated by *Wiscer*, a homegrown benchmarking tool, and we find that VIP hashing improves performance in the presence of skew (22% increase in fetch operation throughput for a hash table with 1M keys under low skew) while adapting to insert and delete operations, and changing popularity distribution of keys on the fly. Our experiments on DuckDB show that VIP hashing reduces the end-to-end execution time of TPC-H query 9 by 20% under low skew.

PVLDB Reference Format:
Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. VIP Hashing - Adapting to Skew in Popularity of Data on the Fly. PVLDB, 15(10): 1978 - 1990, 2022.
doi:10.14778/3547305.3547306

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/aarati-K/wiscer.



(a) Default configuration: VIPs at random spots



(b) VIP configuration: VIPs at the front

Figure 1: Hash Table configurations with VIP keys (in yellow) at (a) random spots, vs. (b) at the front. The throughput of the hash table can be improved by giving VIPs more favorable spots at the front of the bucket.

1 INTRODUCTION

Hash tables are widely used data structures that provide a point lookup interface – mapping a key to a value. In database systems, they are used for in-memory indexing and for query processing operations such as hash joins and aggregation. The lightweight computation involved and the constant time lookup guarantees enable hash tables to achieve high throughput when processing point queries.

However, not all keys contribute equally to the performance, and requests are often skewed towards a smaller set of "hot" keys. In multiple studies involving production workloads, fetch requests have been observed to follow the power law [10, 15, 41] where the popularity of keys exponentially decays with the rank. The Very Important key-value Pairs (VIPs) are the keys with lower rank, as they constitute a larger portion of requests and have a greater impact on the throughput. It is possible to further improve the throughput obtained from the hash table by leveraging the skew in popularity, as we show in our work.

Fig. 1 shows the core motivation behind VIP Hashing – giving more favorable spots to more popular keys. In the VIP configuration (Fig. 1b), the keys are ordered in descending order of popularity and the VIPs are at the front, analogous to seating VIPs in the front row for an event. By placing the popular keys at the front, they can be accessed faster as they require fewer memory accesses and lesser computation (discussed in §4), which improves the overall throughput obtained from the hash table.

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547306

While attaining the VIP configuration is straightforward if the popularity of keys is known in advance (keys can be inserted in the right position in the chain according to their popularity), one might not have this information up front. Also, the popularity of the keys can change over time resulting in a different set of VIPs. Thus, more generally, one needs to learn the popularity of keys and adapt the configuration of the hash table on the fly.

It is important to note that learning requires additional computation and storage. In the case of disk-based data structures, the overhead of learning can be relatively small compared to the latency of accessing storage devices. However, hash tables are cache-sensitive data structures that perform lightweight computation, and adding overhead to hash tables can have significant impact on performance. This makes learning with hash tables notoriously challenging, as shown in §5.1 and prior work [34] as well. Thus, a key requirement for designing fully online learning mechanisms for hash tables is keeping the overhead in check compared to the gains.

Our contributions in this paper are as follows –

(1) ***Wiscer*** (§3) – We developed a benchmarking tool for measuring the performance of hash tables. Wiscer can be used to generate workloads with varying levels of skew in popularity, with different ratios of fetch, insert and delete operations, and shifting hot set of keys over time. To our knowledge, no existing benchmarking tool captures all of this behavior in one place.
(2) ***Roofline Analysis of the VIP configuration*** (§4) – We study the benefit of the VIP configuration (Fig. 1b) given prior knowledge of popularity. This analysis shows the maximum gain one can obtain from adapting to the skew (for a hash table with 10M keys at load factor 0.6, we observe a 57% increase in throughput in the best case), as well as the hardware trends resulting in performance gain for the VIP configuration.
(3) ***Learning on a budget*** (§5) – We developed lightweight mechanisms for *learning* the popularity distribution on the fly, *adapting* to the skew, *sensing* changes in the popularity distribution, and *dynamically switching on/off* the mechanisms to control the overhead. Put together, they give us the VIP Hashing method for adapting to the skew in popularity on the fly.
(4) ***Application to hash joins*** (§6.1) – We study the application of VIP hashing to PK-FK hash joins, and we obtain a 13-23% reduction in canonical join query execution time (for a cardinality ratio of 1:16 in the relations and a hash table with load factor of 1.4). We implemented VIP hashing in DuckDB [31] to speed up PK-FK hash joins in single-threaded mode, and we obtain a net reduction of 20% in end-to-end execution time of TPC-H query 9 [8] under low skew.
(5) ***Application to point queries*** (§6.2) – Another common use of hash tables is processing point queries. We test VIP hashing under a variety of workloads involving insert and delete operations, shifting popularity distribution of keys, different rates of shift, etc. A gain in throughput of 22% is obtained under low skew, while our choice of parameters ensures that the overhead of adapting on the fly is capped in the worst case.

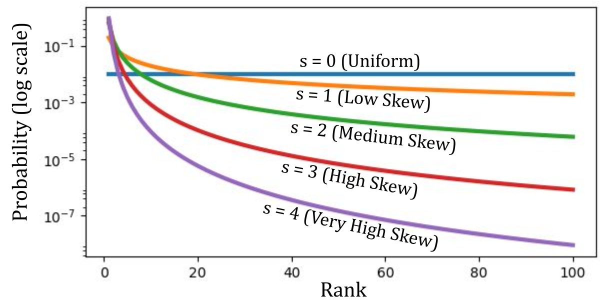


Figure 2: Popularity distribution of keys (number of keys $N = 100$) for different Zipfian skew factors s. Skew factor $s = 0$ corresponds to uniform popularity distribution, while $s = 1, 2, 3, 4$ simulates low, medium, high, and very high skew respectively.

Our experiments in §6 show that VIP hashing is a fully online non-blocking hash table method that adapts to the skew in popularity on the fly, while transparently capturing changes in the workload due to inserts, deletes, and shifting popularity distribution. We discuss related work in §7 and conclude in §8.

2 BACKGROUND

2.1 Hash Tables

A hash table [36] is an associative data structure that maps keys to values. In our work, we focus on *chained hashing* (hereafter referred to as hash table). A hash table (Fig. 1) uses a hash function to map each key to a unique index or bucket. Since more than one key can be mapped to the same bucket, the data structure resolves these collisions by maintaining a chain (linked list) of entries belonging to the bucket. The flexibility provided by this data structure for performing insert and delete operations, along with variable length keys and values make it a popular choice in many data systems [4, 7, 22, 23].

2.1.1 On Properly Configuring the Hash Table. In this paper, we focus on hashing of 8-byte integer keys and values, which is a well studied problem in past research [13, 32]. It is important to configure the hash table correctly to draw reliable conclusions, and there are two important factors to consider. The first is the choice of the hash function. In our work, we use *MurmurHash* [9], which is a strong hash function that provides good collision resistance in practice. The second critical aspect is the *load factor*, which is the ratio of keys to the number of buckets in the hash table. Higher load factors correspond to fewer buckets, which lead to longer chains on an average, whereas lower load factors require more buckets and consume more memory. Informed by parameter choices in popular open-source systems [4, 23, 33], we maintain a load factor between 0.5 and 1.5 to ensure that collisions are at an acceptable level while utilizing memory efficiently. Wherever applicable, we *rehash* the hash table to maintain this range of load factor. The number of buckets in the hash table are set to be a power of two, which is a common choice [23, 29, 33] that speeds up the computation of the hash function. If the load factor exceeds 1.5 (falls under 0.5), we double (half) the number of buckets in the hash table.

Table 1: Configuration options supported by *Wiscer*

Option	Description
zipf	The zipfian factor of the popularity distribution. $zipf = 0$ corresponds to uniform popularity.
initialSize	Initial number of keys in the hash table before running any operations.
operationCount	Total number of operations (fetch, inserts, etc.) to run on the hash table.
(fetch/insert/delete) Proportion	Proportion of operations that are fetch/insert/delete.
distShiftFreq	A shift in popularity distribution occurs after every *distShiftFreq* operations.
distShiftPrct	The popularity distribution shifts by *distShiftPrct%* every *distShiftFreq* operations.
storageEngine	Which storage engine to benchmark. Options are *ChainedHashing* (default), *VIPHashing*, and *none* (store workload to disk).
keyPattern	The pattern of keys to generate – *random* (default) or *sequential* (1 to *n*).
keyOrder	The popularity rank of keys relative to the insertion order. Options are *random* (default) and *sorted* (where keys are inserted in increasing order of popularity; a.k.a. latest).
randomSeed	The seed value (unsigned integer) to initialize the random number generator (default = 0). The random number generator is used to populate the hash table and generate the workload. Different seed values result in different instances of keys and the workload.

2.2 Some Probability Bounds and Theorems

Below we discuss some tools related to probabilistic random variables that we use in our work.

- **Zipfian distribution**: We use Zipfian distribution [43] to model varying levels of skew in fetch operations issued to keys in a hash table. Zipfian distribution has been adopted by multiple studies in the past [10, 13, 44] to statistically model skew in popularity, as it captures the power law [41] characteristics of workloads that are often observed in practice [10, 15].
- **Estimating mean and variance**: Let X be a random variable with mean μ and variance σ^2. Let $X_1,\ X_2, \dots X_n$ be n independent and identically distributed (i.i.d.) measurements of X. The estimated mean $\hat{\mu}$ and estimated variance $\hat{\sigma}^2$ can be evaluated as

$$\hat{\mu} = \frac{\sum_{i=1}^{n} X_i}{n}, \quad \hat{\sigma}^2 = \left(\frac{\sum_{i=1}^{n} X_i^2}{n-1} - \frac{\left(\sum_{i=1}^{n} X_i\right)^2}{n(n-1)} \right)$$

- **Gaussian tail bound confidence interval**: For a random variable X (refer above), the central limit theorem (CLT) [39] states that the error in estimated mean $(\hat{\mu}-\mu)$ is approximately Gaussian distributed $\mathcal{N}(0, \sigma^2/n)$. By applying the Gaussian pdf, a confidence interval can be obtained for the error $(\hat{\mu} - \mu)$ as follows

$$P(|\hat{\mu} - \mu| \le t) \ge \left(1 - exp\left(\frac{-nt^2}{2\sigma^2}\right)\right) = \frac{L}{100}$$

  Thus, we can at least be $L\%$ confident that the error $|\hat{\mu} - \mu|$ is less than t. Note that the confidence increases exponentially with n (number of samples X_i drawn). It is important to note that $(\hat{\mu}-\mu)$ is only *approximately* Gaussian, so the confidence interval obtained from applying Gaussian tail bound is a heuristic.

3 SKEWED WORKLOAD GENERATION WITH WISCER

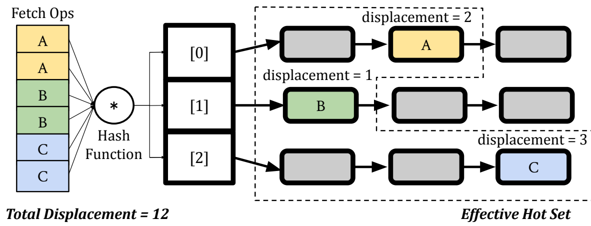
3.1 Overview

Wiscer is a benchmarking tool that we propose in this paper. Wiscer has multiple configuration options (Table 1) that can be used to generate workloads with different levels of skew, varying proportions of fetch, insert, delete operations, different rates of popularity shift, etc. Below are some key features of Wiscer:

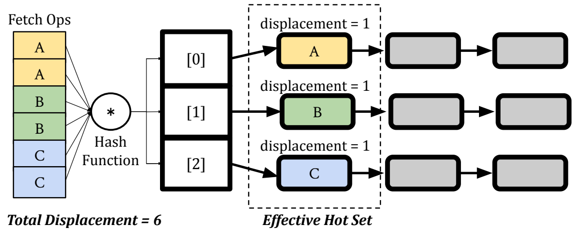
- **Level of skew**: Increasing levels of skew in the popularity distribution can be simulated by increasing the *zipf* factor. For instance, $zipf = 0$ and $zipf = 4$ correspond to uniform distribution and very high skew respectively (see Fig. 2).
- **Simulating popularity distribution shift**: The two related configuration options are *distShiftFreq* and *distShiftPrct*. After every *distShiftFreq* fetch operations, the topmost popular keys that constitute *distShiftPrct* of the requests are randomly replaced by less popular keys. This simulates a behavior where keys in the hot set become less popular after some time, which has also been observed in some real-world workloads [10].
- **Benchmarking hash table implementations**: Wiscer can optionally be used to compare different hash table implementations (option *StorageEngine*) to directly process the generated workloads without intermediate storage.
- **Fine-grained performance metrics using hardware counters**: Wiscer issues operations to the configured hash table in batches of one million requests at a time, and fine-grained metrics are collected per batch. Wiscer uses hardware counters provided by the Intel's Performance Monitoring Unit (PMU) [6] to get low-level performance metrics such as cache misses, number of cycles, retired instructions, etc.

3.2 Experimental Configuration

All experiments in this paper are run on a Cloudlab [21] machine with two 10-core Intel Xeon Silver 4114 CPUs with a peak frequency of 3.0GHz. The server is used exclusively for running Wiscer, and the benchmarking process is pinned to a single core to avoid any overhead of context switching. The CPU scaling governor of the core has been set to *performance*, thus fixing the frequency to 3.0GHz at all times. The CPU has an L3 cache of 13.75MB, and the server machine has 192GB of RAM. This CPU belongs to the Skylake Intel architecture family [2], and the PMU's hardware counters are programmed accordingly.



(a) Default configuration. A total displacement of 12 (=2×(2+1+3)) is required to process the fetch requests. The less popular keys in the path of popular keys need to accessed as well.



(b) VIP configuration. A total displacement of 6 (=2×(1+1+1)) is required to process the fetch requests. Only the popular keys are accessed.

Figure 3: Processing fetch requests in the Default vs the VIP configuration. Unpopular keys have been grayed out. The total *displacement* (number of keys accessed) is higher in the Default configuration requiring more pointer dereferences. Also, the effective hot set is larger, increasing the likelihood of cache misses relative to the VIP configuration.

4 ROOFLINE STUDY

In this section, we compare the performance of the Default and VIP configurations when the popularity of keys is static and known in advance. Since there is no overhead of learning involved in this case, this roofline study shows the maximum gain one can get from the VIP configuration for different levels of skew (§4.2) in popularity at different load factors (§4.3) of the hash table.

4.1 Default vs VIP Configuration

4.1.1 Motivation. Fig. 3 shows an example of processing fetch requests in the Default and the VIP configurations. A key parameter to note is the *displacement* encountered, which is the total number of keys that were accessed to process the fetch requests. Accessing a key requires dereferencing a pointer and some computation. The displacement encountered in the Default configuration is higher as the less popular keys in the path to VIPs need to be accessed when processing the fetch requests and effectively become part of the hot set. A larger hot set increases the likelihood of cache misses, and we observe this trend in our experiments described next.

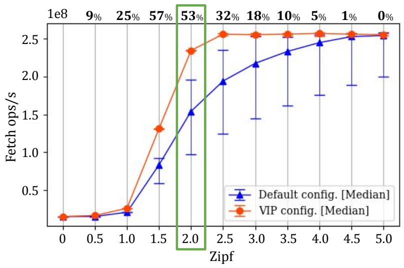
4.1.2 Generating the configurations using Wiscer. In the VIP configuration, keys in the hash table are arranged in descending order of popularity in the bucket chains (see Fig. 3b). We attain this configuration by running Wiscer with the default storage engine (*ChainedHashing*) and inserting keys in increasing order of popularity (*keyorder=sorted*, default is *random*). Insert operations on the hash table are performed at the front of the bucket chain (§2.1). Thus, when inserting keys in the sorted order, entries are automatically placed in decreasing order of popularity as more popular keys are inserted later and are ahead in the bucket chain. The Default configuration is generated using the default parameters of Wiscer.

4.2 Impact of Increasing Skew

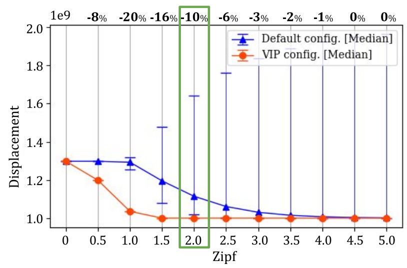
4.2.1 Workload. We compare the throughput of fetch operations in the Default and VIP configurations. We use Wiscer (Table 1) to generate fetch requests with increasing levels of skew ($zipf$ = 0 to 5 in steps of 0.5) which are issued to a hash table with 10 million keys at a load factor of 0.6 (= $10^7/2^{24}$). For each level of skew and hash table configuration, Wiscer is run with 10 distinct random seed values to populate the hash table and generate the workload. Each random seed results in a different arrangement of keys in the hash table. The popularity distribution is static, i.e., the rank of the keys remains the same throughout a run. One billion fetch requests are issued to the hash table for each random seed, and the data points reported in Fig. 4 are the median statistics over the 10 runs. We have run experiments on smaller (1M entries) and larger (100M entries) hash tables and found the trends to be similar.

4.2.2 Results. The results of this experiment are shown in Fig. 4. The gain in throughput ranges from 9%-57% depending upon the level of skew in popularity. Below we discuss our takeaways from the performance metrics measured using Wiscer:

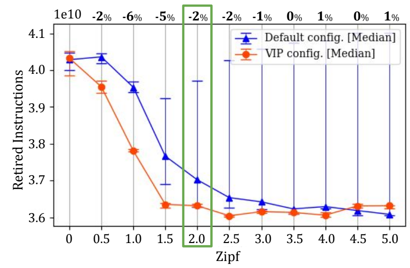
- **Throughput**: The gap in performance between the VIP and the Default configuration increases up to $zipf$ = 2 (medium skew), and gradually diminishes as the skew becomes very high ($zipf$ = 4.5 or 5). This behavior is correlated with the hot sets becoming smaller as the skew increases and becoming (L1/2/3) cache resident at different rates for the two configurations.
- **Displacement**: As expected, the displacement encountered in the VIP configuration is lower than the Default (see Fig. 3). For $zipf$ = 1.5 and up, the total displacement becomes close to 1B (for 1B fetch requests), indicating that popular keys are at the front of their chains (displacement = 1) in the VIP configuration. For the Default configuration, the median displacement approaches 1B at higher levels of skew ($zipf \geq 4$), but the variance is high as some random seeds can result in the popular keys placed further in the chains (however the likelihood of this happening is low as the load factor is not very high).
- **Instructions Executed**: The instructions executed are lower in the VIP configuration (up to 6% lower in the best case). The relative trend observed is similar to that of displacement, as the number of instructions executed is correlated with the number of keys accessed.
- **Cache misses**: The VIP configuration becomes L3 and L1 cache resident (at $zipf$ = 2 and 2.5 respectively) more quickly compared to the Default configuration (at $zipf$ = 3.0 and 4.5 respectively), which is expected as the hot set of the former is smaller than the latter (Fig. 3). At very high skew ($zipf$ = 4.5 and 5), both the configurations are L1 resident and correspondingly, we do not observe much difference in the throughput. This indicates that caching has a big impact on the performance of hash tables.



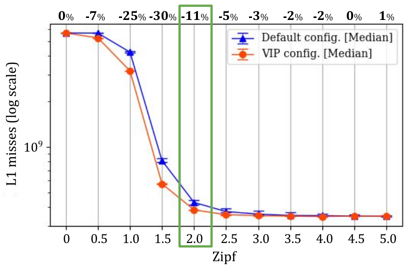
(a) Fetch operation throughput



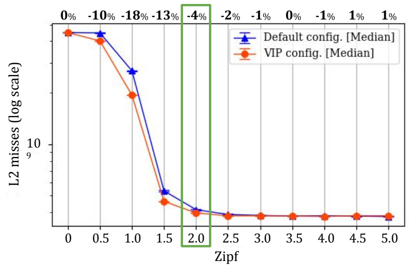
(b) Displacement



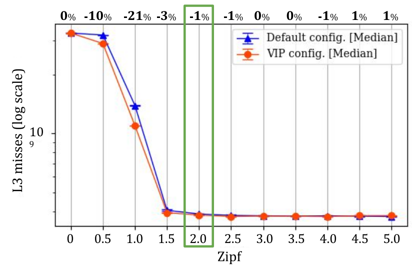
(c) Retired instructions



(d) L1 cache misses



(e) L2 cache misses



(f) L3 cache misses

Figure 4: Relative performance of the VIP vs the Default configurations as the skew in popularity increases. One billion fetch requests are issued to a hash table with 10M keys (load factor 0.6) for varying levels of skew from $zipf = 0$ to $zipf = 5$. Each reported data point is the median over 10 runs with different random seeds. Percentages indicated at the top of each plot is the difference between median metrics of the VIP vs the Default configuration. The gain in fetch operation throughput varies with skew, and we obtain 53% increase in throughput for medium skew ($zipf = 2.0$). Lesser number of cache misses and instructions executed contribute to the gain obtained from the VIP configuration. Further observations are discussed in §4.2.2.

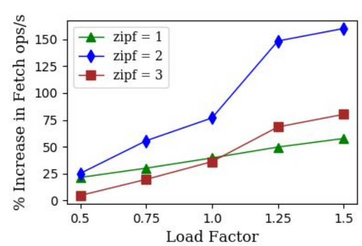
Overall, we note that since the hot set of the VIP configuration is smaller than the Default, we encounter lower cache misses at all levels of cache. This contributes to the gain in performance we obtain from the VIP configuration.

Another important observation we make is that the metric *displacement* indicates the goodness of the hash table configuration. The VIP configuration has lower displacement than the Default in all cases (the VIP configuration has the lowest possible displacement for a given data set, hash table size, hash function, and request skew; see §5.2.3). We use this metric in building the mechanisms for sensing and dynamically switching-on/off learning (§5.2.3).

4.3 Impact of Increasing the Load Factor

4.3.1 Workload. In this experiment, we increase the load factor while holding the size of the hash table constant. Similar to §4.2.1, we run one billion fetch operations on a hash table with 2^{24} buckets while varying the load factor from 0.5 to 1.5 in steps of 0.25 (this is achieved by increasing *initialSize* from 2^{23} to $3 \cdot 2^{23}$). Each configuration is run with 10 distinct random seeds and we compare the median statistics over the 10 runs.

4.3.2 Results. Fig. 5 shows the median gain obtained as we increase the load factor – we obtain 1.6x, 2.6x, and 1.8x higher throughput from the VIP configuration at low ($zipf = 1$), medium ($zipf = 2$), and high skew ($zipf = 3$) respectively at load factor 1.5. In all cases, the gain from the VIP configuration increases as the load factor increases, which is expected as the likelihood of collisions is higher when more keys are present in the hash table. We find that the performance metrics of the VIP configuration are mostly stable (refer to Table 2) indicating a stable hot set size, while the performance of the Default configuration becomes steadily worse as the effective hot set grows larger with the load factor.



Skew / Load Factor	Low skew (zipf = 1) ▲	Medium Skew (zipf = 2) ◆	High Skew (zipf =3) ■
0.5	+21%	+25%	+5%
0.75	+30%	+56%	+20%
1.0	+40%	+77%	+36%
1.25	+50%	+148%	+68%
1.5	+58%	+160%	+80%

Figure 5: Roofline gain in throughput from the VIP vs the Default configuration as the load factor increases. Keeping the number of buckets fixed at 2^{24}, we increase the load factor from 0.5 to 1.5. The performance gain obtained from the VIP configuration increases with the load factor, and can be as high as 160% (2.6x) for medium skew at load factor 1.5.

5 ADAPTING TO POPULARITY ON-THE-FLY

In this section, we first highlight the challenges of learning in-the-loop (§5.1) which motivated the lightweight mechanisms we built for VIP hashing. We then describe how we learn, adapt, sense, and dynamically control the overhead on the fly (§5.2-5.3).

Table 2: Relative Metrics of VIP vs Default configuration as we increase the load factor (*lf*) at $zipf = 2$. The trends for low and high skew are similar.

lf	Throughput (fetch ops/s)	Avg. Disp- -lacement	L3 Misses	L1 Misses
0.5	235M *vs* 188M *(+25%)*	1.0 *vs* 1.03 *(-3%)*	378M *vs* 385M *(-1.8%)*	380M *vs* 412M *(-8%)*
1	236M *vs* 134M *(+77%)*	1.0 *vs* 1.17 *(-15%)*	376M *vs* 387M *(-2.6%)*	380M *vs* 436M *(-13%)*
1.5	236M *vs* 90M *(+160%)*	1.0 *vs* 1.62 *(-38%)*	382M *vs* 392M *(-2.6%)*	382M *vs* 458M *(-17%)*

5.1 Learning In-the-Loop is Costly

Hash tables execute a tight loop of instructions – compute the hash function, access keys in the bucket, and perform required operations to process the request. Adding any amount of additional computation or storage to this loop can degrade performance considerably. To demonstrate this behavior, we conduct a simple experiment of adding a 1-byte requests counter per key in the hash table, such that the entries become 17 bytes long (8 byte key and value, and 1 byte counter).

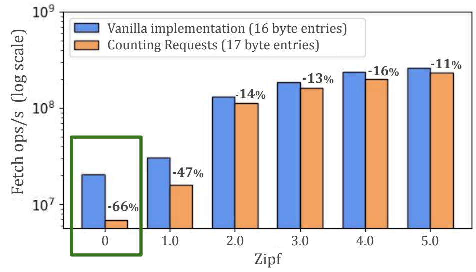
We use Wiscer to compare the performance of the vanilla implementation of hash table (16 byte entries) to the implementation with request counters (17 byte entries). We issue 500M fetch requests to a hash table with 1M entries (load factor $0.95 = 10^6/2^{20}$) for different levels of skew in the popularity distribution ($zipf = 0$ to 5 in steps of 1). The remaining configuration options of Wiscer are set to the defaults (refer to Table 1). Fig. 6 shows the relative performance of the two hash table implementations at different levels of skew in the workload. There is a significant loss in throughput ranging from 11-66% due to increase in cache misses and instructions executed.

Counting requests is a fundamental requirement for learning the popularity distribution. However, this experiment shows that even adding a small amount of additional memory can hurt performance significantly. Thus, the challenge here is to work with a restricted "budget" when learning in-the-loop, to balance the gains against the overhead of learning.

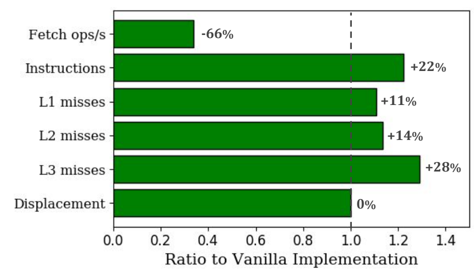
5.2 VIP Hashing

From §5.1, we know that using additional memory and computation can really hurt the performance of hash tables. In this section, we describe how VIP hashing overcomes these challenges by using lightweight mechanisms for learning and adapting to the popularity distribution (§5.2.2), while controlling the overhead by sensing and dynamically switching-on/off learning as necessary (§5.2.3). We first give an overview of VIP hashing (§5.2.1) followed by describing the mechanisms used in detail (§5.2.2-3).

5.2.1 Overview. Fig. 7 shows the VIP hashing method. At any given time, there are three possible modes that the hash table implementation can be in – *learn+adapt*, *sense*, and *default* (or vanilla). In the learn+adapt mode, the hash table learns the popularity distribution and rearranges keys to move closer to the VIP configuration. This mode is costly in terms of both computation and storage, and we control how much we run this mode by configuring the parameter N_L. The learn+adapt mode is run at the start, and subsequent triggers of this mode happen only if the popularity distribution changes, which is determined during the sense mode.



(a) Loss in performance when adding a 1-byte counter per key in the hash table. Both hash tables are identical (in Default configuration) except for the size of the entries (16 vs 17 bytes).



(b) Relative metrics for zipf = 0. Instructions executed and cache misses increase after adding the 1-byte counter.

Figure 6: The effect of adding a 1-byte requests counter per key in the hash table. 500M fetch operations are issued to a hash table with 1M keys at load factor 0.95. Performance can take a significant hit – we observe a 66% loss in fetch operation throughput at $zipf = 0$. This experiment demonstrates the sensitivity of hash tables to effects of caching and computation, which makes learning on the fly challenging.

The sense mode is triggered after the learn+adapt mode to measure some statistics (γ_B) that characterize the popularity distribution. These statistics require a total of 24 bytes of memory for the whole hash table (irrespective of the size) and a few additional arithmetic operations in the loop. Since the memory and computation footprint of this mode is low, it does not add much overhead to the execution. The sense mode is run for N_S requests at a time, and is triggered periodically (every N_D requests) to characterize the popularity distribution at the time (γ_C). Comparing the statistics (γ_B and γ_C) helps determine if the popularity distribution has changed, and informs the decision of whether to switch on learning.

The default mode is the vanilla implementation of chained hashing (§2.1) with 16 byte entries. There is no additional overhead of storage or computation. This mode is run most of the time ($N_D > N_L, N_S$), so the performance is close to the vanilla implementation of hash table in the worst case.

In the following sections, we discuss the mechanisms we use for the learn+adapt (§5.2.2) and sense (§5.2.3) modes. We discuss our choice of parameters (N_L, N_S, N_D, etc.) in §5.3, that allow us to balance the performance gains against the overhead of learning.

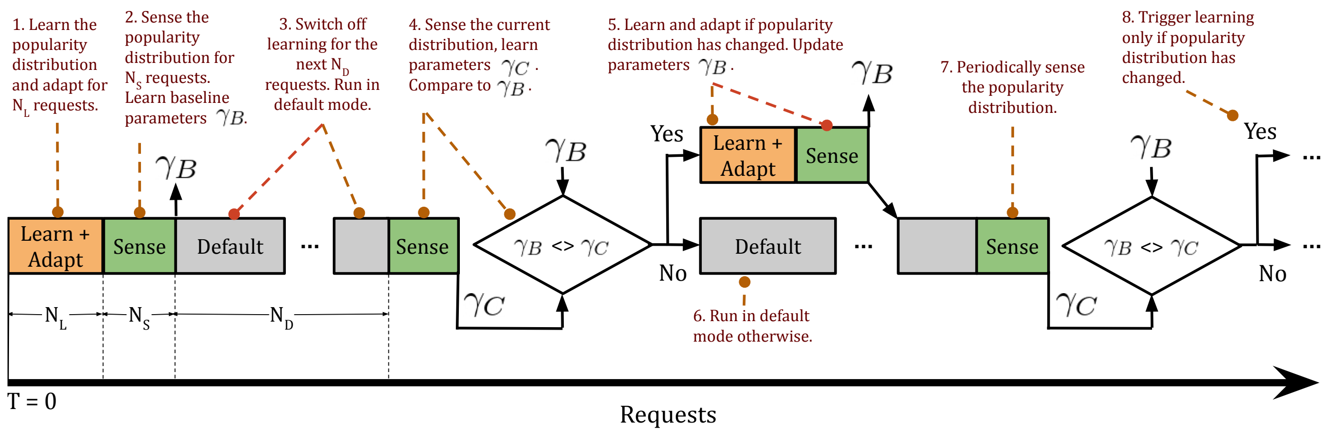


Figure 7: Overview of VIP Hashing. At any time, the hash table is in one of the three modes – *learn+adapt*, *sense*, or *default*. The amount of time spent on learn+adapt mode is controlled through the parameter N_L to cap the overhead of executing on the fly. The popularity distribution is sensed periodically and learning is triggered only when a change is detected.

Algorithm 1 Learning and Adapting on-the-fly

```
procedure FetchAdaptive(requests)
    ht ← getHashTable()
    /* Requests are counted in a separate data structure*/
    req_cnt_ht ← getRequestsCountingHashTable()
    for r in requests do
        hash ← murmurHash(r.key)
        ht_entry ← ht[hash]
        req_entry ← req_cnt_ht[hash]
        /* Keep track of entry with minimum requests */
        min_req_ht_entry = ht_entry
        min_req_entry = req_entry
        while ht_entry and ht_entry.key ≠ r.key do
            if req_entry.count < min_req_entry.count then
                min_req_ht_entry = ht_entry
                min_req_entry = req_entry
            ht_entry = ht_entry.next()
            req_entry = req_entry.next()
        if ht_entry == null then
            r.found = false
            continue
        r.found = true
        r.value = ht_entry.value
        req_entry.count = req_entry.count + 1
        if req_entry.count > min_req_entry.count then
            /* Swap this entry with the min requests entry */
            swap(ht_entry, min_req_ht_entry)
            swap(req_entry, min_req_entry)
    /* Reclaim cache space by clearing req_cnt_ht */
    clearCache(req_cnt_ht)
```

5.2.2 Learning & Adapting. Algorithm 1 describes how we learn the popularity distribution and adapt to the skew on the fly. The popularity of a key is estimated as the proportion of requests made to the key (§2.2). Thus, learning the popularity distribution requires counting requests, which we know is challenging from the experiments in §5.1.

To overcome the challenge of counting requests in-the-loop, we perform two optimizations. First, we count requests in a separate data structure that mimics the hash table in arrangement (for every entry in the hash table, there is a corresponding entry in the request counting hash table). Although this temporarily requires more memory (about 50-60% increase in memory usage depending on the load factor) than maintaining a counter per key in the hash table, the cost is incurred only during the learn+adapt mode. Second, at the end of the learn+adapt mode, we clear the requests counting hash table (*req_cnt_ht*) from the cache by issuing cache flush instructions (`_mm_clflushopt` on Intel CPUs [5]), which mitigates the cache pollution caused by the requests counting data structure used during the learn+adapt mode.

To attain the VIP configuration, we need to sort the keys in descending order of popularity in the bucket chains. Given that the proportion of requests made to a key is an estimate of popularity, we use Algorithm 1 to stochastically sort the keys in descending order of requests received on the fly. When performing a fetch operation, we keep track of the entry with minimum requests (*min_req_ht_entry*) encountered in the path to the entry being fetched. If the entry being fetched has received more requests, then it is swapped with the *min_req_ht_entry* and it moves forward in the chain. We propose the following theorem which is formally proved in Appendix A:

Theorem 5.1. *Let there be a bucket chain with n keys $K_1, K_2 \ldots K_n$ which have popularity $p_1 > p_2 \ldots > p_n > 0$. Let the keys be in a random order in the chain. Then, by applying Algorithm 1, the keys will converge to the sorted order of popularity as number of fetch requests $N \to \infty$.*

There are two noteworthy properties of Algorithm 1. First, the VIPs move to the front quickly, as they can skip over multiple entries in a single fetch request. This algorithm is, in essence, similar to selection sort as we are moving the entry with minimum requests to the end of the (sub-)chain being accessed. An alternative would be to compare only adjacent keys (bubble sort), which empirically requires more requests for a VIP to move to the front.

Second, the cost of swapping is amortized, as there is at most one swap performed per fetch operation. This approach is faster compared to performing a full sort on every request, or sorting at the end after counting requests for some time (we will have to access all the buckets in order to perform a full sort, which will block operation, incur cache misses, and pollute the cache).

5.2.3 Sensing & Dynamically Switch-on/off Learning. Algorithm 2 describes how we sense some key statistics of the popularity distribution, which enable us to dynamically switch-on learning only when the distribution has changed (Algorithm 3). While there are multiple ways to quantify the difference between two probability mass functions (*pmf*s) [37, 38, 42], we choose a lightweight statistic to compare distributions – *average displacement.* In §4.2.2, we saw that displacement encountered indicates the "goodness" of the hash table configuration. Every popularity distribution imposes a pmf over the displacement encountered on a request, which is a derived random variable. Formally stated:

Axiom 1. *Let $K_1, K_2, \ldots, K_N$ be N keys in the hash table with popularity $p_1, p_2, \ldots, p_N$ ($\sum p_i = 1$) at displacement $d_1, d_2, \ldots, d_N$ ($d_i \leq N$). Let D be the random variable of the displacement encountered on a successful fetch request. Then,*

$$P(D = d) = \sum_{i=1}^{N} p_i \cdot 1_{d_i = d}$$

i.e, the probability that displacement d is encountered on a fetch request is the probability that any of the keys with displacement d were fetched. The average displacement is calculated as

$$\mu_D = E[D] = \sum_{i=1}^{N} i \cdot P(D = i)$$

We make the following observation:

Axiom 2. *The VIP configuration minimizes E[D] over all possible arrangements of keys in the hash table for a fixed load factor, popularity distribution, and hash function.*

The VIP configuration orders keys by popularity, thus giving more "weight" to lower values of D which minimizes the average displacement. It is straightforward to see that for a given hash table configuration, two popularity distributions with different average displacement will not be identical (although the opposite is not true). Thus, a change in average displacement reflects a shift in the popularity distribution.

The parameters we learn from sensing are $\gamma = (\hat{\mu}_D, \hat{w}_D) = (u, w)$ (Algorithm 2), where $\hat{\mu}_D$ is the estimated average displacement, and $\hat{w}_D$ is the width of the confidence interval around $\hat{\mu}_D$ obtained using Gaussian tail bounds (§2.2). Average displacement is estimated as

$$\hat{\mu}_D = \frac{\sum_{i=1}^{N_S} D_i}{N_S}$$

which is the sample mean[1] of displacement encountered D_i ($1 \leq i \leq N_S$) over N_S fetch requests in the sense mode. Similarly, we also estimate sample variance $\hat{\sigma}_D^2$ (§2.2).

[1]Note that instead of sampling, we could also use the request counting data structure (*req_cnt_ht* in §5.2.2). However, this would incur cache misses and also pollute the cache affecting performance (§5.1).

Algorithm 2 Sensing

```
procedure FetchSensing(requests)
    ht ← getHashTable()
    /* Metrics to track */
    disp ← 0                          ▷ cumulative displacement
    disp_sq ← 0                       ▷ cumulative disp. square
    count ← 0                         ▷ number of requests
    c = 0.95                          ▷ confidence level of the interval
    for r in requests do
        hash ← murmurHash(r.key)
        ht_entry ← ht[hash]
        d ← 1
        while ht_entry and ht_entry→ key ≠ r.key do
            ht_entry = ht_entry.next()
            d = d + 1
        if ht_entry == null then
            r.found = false
            continue
        r.found = true
        r.value = ht_entry.value
        count = count + 1
        disp = disp + d
        disp_sq = disp_sq + d × d
    /* Estimating mean u, variance v, and C.I. width w*/
    u = disp/count
    v = disp_sq/(count − 1) − disp²/(count ∗ (count − 1))
    w = √(−2.v.log(1 − c)/count)      ▷ Gaussian tail bound
    γ = (u, w)
    return γ
```

Algorithm 3 Dynamically Switch-on/off Learning

```
procedure HasDistributionChanged(γ_B, γ_C)
    (u_B, w_B) = γ_B
    (u_C, w_C) = γ_C
    if |u_B − u_C| > (w_B + w_C) then
        return true
    else
        return false
```

We further characterize the pmf by building a confidence interval using Gaussian tail bounds (§2.2). The width ($\hat{w}_D$) of the interval at confidence level c ($c = 0.95$ in our experiments) is calculated as

$$\hat{w}_D = \sqrt{\frac{-2 \cdot \hat{\sigma}_D^2 \cdot (1 - c)}{N_S}}$$

Note that $\hat{\sigma}_D$ is estimated variance from a sample of N_S observations, and $(\hat{\mu}_D - \mu_D)$ only approximately Gaussian according to CLT (§2.2). Thus, the width $\hat{w}_D$ obtained by applying Gaussian tail bounds is a heuristic.

We switch-on learning (Algorithm 3) only if we detect a significant change in the average displacement. Given two sets of parameters $\gamma_B = (u_B, w_B)$ and $\gamma_C = (u_C, w_C)$ where u_B and u_C are estimated means, we check if the confidence intervals are disjoint. If so, then heuristically with a probability $c^2 = (0.95)^2 = 0.9$, we can be sure that the real means are not equal and the distributions have diverged. Thus, we detect changes in popularity distribution in a non-intrusive manner by computing lightweight statistics.

5.3 Parameters

The parameters N_L, N_S, and N_D determine how long the hash table runs in learn+adapt, sense, and default modes respectively. Our goal is to choose these parameters such that the gains of learning are balanced against the overhead.

Our choice of parameters is general, made using theoretical and empirical evidence that is independent of the popularity distribution. Thus, our techniques (§5.2) apply to any distribution with skew irrespective of its specific properties. Note that it is possible to further tune the parameters and the techniques with additional knowledge such as total number of requests, patterns in the workloads, family of distribution, etc.

5.3.1 Allocating the budget for learning – N_L vs N_D. Learning in-the-loop is costly. In our experiments, we find that the learn+adapt mode can be as much as $4x$ slower than the vanilla implementation in the worst case (under no skew for different hash table sizes from 1M to 100M keys). If a total of $(N_L + N_D)$ requests are issued, the loss in throughput due to the learn+adapt mode would be:

$$1 - \frac{T_{vanilla}}{T_{vip}} \leq \left(1 - \frac{N_D.t + N_L.t}{N_D.t + N_L.4.t}\right)$$

assuming that the vanilla implementation takes time t on an average to process each request. We cap the overhead of learning to at most 5% by choosing $N_D = 60 \cdot N_L$ in our experiments (i.e, learn+adapt mode is run for at most $1/61$ of the total requests). More generally, the cap on overhead is $(1 - 61/(60 + k))$, where k depends on the experimental configuration ($k = 4$ on our hardware). Thus, we cap the overhead of learning by fixing a budget for N_L/N_D.

5.3.2 Choosing N_L – how much to learn? The learn+adapt mode is run for N_L requests at a time. Our goal is to capture the popularity distribution while learning for a finite number of requests. From previous work [16], we know that it takes $\Theta(N)$ i.i.d. samples to learn a probability mass function over N items (with error $\epsilon = 1$ in KL divergence compared to the true pmf). When the cardinality of the hash table is not known/can vary, we choose $N_L = 1.5 \cdot (htsize)$, i.e, 1.5 times the number of buckets in the hash table. Since we maintain a load factor of at most 1.5 at all times, the number of keys in the hash table $N \leq 1.5 \cdot htsize$, which satisfies our requirements.

5.3.3 Parameters for sensing – N_s and c. We sense the distribution for N_S requests at a time to estimate the average displacement $\hat{\mu}_D$ and build an interval with confidence c. Since the load factor is low and the longest chain length is likely to be low as well (except in pathological cases), we have found that choosing N_S to be a large number (1000) has been sufficient in our experiments. We build a $c = 95\%$ confidence interval that gives us a heuristic probability of $c^2 = (0.95)^2 = 0.9$ when we detect a shift in the popularity distribution. By increasing (decreasing) the confidence level, we can be less (more) sensitive to changes in popularity.

6 APPLICATIONS

6.1 PK-FK Hash Joins

Hash tables are frequently used in database systems for processing join queries. In this section, we describe how VIP hashing can improve the performance of primary key-foreign key (PK-FK) hash joins in the presence of skew.

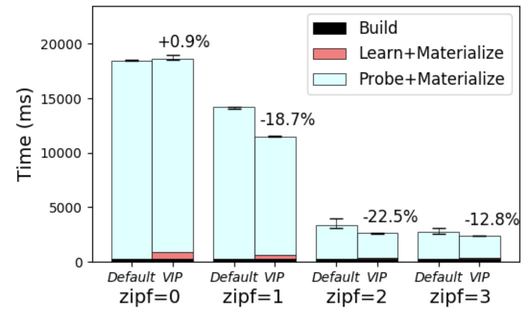


Figure 8: Performance of PK-FK canonical hash join on tables R and S ($|R| : |S| = 1 : 16$) using the default and VIP hash table implementations. For medium skew, we observe a 22.5% reduction in median (over 10 random seeds) total execution time.

Table 3: Relative metrics for default and VIP hash join at $zipf = 2$, $|R| : |S| = 1 : 16$.

Metric	*Default*	*VIP*	*Diff*
Time	3.4s	2.6s	-22.5%
Avg. Displacement	1.23	1.0003	-18.7%
L3 Misses	75.5M	75.3M	-0.3%
L2 Misses	127.9M	124.6M	-2.6%
L1 Misses	161.2M	155.7M	-3.4%
Instructions	8.5B	8.2B	-3.5%

6.1.1 Experimental Setup. Motivated by past research [11, 13, 27], we consider the canonical PK-FK join query on tables R and S ($|R| \leq |S|$) with 8-byte integer attributes (16-byte tuples). Skew can arise in PK-FK relations [11, 13] when some keys occur more frequently than others in the outer relation S. We use Wiscer to instantiate R and S using the sequential key pattern for primary keys in R, and varying the level of skew in the outer relation S from uniform ($zipf = 0$) to high ($zipf = 3$) for 10 distinct random seeds. We compare the performance of the canonical hash join algorithm [11, 27] implemented using the default and VIP hash tables, while materializing pointers to output tuple pairs. We assume that the tuples in S are i.i.d, i.e, the popularity distribution is static. We explore effects of dynamic popularity distribution in §6.2.

6.1.2 Default vs VIP Hash Join. Fig. 8 shows the relative execution time of the default vs VIP hash join implementations. The cardinalities of R and S are 12M and 192M respectively ($|R| : |S| = 1 : 16$) [11, 13], and the load factor is 1.4 ($= 12 \cdot 10^6/2^{23}$). For medium skew in the outer relation, the average displacement encountered by the default hash join implementation is 1.23 (Table 3)[2].

For the case of canonical hash join query, the learning budget of the VIP hash table implementation can be calculated in advance while maintaining $N_L : N_D = 1 : 60$ (§5.3) since we almost always know the cardinalities of the relations from system catalogs. Learning is triggered at the beginning of the probe phase with a

[2]Note that the average displacement is low for the default configuration in this case, since the keys are sequential. Holding the load factor constant, randomly generated keys result in a median (over 10 random seeds) average displacement of 1.48.

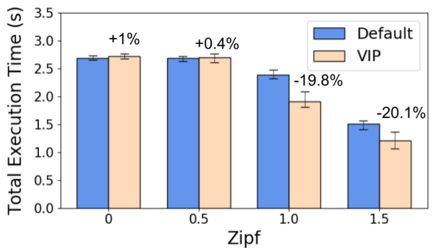


Figure 9: Execution time of TPC-H query 9 (scale factor = 1) on DuckDB. VIP hashing speeds up PK-FK hash join probes, and results in 20% reduction in median (over 10 random seeds) end-to-end query execution time at $zipf = 1$ and $zipf = 1.5$.

budget of $N_L = min(|R|, \frac{|S|}{61}) = \frac{16 \cdot |R|}{61} = 0.26 \cdot |R|$ lookups from the outer relation. Learning takes about 3% of the total execution time, ranging from 70-600ms depending on the level of skew. Note that the average displacement of the VIP hash join implementation is very close to 1 (Table 3) indicating that the learning mechanism efficiently captures the popularity distribution, and reduces cache misses and instructions executed.

To show the impact of varying the learning budget, we repeated the experiment for lower and higher cardinality ratios. For a ratio of $1:4$, we have a learning budget of $\frac{4 \cdot |R|}{61} = 0.07 \cdot |R|$ requests and the overall reduction in execution time is 18.6%. On the other hand, a cardinality ratio of $1:64$ allows a learning budget of $|R| = min(|R|, \frac{64 \cdot |R|}{61})$ and results in 25.8% reduction in execution time. Thus, the available learning budget impacts the gain in performance.

6.1.3 Application to Skewed TPC-H. We focus our attention on TPC-H query 9 [8], which is the most expensive TPC-H query involving multiple PK-FK joins. We implemented VIP hashing in DuckDB [31], an in-memory vectorized DBMS, to speed up PK-FK hash joins in single-threaded mode. Fig. 9 shows the median execution time of VIP hash join relative to the default, tested on skewed TPC-H data [26] at varying levels of skew for 10 different random seeds. VIP hash join reduces the end-to-end query execution time by 20% at $zipf = 1$ and $zipf = 1.5$, while the increase in execution time at lower skew is negligible. The remaining TPC-H queries spend < 1% of the total execution time in skewed PK-FK hash joins, and consequently the impact of VIP hashing is negligible.

6.2 Point Queries

Another common use of hash tables is for in-memory indexing in database systems [1, 22] and in key-value stores [4, 23] for processing point queries. In this section, we evaluate VIP hashing against a range of workloads generated using Wiscer that highlight the robustness of our techniques for learning in-the-loop under different conditions. In all the experiments, we assume no prior knowledge of the characteristics of the request distribution. The first two workloads (§6.2.1-§6.2.2) involve fetch operations, and the last two (§6.2.3-§6.2.4) perform insert and delete operations.

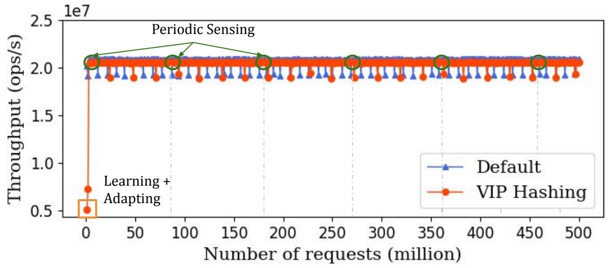
We run these workloads on a hash table with 1M entries (load factor $0.95 = 10^6/2^{20}$) in the Default configuration. Each of these workloads issue 500M operations to the hash table at low skew ($zipf = 1$) unless specified otherwise. The performance gain under medium skew ($zipf = 1.5$) is higher, and those results are included in the extended version of the paper [26]. The remaining configuration options of Wiscer are set to the defaults (Table 1). We compare the performance of VIP hashing to the default hash table in Fig. 10.

6.2.1 Static Popularity. In this workload, the popularity of keys in the hash table remains the same throughout. For the case of uniform popularity distribution ($zipf = 0$), the loss in throughput is 2% (Fig. 10a) which is within our budget of 5% (§5.3.1), whereas for low skew ($zipf = 1$), we obtain a net gain of 22% (Fig. 10b). Since the popularity distribution is static, the learn+adapt mode is triggered only at the start of the experiment for $1.5 \cdot htsize$ requests. The periodic runs of the sense mode do not detect a change in popularity and the learn+adapt mode is not triggered again, thus minimizing the overhead of learning.

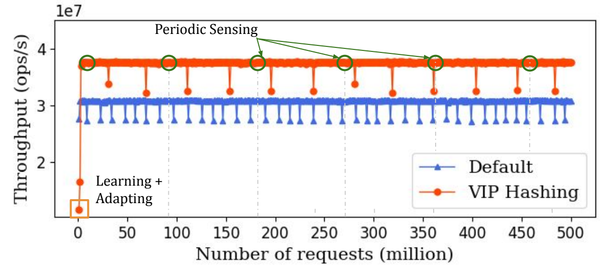
6.2.2 Popularity Churn. In this workload, the popularity distribution shifts over time – we simulate a medium (25%) and high (50%) rate of shift every 100M (about 3s) and 10M (< 1s) requests respectively. Fig. 10c shows the behavior of VIP hashing under medium churn – 3 out of the 4 times when the popularity shifted, there was a substantial change in average displacement (accompanied by a decrease in performance) which was detected in the sense mode, and learning was triggered only when necessary. For the case of high churn (Fig. 10d), popularity shift occurs 50 times during the experiment, and every run of the sense mode detects a change in distribution and learning is triggered. We obtain a net increase of 19% and 12% in throughput for the case of medium and high churn respectively. Thus, VIP hashing is able to sense changes in distribution and re-learn on the fly.

6.2.3 Steady State. Next, we create a workload with 98% fetch, 1% insert, and 1% delete requests. The cardinality of the hash table doesn't change substantially during the experiment, as the number of insert and delete operations are balanced. The keys are inserted (deleted) in random positions of the popularity order. We observe that as new keys (which are less popular with high probability) are inserted at the front of the chains, the hash table arrangement steadily becomes worse and the performance of VIP hashing approaches the default. A change in average displacement is sensed every time and learning is triggered, which bounces back the performance of VIP hashing. We observe a 5.4% gain in throughput.

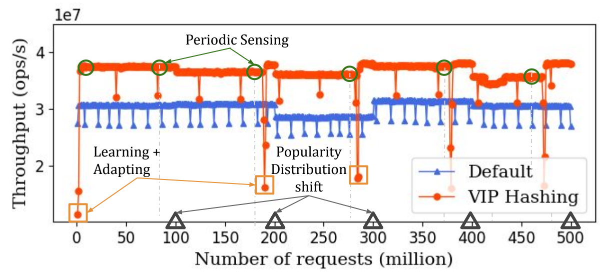
6.2.4 Read Mostly. In this workload, we issue 98% fetch requests and 2% insert requests. New keys are inserted in arbitrary positions in the popularity order. Similar to §6.2.3, we observe that the performance steadily becomes worse as new keys are inserted at the front of the bucket chains. Inserting new keys increase the load factor, which degrades the throughput of the default implementation as well. Rehashing is triggered when the load factor exceeds 1.5 (happens every $75 \cdot htsize$ requests), which bounces back the performance for both the default and VIP hashing implementations. The periodicity at which sensing is triggered (every $90 \cdot htsize$ requests) increases every time rehashing is performed, as we update the parameters N_S and N_L according to the size of the hash table ($htsize$). Given that the change in the distribution is substantial, every run of the sense mode detects a change in popularity and triggers learning. Overall, we obtain a gain of 1% in throughput.



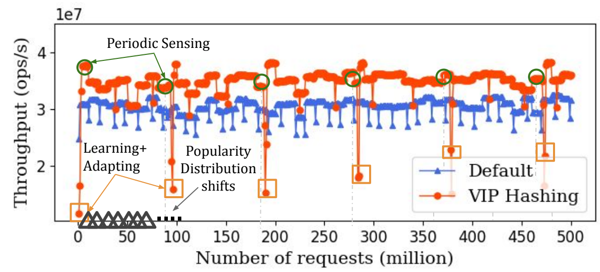
(a) Static popularity (§6.2.1) with $zipf = 0$ (uniform distribution). Since there is no skew in popularity, no performance gain can be obtained from VIP hashing. Learning adds overhead to VIP hashing ($4x$ slower), and is only triggered at the start for $(1.5 \cdot 2^{20})$ requests (0.3s). Subsequent sensing of the popularity distribution does not detect any change, and learning is not triggered. Total loss in throughput is 1.9%, which is within our allocated budget.



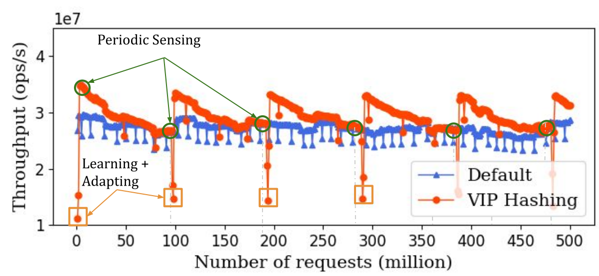
(b) Static popularity (§6.2.1) with $zipf = 1$ (low skew). Learning is only triggered at the start and is 3x slower than the default (0.13s vs 0.05s respectively). Sensing does not detect any changes to the popularity distribution, so learning is not triggered again. The overhead of learning is offset by the gain in performance from the VIP configuration. We observe an overall increase in throughput of 21.8%.



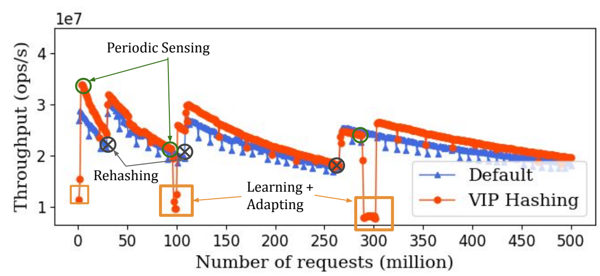
(c) Medium churn rate (§6.2.2) with $zipf = 1$. Popularity distribution shifts every 100M requests by 25% (top 21 out of 1M keys are replaced by less popular keys). Distribution shift increases average displacement and can reduce performance (notice drop in performance of VIP hashing at 200M requests). Sensing triggers learning whenever it detects a significant increase in average displacement. Throughput increases by 18.9% overall.



(d) High churn rate (§6.2.2) with $zipf = 1$. Popularity distribution shifts every 10M requests by 50% (top 750 out of 1M keys are replaced by less popular keys). The benefit of learning dimishes as the popularity order becomes shuffled. Periodic sensing triggers learning every time, as frequent distribution shifts cause significant change in average displacement. Overall, 11.8% increase in throughput is observed.



(e) Steady state (§6.2.3) with $zipf = 1$. 98% fetch requests, 1% insert requests, and 1% delete requests. With new keys being inserted (at the front of the buckets) and existing keys being deleted, the hash table arrangement steadily becomes worse. Learning is triggered periodically which bounces back the performance. An overall gain of 5.4% is observed.



(f) Ready mostly workload (§6.2.4) with $zipf = 1$. We issue 98% fetch requests and 2% insert requests. Rehashing is triggered when the load factor reaches 1.5, which happens every $75 \cdot htsize$ requests. When rehashing occurs, we double the periodicity of sensing (N_S) and the duration of learning (N_L), i.e., learning is triggered less frequently for longer duration. We observe a gain of 1% in throughput.

Figure 10: Comparing the performance of VIP hashing to the default (vanilla) implementation of hash table when subjected to identical workloads. Requests are issued in batches of 1M to a hash table with 1M keys (load factor 0.95 = $10^6/2^{20}$) at the start in Default configuration. Workload 10a has uniform popularity distribution ($zipf = 0$) and workloads 10b-10f are run with low skew ($zipf = 1$). The loss in throughput[2] is 2% in the worst case, while we obtain a gain in performance ranging from 1% to 22% depending on the workload.

[2] The small periodic dips in throughput in both VIP hashing and the default implementation are due to monitoring activity performed by the Cloudlab environment [21] and are unrelated to our workloads.

7 RELATED WORK

Hash tables are well studied data structures in literature. Two major categories of hash tables are chained hashing [36] where collisions are resolved by chaining (§2.1), and open addressing [40] where collisions are resolved by searching for alternate positions in an array. Richter et al. [32] study different hash table implementations spanning both the categories, hash functions, workload patterns, etc. while highlighting the variability in the performance of hash tables based on a host of factors. Similar to our work, they consider the problem of hashing 8-byte integer keys and values.

Multiple open source hash tables [3, 12, 35] use both categories of implementations. For instance, Google's flat hash table [12] uses open addressing, while the bytell (byte linked list) hash table [35] uses chaining to resolve collisions. When it comes to data systems, DBMS such as SQLite3 [7] and PostgreSQL [33], as well as key-value stores such as Redis [29] and Memcached [23] use data structures that involve chaining of entries. Thus, we find that chained hash tables are a popular choice commonly used in practice.

Skew in popularity is a well studied phenomenon. Multiple studies involving production workloads have found fetch requests to follow a power-law behavior [10, 15], which is often captured using the zipfian distribution [13, 20, 44]. For instance, the request distribution in the core workloads of YCSB [19] is zipfian by default. Alongside skew in popularity, previous work [10] also discusses effects such as churn in popular keys in real world workloads. This is a key feature captured by Wiscer (§3), which is not present in any of the existing workload generators to the best of our knowledge.

Broadly speaking, caching algorithms such as LRU-k [30] and MRU [18] attempt to capture the current popularity distribution. Key-value stores designed for disk-based settings, such as Anna [44] and Faster [17] incorporate techniques to keep hot data in memory for better performance. Recent work by Herodotou et al. [25] uses machine learning (ML) to automatically move data between different storage tiers in clusters. A recurring trend to note here is that the complexity of these schemes depend on the "budget" available, ranging from simple LRU approach used even in processor caches, to a more complex approach involving ML in large-scale clusters.

The budget available for learning with hash tables is extremely limited (Fig. 6). In the seminal paper on learned indexes [28], the authors propose learning a hash function from the keys such that collisions can be avoided altogether. However, recent work on learned hash functions [34] shows that this approach encounters two major limitations – cache sensitivity, and model complexity. While larger models are necessary to accurately capture arbitrary key distributions, the computation times become prohibitively high ($50x$ higher [34]) due to increased cache misses from accessing the model parameters. The high cache sensitivity and low latency requirements of hash tables preclude the use of costly ML techniques.

A noteworthy aspect of the VIP hashing method is that learning is performed online, i.e., the hash table does not pause operation at any time. In contrast, recent work [24, 34] involves learning from the data offline before populating the hash table. Adapting to changing key distributions remains a challenge with these approaches, as their fallback mechanism is reverting to the default hash table implementation [24] or relearning [28, 34], both of which require costly rehashing that pauses execution.

8 CONCLUSIONS & FUTURE WORK

The sensitivity of hash tables to effects of caching makes learning on the fly very challenging (§5.1, [34]). In this paper, we describe the VIP hashing method for adapting to the skew in popularity of data on the fly. VIP hashing is comprised of four lightweight mechanisms – *learning*, *adapting*, *sensing*, and *dynamically switching-on/off* learning – that execute in a fully online fashion. Our choice of parameters (§5.3) carefully balances the gains against the overhead of executing online. We evaluate VIP hashing using an extensive set of workloads (Fig. 8, 10) that demonstrate the ability to learn on the fly, while being robust to changes caused by insert/delete operations, shifting distributions, etc. In our experiments, the gain in performance obtained was 22% in the best case.

Possible future work could involve studying other low latency data structures such as bloom filters [14], to see how cache locality can be improved by adapting to the data. Learning tasks involving such cache sensitive data structures will necessitate controlling the overhead, potentially using our approach of budgeted learning and non-intrusive sensing.

ACKNOWLEDGMENTS

This research was supported in part by a grant from the Microsoft Jim Gray Systems Lab, by the National Science Foundation under grant OAC-1835446, and by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program.

A PROOF OF THEOREM 1

Theorem 1 (§5.2.2) states that given keys $K_1,\ K_2,\ \ldots, K_n$ in a bucket with probability $p_1 > p_2 > .. > p_n$, such that the keys are in a random order initially. Then by applying Algorithm 1, the keys will converge to the sorted order of popularity as the number of fetch requests $N \rightarrow \infty$.

From the frequentist definition of probability, we can be sure that a more popular key will receive more requests compared to a less popular key as $N \rightarrow \infty$. This will hold pairwise for all the keys $K_1,\ K_2,\ \ldots, K_n$ in the bucket, which motivates the following claim.

Lemma A.1. *Let $\{K_i\}$ be keys in a bucket with probability $\{p_i\}$, $i \in [N]$. Let K_1 be the most popular key in the bucket, i.e., $p_1 > p_j\ \forall j \in \{2, .., N\}$. Let the initial order of keys be random. Then, by running Algorithm 1, K_1 will be at the front of the chain as $N \rightarrow \infty$.*

Proof. Suppose K_1 is at displacement $d > 1$ and has received n requests. Let there be keys $K'_1,\ ..,\ K'_{d-1}$ in front of K_1 that have received requests $n_1,\ ..,\ n_{d-1}$ respectively. From Lemma 2, we have

$$\lim_{N \rightarrow \infty} n > n_i,\ \forall\ i \in [(d-1)]$$

Thus, K_1 would have received more requests than all the keys in front of it as $N \rightarrow \infty$. From Algorithm 1, on the last request that K_1 received, it should have been swapped with a key with lower number of requests ahead of it. This contradicts our assumption that K_1 is at position $d > 1$. □

Thus, the most popular key in the chain will be in the front as number of requests approaches infinity. By recursively applying Lemma 3 to the remaining keys in the bucket, we can prove that the keys will be in the sorted order of popularity as $N \rightarrow \infty$.

REFERENCES

[1] MariaDB 2013. *MariaDB Storage Index Types*. MariaDB. Retrieved June 23, 2022 from https://mariadb.com/kb/en/storage-engine-index-types/

[2] Intel Corporation 2017. *Intel Xeon Silver 4114 processor*. Intel Corporation. Retrieved June 23, 2022 from https://intel.ly/3fDidSb

[3] Intel Corporation 2020. *Intel TBB hash map*. Intel Corporation. Retrieved June 23, 2022 from https://oneapi-src.github.io/oneTBB/main/tbb_userguide/concurrent_hash_map.html

[4] Redis Ltd. 2022. *Data types in Redis*. Redis Ltd. Retrieved June 23, 2022 from https://redis.io/topics/data-types

[5] Intel Corporation 2022. *Intel Intrinsics*. Intel Corporation. Retrieved April 22, 2022 from https://intel.ly/3nxA416

[6] Intel Corporation 2022. *Intel performance monitoring events*. Intel Corporation. Retrieved June 2, 2022 from https://perfmon-events.intel.com/

[7] SQLite 2022. *SQLite hash table implementation*. SQLite. Retrieved June 23, 2022 from https://sqlite.org/src/file/src/hash.c

[8] TPC 2022. *TPC-H Benchmark (Version 3)*. TPC. Retrieved June 23, 2022 from http://www.tpc.org/tpch/

[9] Austin Appleby. 2016. *MurmurHash3*. Retrieved June 23, 2022 from https://github.com/aappleby/smhasher/wiki/MurmurHash3

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. *Sigmetrics Performance Evaluation Review - SIGMETRICS* (2012). https://doi.org/10.1145/2318857.2254766

[11] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Ozsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering*. https://doi.org/10.1109/ICDE.2013.6544839

[12] Sam Benzaquen, Alkis Evlogimenos, Matt Kulukundis, and Roman Perepelitsa. 2018. *Swiss Tables and absl::Hash*. Google. Retrieved June 23, 2022 from https://abseil.io/blog/20180927-swisstables

[13] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/1989323.1989328

[14] B. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* (1970). https://doi.org/10.1145/362686.362692

[15] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99*. https://doi.org/10.1109/INFCOM.1999.749260

[16] Clément L. Canonne. 2020. A short note on learning discrete distributions. *arXiv: Statistics Theory* (2020). https://doi.org/10.48550/ARXIV.2002.11457

[17] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *2018 ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/3183713.3196898

[18] H. Chou and D. DeWitt. 2005. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica* (2005). https://doi.org/10.5555/1286760.1286772

[19] Brian F. Cooper. 2010. *YCSB Core Workloads*. Retrieved November 29, 2020 from https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads

[20] 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 (SoCC 2010)*. https://doi.org/10.1145/1807128.1807152

[21] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. https://doi.org/10.5555/3358807.3358809

[22] Erik Frøseth. 2019. *Hash join in MySQL 8*. MySQL. Retrieved November 13, 2019 from https://mysqlserverteam.com/hash-join-in-mysql-8

[23] Holmes He. 2021. *Understanding the Memcached source code*. Retrieved January 1, 2021 from https://holmeshe.me/understanding-memcached-source-code-V

[24] Brian Hentschel, Utku Sirin, and Stratos Idreos. 2022. Entropy-Learned Hashing: Constant Time Hashing with Controllable Uniformity. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. https://doi.org/10.1145/3514221.3517894

[25] Herodotos Herodotou and Elena Kakoulli. 2019. Automating distributed tiered storage management in cluster computing. *Proc. of the VLDB Endowment* (2019). https://doi.org/10.14778/3357377.3357381

[26] Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. 2022. VIP Hashing – Adapting to Skew in Popularity of Data on the Fly (extended version). https://doi.org/10.48550/ARXIV.2206.12380

[27] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 2009. Application of hash to data base machine and its architecture. *New Generation Computing* (2009). https://doi.org/10.1007/BF03037022

[28] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. https://doi.org/10.1145/3183713.3196909

[29] Kousik Nath. 2017. *A little internal on Redis hash table implementation*. Retrieved June 23, 2022 from https://bit.ly/3pfVvTm

[30] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 International Conference on Management of Data (SIGMOD '93)*. https://doi.org/10.1145/170035.170081

[31] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. https://doi.org/10.1145/3299869.3320212

[32] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proceedings of the VLDB Endowment* (2015). https://doi.org/10.14778/2850583.2850585

[33] Egor Rogov. 2019. *Indexes in PostgreSQL*. Retrieved March 19, 2019 from https://bit.ly/3c7L52A

[34] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. 2021. When Are Learned Models Better Than Hash Functions? *CoRR* abs/2107.01464 (2021). https://arxiv.org/abs/2107.01464

[35] Malte Skarupke. 2018. *Bytell hash map*. Retrieved June 16, 2018 from https://bit.ly/3fB8NX6

[36] Wikipedia. 2022. *Hash table*. Wikimedia Foundation, Ltd. Retrieved June 23, 2022 from https://en.wikipedia.org/wiki/Hash_table

[37] Wikipedia. 2022. *Hellinger's distance*. Wikimedia Foundation, Inc. Retrieved May 24, 2022 from https://en.wikipedia.org/wiki/Hellinger_distance

[38] Wikipedia. 2022. *Kullback-Leibler divergence*. Wikimedia Foundation, Ltd. Retrieved June 19, 2022 from https://en.wikipedia.org/wiki/Kullback-Leibler_divergence

[39] Wikipedia. 2022. *Lindeberg-Levy CLT*. Wikimedia Foundation, Inc. Retrieved June 10, 2022 from https://en.wikipedia.org/wiki/Central_limit_theorem#Classical_CLT

[40] Wikipedia. 2022. *Open addressing*. Wikimedia Foundation, Ltd. Retrieved April 17, 2022 from https://en.wikipedia.org/wiki/Open_addressing

[41] Wikipedia. 2022. *Power Law*. Wikimedia Foundation, Ltd. Retrieved June 3, 2022 from https://en.wikipedia.org/wiki/Power_law

[42] Wikipedia. 2022. *Z-test*. Wikimedia Foundation, Inc. Retrieved April 18, 2022 from https://en.wikipedia.org/wiki/Z-test

[43] Wikipedia. 2022. *Zipf's law*. Wikimedia Foundation, Ltd. Retrieved June 19, 2022 from https://en.wikipedia.org/wiki/Zipf's_law

[44] Chenggang Wu, Vikram Sreekanti, and Joseph Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment* (2019). https://doi.org/10.14778/3311880.3311881