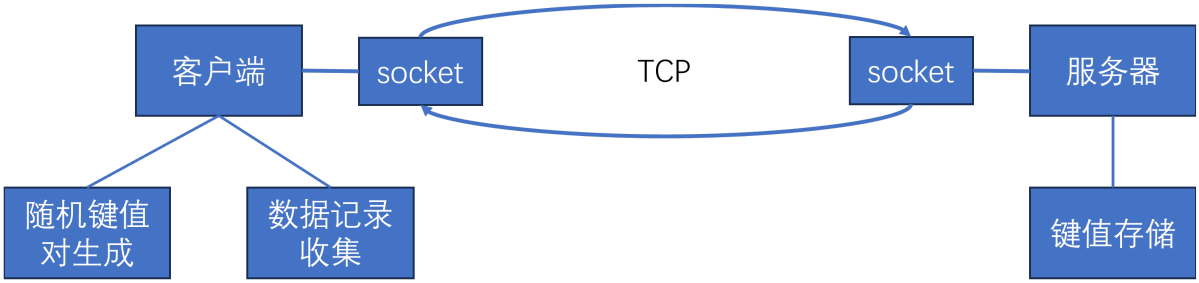
Extensible Hash Table based on Hot Ring

系统设计

本项目系统主要分为客户端和服务器两个部分。客户端包括随机生成器和数据收集器两部分，服务器包括键值存储，两者之间通过socket通信。



随机键值生成。本次实验设置键值对总共有5万个，其中热点key占比X%。热点key生成的概率为80%，非热点key生成的概率为20%。

数据记录收集。作为一个线程启动，每隔60s唤醒一次，记录负载数量。总共唤醒5次。

socket。客户端和服务器之间使用socket进行通信，协议为TCP。

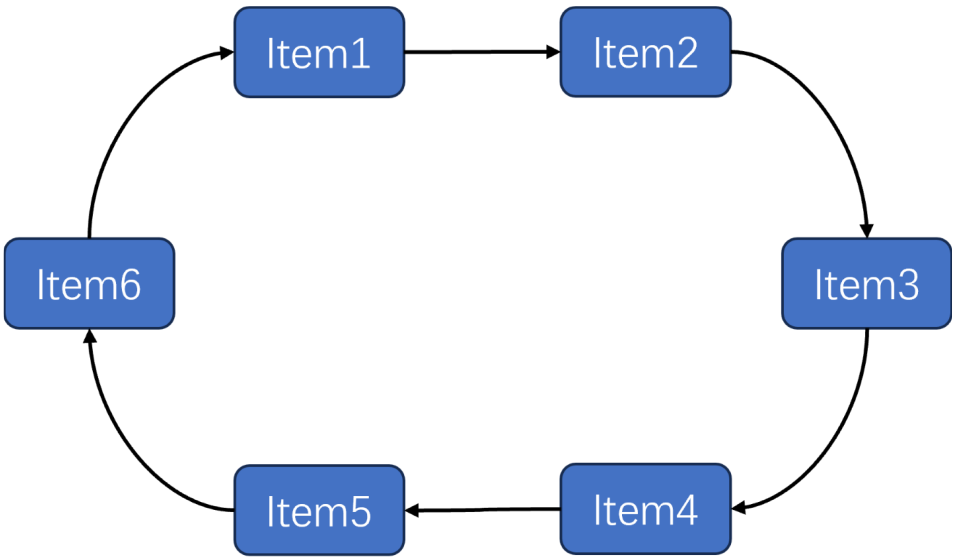
键值存储。位于服务器上的键值存储系统，可以完成put和read操作。

键值存储

本项目使用的键值存储数据结构为“Extensible Hash Table based on Hot Ring”。Hot Ring参考论文《HotRing: A Hotspot-Aware In-Memory Key-Value Store》，是一种热点感知的内存键值存储。本项目在原论文的基础上，结合可扩展哈希表（Extensible Hash Table），完成了一个可以按需扩展和热点感知的内存键值系统。

HotRing结构

在传统的哈希表中，处理哈希冲突的方式包括拉链法、开放寻址法等。拉链法无法做到热点感知，在链表中越靠后的键值对，查找时消耗的时间越快。HotRing在拉链法的基础上做了两点改动。首先，在插入时，按照键值对的键大小插入到规定位置，保持链表有序。这样做可以加快查找速度。其次，让链表首尾相连，构成一个环。下图就是一个hotring的结构。



哈希表的每个bucket保存一个head指针，指向其中的某个Item。距离head指针越近的Item在访问时所需要访问内存的次数越少。对于拉链法来说，Head指向的Item是固定的或者随机的。在HotRing数据结构中，head指向的Item是经过采样分析之后计算得出的，是使得某段时间内整体访问次数最少的最优解，这也是为什么hotring可以做到热点感知。

采样分析和调整head

对于一个环来说，环上的热点可能是不断变化的。这就要求每个一段时间就需要重新调整head的位置，这依赖于前期收集的数据。**采样**就是收集数据的过程。对整个环的访问每进行R次，如果第R次访问时要访问的Item并不是head指向的Item（也叫做hotspot miss），就启动采样。采样的范围也是R次访问。在采样进行时，系统会记录环上每个键的访问次数，以及每个键在被访问时hotspot hit的次数（作为head指向的Item被访问）。在采样结束后，会启动adjust hotspot程序（调整head）。

在adjust hotspot程序中，会计算每个键作为head时环的平均访问开销。具体来说，$Item_t$ 作为head的环平均访问开销计算如下。k为环上Item的数量，$1 \leq t \leq k$。

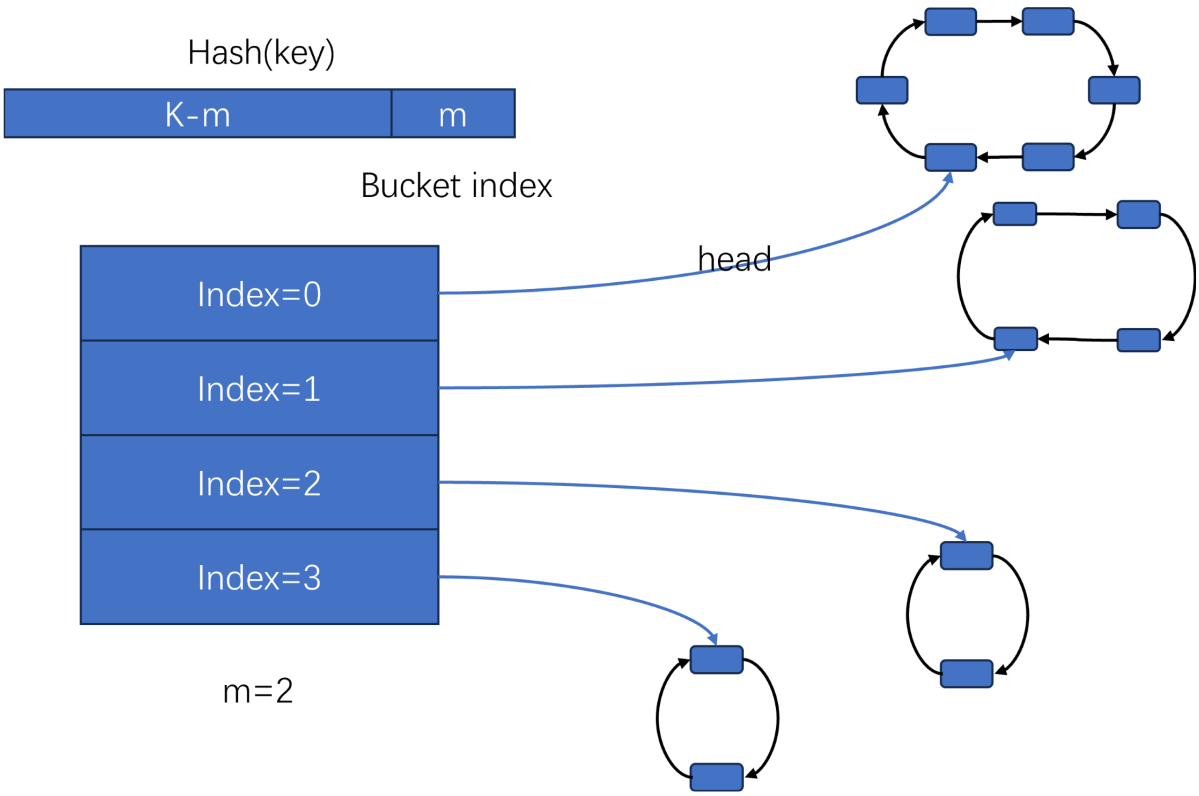
$$W_t = \sum_{i=1}^{k} \frac{n_i}{N} \left[(i-t) \; mod \; k \right]$$

算出的最小 W_t 代表可能存在的最小环平均访问开销。将 $head$ 指向 $Item_t$ 是理论上的最优解。

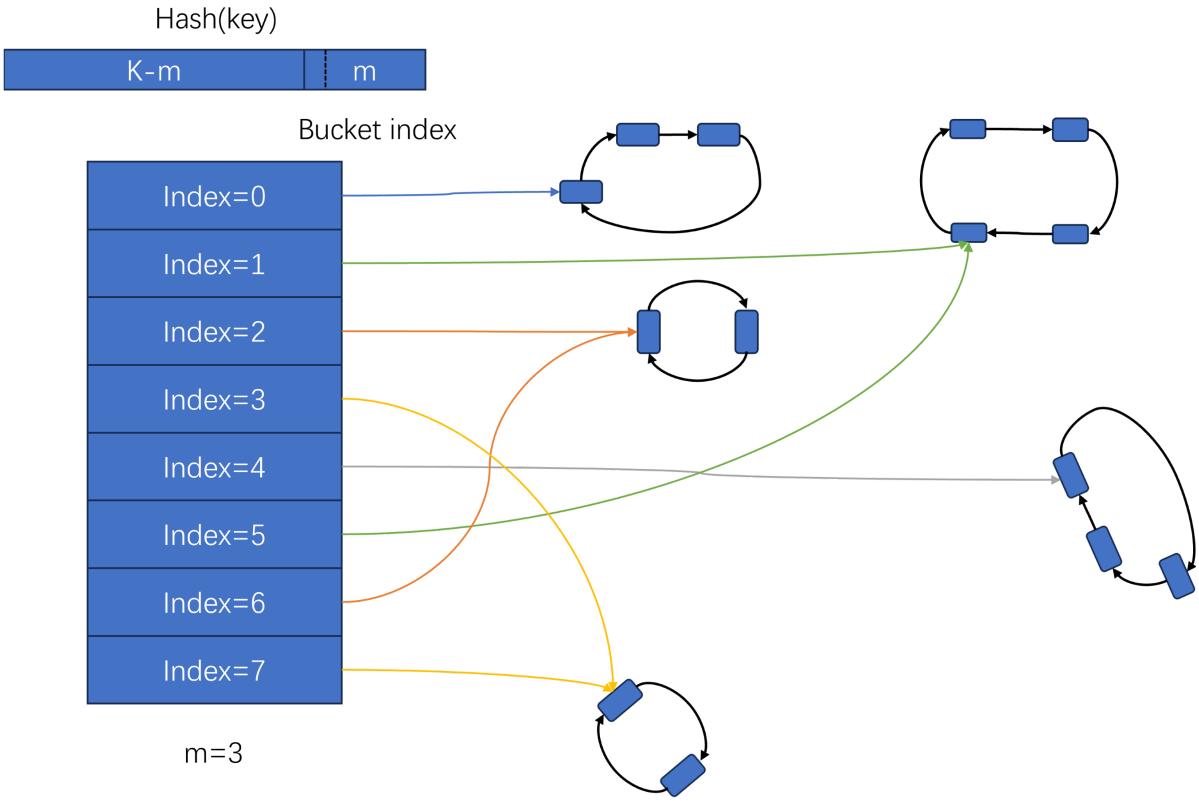
可扩展性和rehash

上面讲的环可以作为哈希表的一个bucket。随着put的进行，环上的元素会越来越多，导致性能下降。需要进行哈希表的扩展和rehash。一般来说，哈希表在扩展时会将桶的数量增加一倍，然后将原来桶中的元素rehash到新的桶中。但是如此大规模的rehash会导致环上的结构遭到破坏，一是有序性，rehash需要在新的ring上重新排序一遍。二是经过采样分析和调整后，目前head指向的位置为最优位置，一旦触发扩展和rehash，新的环需要重新采样和调整到最优位置。所以，我使用extensible hash table。这种哈希表的扩展是逐步的，只会将元素较多的bucket进行rehash。

具体来说，我将key进行哈希，并且将结果的后m位作为哈希值。如下图所示，此时m=2，哈希表有4个bucket。



如果设置的触发扩展的元素数量位6，此时index=0的桶触发哈希表扩展。扩展时，首先m=m+1=3，此时哈希表有8个bucket。由于只有index=0的桶需要扩展，所以只会对index=0的桶中的元素进行rehash，其他桶中仍然保持原来环的结构不变。如下图所示。



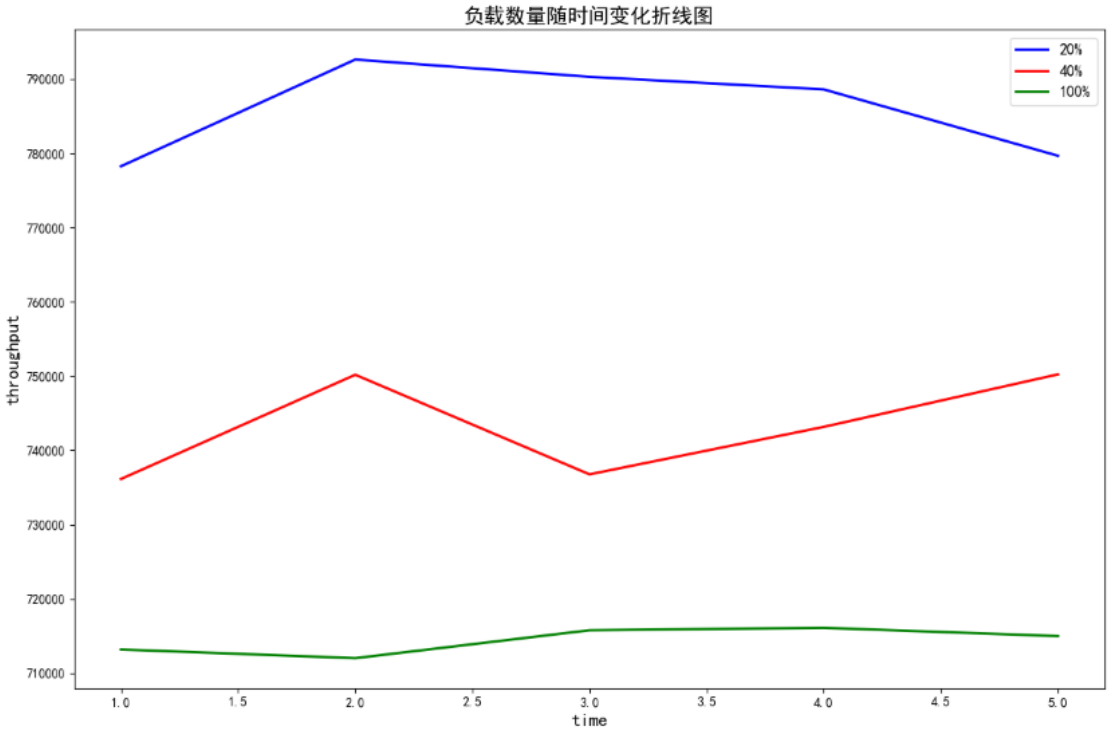
可以看到原来index=0的桶经过rehash后分割成了两个新的ring，分别由index=0和index=4指向。此外，其他桶没有变化。新增加的index=5的桶指向了index=1相同的ring，index=6的桶指向了index=2相同的ring，index=7的桶指向了index=3相同的ring。同时m增加了1。这样的扩展和按需rehash会尽可能保护已经形成的hotring的有序结构和head的指向。extensible hash table非常适合hotring这样有重要结构信息需要保护的桶。

实验评估

本次实验采用的实验环境如下。

CPU	L1 cache	L2 cache	L3 cache	Main Memory
AMD Ryzen 5 6600H with Radeon Graphics	384KB	3.0MB	16.0MB	16GB

下面将展示每分钟负载数量随时间的变化图

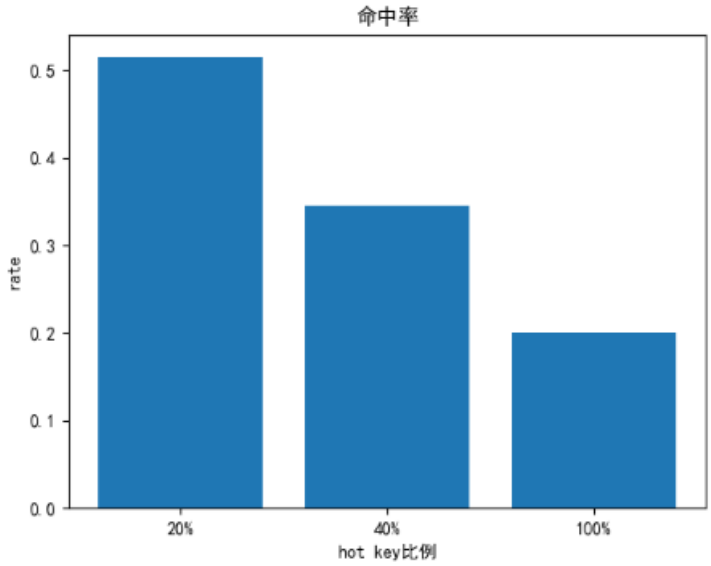


三条折现分别代表当hot key所占比例分别为20%，40%和100%时键值存储每分钟的吞吐量。从20%到40%再到100%，吞吐量越来越小。主要是因为hot key比例越高，生成的key热点会越不集中，从而导致热点的命中率（hotspot hit）更低。导致键值存储无法很好的捕捉到hotspot。其中最坏的情况是当hot key比例为100%时，也就是热点是均匀分布的，会导致热点感知完全失效。

除此之外，随着时间的变化，吞吐量是不断变化的。一方面与网络传输的IO有关，另一方面随着时间的增加，基本所有的键都已经插入过，read和put都退化成了read。此时热点感知的优势会体现出来，吞吐量会比刚开始的时候要高。由于第一分钟就已经产横了70万的吞吐量，只有一小部分时间（几秒钟）会出现部分键没有插入过的状态，在以分钟为单位进行的数据收集中，无法体现出来。

总的来说，不同hot key比例的吞吐量的大小关系与期待的一致。但是仔细看纵坐标可以看到，差别并不是很大。主要原因是socket通信带来的网络IO开销。与网络IO开销相比，热点命中率低带来的内存访问开销影响较小。从而导致100%hot key的吞吐量并没有比20%的吞吐量小很多。

所以，下面我将比较另外一个数值，也就是热点命中率（hotspot hit rate）。将每次put和read操作看作一次访问，如果每次访问时，head所指向的item刚好是要put或read的item，则表示一次热点命中。最终，热点命中率 = 热点命中次数 / 访问次数。



从图中可以看到，20%比例的命中率达到了50%以上，而40%和100%的命中率分别为33%和17%。三者之间有比较大的差距。我相信如果排除网络IO开销的影响，比如在放大每次内存访问的开销（每次内存访问时，sleep比较小的时间），各个比例的吞吐量之间的差异还是比较明显的。根据上面的分析和图表的展示，可以看到Extensible Hash Table based on Hot Ring在热点感知方面是有比较好的效果的。

总结与改进方向

hotring相对于传统的拉链式哈希冲突处理方式，有更好的热点感知能力。可扩展哈希表的按需rehash特性非常适合hotring的需要保护结构信息的需求。通过实验结果也能看出，键值存储有着比较好的热点感知能力。

在实验时，我发现hotring在环中有多个热点时，有个现象非常值得分析。

```
index : 10971
key : Y18WnmRS value : bI78ED0O6S0jPTgt total count : 599 total count as hot : 573
key : vHI3wexT value : LYGVLEmTb69Vp6F9 total count : 32 total count as hot : 0
key : CHg9DMOT value : tnPYi0fOU2tCADkB total count : 40 total count as hot : 0
key : LomFpl1E value : UEpuHNOp2R6htdnU total count : 42 total count as hot : 0
```

```
index : 10987
key : az24Tu29 value : pJBQTtt8NklkkQJo total count : 601 total count as hot : 558
key : k0YAfNwd value : R6G6IwcurQPdUEf1 total count : 600 total count as hot : 39
key : 4Cbq1VVx value : xtPnsIv0KyfduLXn total count : 34 total count as hot : 0
key : FUoT0CQ5 value : 86NiGUNDGL4JluKs total count : 38 total count as hot : 0
key : GGalSQEe value : E07QuBeNAKjYMRre total count : 44 total count as hot : 0
key : QJnbx5mw value : hX6krEy9JmS5pAH5 total count : 29 total count as hot : 0
```

上面是hot key比例为20%时，两个桶中每个Item的总访问次数和作为head被访问的次数（热点命中）。明显看到第一个桶只有第一个键是热点键，被访问了599次，其他键只被访问了不到50次。热点感知系统也不负众望，599次访问中有573次都是作为head被访问的（热点命中）。而第二个桶有两个热点键，分别是第一个（az24Tu29）和第二个（k0YAfNwd），被访问了601次和600次。其中第一个键（az24Tu29）比较好，601次中有558次都是热点命中。但是第二个键（k0YAfNwd）600次中只有39次是热点命中。这是没办法的，因为一个桶只有一个head，不可能同时让两个热点键都成为head。但是热点感知会尽可能的让环中每次访问的平均内存访问次数最小。也就是说，虽然不可能让多个热点键同时作为head，但是热点感知会尽可能选择一个head，让这个head离每个热点键的平均距离最近。这就是为什么第二个桶head会一直在第一个键（az24Tu29）上，因为此时第二个热点键只需要二次访问即可。而如果head在第二个键（k0YAfNwd）上，另一个热点键（az24Tu29）需要6次访问才能访问到！

所以，当环中有多个热点键时，hotring会尽可能让每个热点键被访问的平均次数最少。虽然如此，当一个环中有多个热点键时，性能仍然会受到影响。这显然比多个热点键分布到不同的桶中要承担更大的开销。所以，当环中有多个热点键时，就需要考虑扩大哈希表并且对该桶进行rehash操作了，让热点键尽可能分散开以达到更好的性能。这是该项目可以改进的部分。

源代码

本次实验的源代码已上传github，如需查看访问网址：https://github.com/ZepengLi111/hotring

参考文献

[1] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: a hotspot-aware in-memory key-value store. In Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20). USENIX Association, USA, 239–252.