Spatio-Temporal Locality in Hash Tables

Matt A. Pugh
Supervised by Prof. Stratis Viglas
University of Edinburgh
10 Crichton St.
Scotland
matt.pugh@ed.ac.uk

ABSTRACT

The overall theme of this Ph.D. is looking at ways to use emerging NVM (Non-Volatile Memory) technologies in real-world data-science scenarios. It is hoped that the exploitation of the characteristics of the technology will result in performance improvements, defined as being either/or an increase in computational throughput and energy-use reduction. Primarily, this has been through the inclusion of temporal locality into HopH (Hopscotch Hashing) by [2]. The problem of highly-skewed access patterns affecting lookup time and required computation is shown through a simple model. A simulator is then used to measure the expected performance gains of incorporating temporal locality, given different HT (Hash Table) configurations. This work was originally motivated by NVM, as a way to mask the extra latency anticipated, but the work is applicable to HTs in DRAM (Dynamic Random-Access Memory) also. The second area of interest in the Ph.D. is looking at exploiting the characteristics of NVM for different families of machine learning algorithms, though this paper focuses solely on the former.

1. INTRODUCTION

At this stage, the primary focus has been at introducing spatial and temporal locality into HopH – taking the concept of minimising amortised average lookup time into HopH will provide spatio-temporal locality with an amortised complexity of $\mathbf{O}(1)$ for the typical HT operations; `Get()`, `Insert()`, and `Delete()`. Details on the approaches undertaken are provided in Section 3. This is motivated by skewed access patterns, following Zipf's law, and high-performance systems, such as RDBMS (Relational DataBase Management System) join operators.

In its simplest form, a Hash Table (HT) T is an M-bucket long associative array-like data structure that maps a key k of any hashable type to value v. A given bucket B_i in T can have S slots for key/value tuples. The occupancy (or load factor) of a HT is $0 \geq L \geq 1$[1]. The benefit of a HT is that the associated value's position i in memory is calculated directly using a hash function $h(k)$, $i = h(k) \mod M$.

Proceedings of the VLDB 2017 PhD Workshop, August 28, 2017. Munich, Germany.

1.1 Self-Reorganising Data-Structures

A ST (Splay Tree) is a self-adjusting binary search tree introduced by [6] that has the property that the most frequently accessed node is near the head of the tree. STs have the amortised time-complexity for `Get()` and `Insert()` operations of $\mathbf{O}(\log n)$, and the goal of their invention was to minimise average time of lookups in worst-case scenario in real-time terms. Their solution to achieving this is to minimise the number of node traversals by ensuring the most-frequently-accessed nodes are as close to the head as possible, a property which is closely tied to temporal locality, although not exactly the same thing. [6] acknowledge that the computational cost of ongoing reordering is high, and may be problematic given the target application workload. This is clear when considering the rotation of sub-trees in a `Splay()` operation – there is no spatial locality, or cache-friendliness, guaranteed by the structure of the tree in memory, as such pointer-chasing in rotations is inevitable.

In databases, [3] describe *cracking*; a method of self organisation data by query workload. This process is tuned for a RDBMS in which a column of interest A_{CRK} is retained as a copy, and is continuously reorganised based on the queries touching it. This is analogous to data *hotness* as we wish to view it, and can be considered a cache-tuning problem, similar to a LRU (Least Recently Used) cache. As A_{CRK} is a column, they are able to effectively partition the data itself to improve probing response times significantly. Partitioning within a HopH context is less intuitive as there are, at any position, H overlapping neighbourhoods.

2. MODEL AND SIMULATOR

The structure of a HT is entirely dependent on the order in which keys were inserted into it. For example, the hottest element of the neighbourhood, or even entire table, over some given time period, may be in the last available bucket in a neighbourhood, leading to multiple evaluations before finding the tuple required.

2.1 Model

Assume a constructed Hash Table T, of which we have a set of all keys K in T. Let X be the sequence of accesses,

[1]Assuming no chained-structures resulting in $L > 1$ load factor

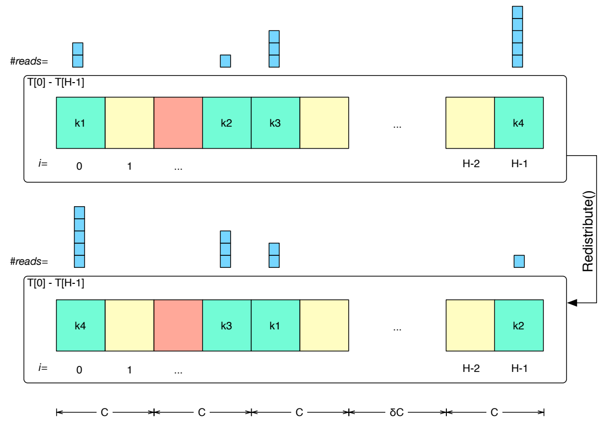

Figure 1: This figure illustrates buckets (coloured blocks) in neighbourhoods (colours) over some contiguous range of buckets within the table T. For a given neighbourhood, access patterns may naturally fall as the above distribution, where C is the cache-line size and each blue block represents a constant number of accesses to the associated key k_i. Suppose a request is made for the hottest item k_4, there will be $3 + \delta$ cache-invalidations before hitting the hottest data k_4 if $T[h(k_4)]$ is not already cached. Reordering the data such that k_4 is in position 0, minimising cache-invalidations, is the optimal distribution. Note that the distribution of colours in T does not change.

such that every element $x \in X$ exists in K. We define a simple cost model $\Phi(T, X)$ that gives a unit cost of a bucket traversal as β, and a slot traversal and key comparison as σ unit cost, where α and γ are the number of respective operations.

$$\Phi(T, X) = \alpha\beta + \gamma\sigma \qquad (1)$$

It is simple to use this model with a sufficient access-skew in X to show good potential gains in the worst case of the configuration of T. Consider a RO (Read-Only) T where some key k has hashed into bucket B_i, but the only available slot in the otherwise fully-occupied neighbourhood ν_i is at B_{i+H-1}, slot S. Assume that X is completely skewed, such that there is only one k present, repeated A times. In this case, the cost of accesses is:

$$\Phi(T, X) = AH(\beta) + ASH(\sigma)$$

In this example, it is clear to see that minimising the values of $\alpha = AH$ and/or $\gamma = ASH$ are the only areas of movement upon which we can optimise, if we assign some real-world values, where $S = 3, H = 64, \sigma = 15, \beta = 15, A = 1 \times 10^9$, we obtain a cost of:

$$\Phi(T, X) \simeq 3.84 \times 10^{12}$$

When we assume that X' the hottest element k' in the first slot of the first bucket in a neighbourhood, the cost is naturally far lower:

$$\Phi(T, X') = A(\beta + \sigma) \simeq 3.0 \times 10^{10}$$

This is the lowest possible cost incurred for T and X', and is therefore referred to as the *oracle*. With this reordering, 99.22% of cycles would be saved versus T with X. This approach has the problem that manually attempting to cover a number of configurations of T to find the expected benefit of rearrangement would be tedious. To deal with this, we employ a simulator.

2.2 Simulator

The developed simulator provides the estimated performance benefit of performing reordering for varying table sizes, load factors and skews. This is done using a number of discrete probability distributions to obtain different configurations of T, in order to explore the problem space we are interested in:

1. The number of slots that each bucket will have populated in the simulator's construction is sampled from a Multinomial distribution, for bucket B_i this is ς_i.

2. $N = \lfloor S \times M \times L \rfloor$ samples are drawn from a Zipfian distribution $\mathcal{Z}(\alpha)$. These are the random *hotness* values that are then inserted into each bucket $B_i \in T$ over ς_i slots in each bucket. This approach gives a load factor L and skew α over the data. T is then randomly permuted to distribute the remaining (for $L < 1$) empty buckets throughout the table.

3. As the number of buckets occupied within a neighbourhood ν_i is not a constant, either $\lfloor \xi \rfloor$ or $\lceil \xi \rceil$ buckets per neighbourhood are allocated with probability $\pi_\beta = \lceil \xi \rceil - \lfloor \xi \rfloor$, where ξ is the *expected number of elements in a neighbourhood*, given in [5]. π_β must be within the $[0, 1]$ interval, and is interpreted as the probability that the neighbourhood contains the upper-bound of entries. At each neighbourhood root, neighbourhood occupancy is sampled from a Binomial distribution with probability π_β.

4. Finding the locations within the neighbourhood for occupancy should not be done linearly, and must be at least partly stochastic to emulate the chronological insertions over multiple neighbourhoods. Approaching this in a linear manner would construct T such that all insertions happened to populate the table in exactly linear order; this behaviour is not realistic. Instead, we randomly draw samples from a Poisson distribution that has a mean $\lambda = 2$, in order that the mass of the distribution is towards the beginning of the neighbourhood, but may be further on.

2.2.1 *Output & Discussion*

Table 1 shows the output obtained thus far from the simulator, this shows that even on a coarse reordering policy, we begin to approx a factor of 2 improvement in terms of work performed. Performing fine reordering over the same configuration invariably leads to better results than coarse. A key caveat is that this metric concerns itself only with the hottest element in the table, extensions are underway to take a more wholistic view of potential performance gains. The fact that as $L \to 1$, gains appear to disappear is entirely expected. This is due to the fact that as a table becomes full, it is to be expected that most neighbourhoods will only have one bucket in T, therefore reordering is not possible.

Table 1: Simulator results showing the oracle measurements for the simulator, in terms of percentage of instructions avoided.

Size M	Load L	Skew α	Coarse (%)	Fine (%)
1.00E+03	0.7	1.1	30.59	92.1
1.00E+03	0.7	2.1	31.49	99.53
1.00E+03	0.7	3.1	1.13	3.53
1.00E+03	0.8	1.1	27.86	83.61
1.00E+03	0.8	2.1	44.04	96.3
1.00E+03	0.8	3.1	0.58	2.08
1.00E+03	0.9	1.1	47.5	95.7
1.00E+03	0.9	2.1	48.43	99.04
1.00E+03	0.9	3.1	0.83	1.71
1.00E+03	1	1.1	0.99	2.64
1.00E+03	1	2.1	9.36	14.9
1.00E+03	1	3.1	0.59	1.1
1.00E+05	0.7	1.1	29.86	82.3
1.00E+05	0.7	2.1	35.39	98.92
1.00E+05	0.7	3.1	35.74	95.81
1.00E+05	0.8	1.1	32.82	93.67
1.00E+05	0.8	2.1	33.68	99.64
1.00E+05	0.8	3.1	32.64	95.69
1.00E+05	0.9	1.1	32.97	99.33
1.00E+05	0.9	2.1	35.43	97.79
1.00E+05	0.9	3.1	31.42	97.17
1.00E+05	1	1.1	0.01	0.02
1.00E+05	1	2.1	0.65	1.96
1.00E+05	1	3.1	0.01	0.01
1.00E+07	0.7	1.1	34.16	97.81
1.00E+07	0.7	2.1	34.14	99.2
1.00E+07	0.7	3.1	33.8	97.37
1.00E+07	0.8	1.1	33.03	98.04
1.00E+07	0.8	2.1	32.52	96.91
1.00E+07	0.8	3.1	31.81	94.19
1.00E+07	0.9	1.1	31.87	95.88
1.00E+07	0.9	2.1	26.05	77.39
1.00E+07	0.9	3.1	33.89	99.79
1.00E+07	1	1.1	0	0
1.00E+07	1	2.1	0	0
1.00E+07	1	3.1	0	0

3. METHODS

HopH is used as the basis for the solution. The argument of reordering based on the *hotness* of the data itself given by [1] is highly aligned with the goals for this work. This work differs as the specific objective is achieving spatio-temporal locality, in order to minimise average lookup times. The value of S is selected such that a bucket B_i fits within a cache-line size $C = 64$B. In order that the size of the bucket $|B_i| \leq C$, we choose S, where the size of a tuple $|\tau| = 8 + 8 = 16$B, and μ is the size of any required meta-data, to be $S = \left\lfloor \frac{C-\mu}{|\tau|} \right\rfloor$. Different access patterns are simulated by drawing samples from a Zipfian distribution $\mathcal{Z}$, whose parameter α affects the skew of the samples obtained.

3.1 Reordering Strategies

This section describes a number of re-ordering strategies for the placement of data in a neighbourhood. These methods look at coarse, down to fine tuple-level, and heuristic reordering operations.

3.1.1 Fine - Intra-Bucket (FIntrB)

This method simply sorts the tuples within a bucket B_i (that must be of the same neighbourhood) by hotness. This is performed using a priority queue, inserting the tuples and ordering by their number of accesses, before reinserting them into B_i. As we know that $|B_i| \leq C$, there are no further bucket traversals required, and any potential gains are purely in terms of operations performed within B_i for a `Contains()` or `Get()` and, as such, will be minimal.

3.1.2 Coarse Inter-Bucket (CIB)

We can express the overall hotness of a bucket B_i by the summation of accesses to all tuples contained within it. With this method, we do not care about the order of the tuples within slots, but simply that the most-frequently-hit bucket is closest to the neighbourhood root. A clear compromise of this strategy is that it does not care about the distribution of accesses within B_i; in the example where $S = 3$ and B_i has one element with many accesses, but two without, and B_j has a uniform distribution whose total (summed) access is more than B_i, B_j will be promoted first.

3.1.3 Fine Inter-Bucket (FIB)

By far the most expensive operation, this seeks to redistribute the neighbourhood at a tuple-level using a priority queue to order tuples by hotness, before reinserting them into all buckets within the neighbourhood. The positions of buckets within the neighbourhood do not change. In terms of memory use, this strategy is the most demanding. Potentially, if every possible bucket in a neighbourhood ν_i belongs to that neighbourhood, there will be many elements to copy and reinsert over H buckets. Although the memory traversal will be sequential, this will involve $H-1$ cache-invalidations. The trade-off for this cost is that we are guaranteed to have all tuples in correct order, with none of the compromises of FIB (Fine Inter-Bucket) or CIB (Coarse Inter-Bucket).

3.1.4 Heuristic

The simplest approach is a direct-swap heuristic, which simply compares the current tuple τ_i with the first tuple of the neighbourhood, τ_r, if the τ_i is the hotter of the two. This approach should not have a high computational overhead, as in traversing the neighbourhood to find τ_i, we already stored a reference to τ_r upon first encountering it. If τ_i is hotter than τ_r, swap them and their distribution entries.

3.1.5 Approximate Sorting

This approach exploits the spatial locality afforded by HopH; we are guaranteed that all H buckets within neighbourhood ν_i are in the same, homogeneous region of memory. Once these pages are in the cache-hierarchy, latencies in accessing them are reduced and, depending on the cache layer, very efficient. As the `Get()` or `Contains()` operation traverses ν_i, a Bubblesort-like operation can be applied based on hotness.

3.2 Epochs & Squashing

In order to be adaptive over time, there must be a series of rules that govern how the hotness of data changes over times and accesses. For approaches where there is an absolute trigger for reordering, an epoch is defined at a neighbourhood level, and is its state before a reordering method is invoked.

For non-triggered reordering methods, there must be a continual state of adaptation for the associated meta-data.

Once tuples in ν_i have been rearranged, and the numeric values of tuple accesses have been reduced in some manner, the hottest element in ν_i should remain so (*preservation of hotness*). For epoch-based approaches, this means the skew and shape of the distribution should be roughly equal after an epoch, but smaller in magnitude. For non-epoch-based approaches, the distribution should be more fluid, dynamically changing all values in ν_i regularly. Those tuples in ν_i that have not had many accesses should have historic access counts reduced, until sufficient epochs have passed that they no longer have any hotness in epoch-based reordering (*cooling*).

3.3 Deletions

Handling deletions in HotH (Hotscotch Hashing) should follow two principles further to a baseline HopH deletion:

1. For per-slot level meta-data, the distribution entry (number of reads) for the tuple to be deleted should also be deleted from any per-bucket counter if present, before being erased itself.

2. If any slot that is not the first slot of a bucket contains the tuple to be deleted, subsequent slots should be shift towards the first slot, to minimise unnecessary traversal gaps in the bucket structure.

After this, should the bucket be empty after tuple deletion, the standard HopH process is followed.

3.4 Invocation

Finding the balance between the severity of the reordering strategy employed, and how and when to trigger it are key points in this work. We explore a number of invocation strategies, and the various forms of meta-data required to permit them.

3.4.1 Trigger-based

The simplest method of invocation is that of setting a minimum number of accesses to a bucket or neighbourhood, dependant upon the reordering strategy used, where a counter $\rho = 0$ is meta-data within the bucket structure. Upon every `Get()` or `Contains()` call, ρ is incremented and evaluated. Once ρ exceeds some instantiation-defined value σ, the reordering is invoked.

3.4.2 Probabilistic-based

Instead of storing any meta-data at a bucket/neighbourhood level, we can simply state that with some probability $\pi_r = \mathbf{P}(\text{Perform Redistribution})$, we will perform a reordering. A key point in using this method is to avoid paying the cost of the given reordering strategy often; however the generation and evaluation of PRN (Pseudo Random Number)s is itself a non-zero cost. Conclusions drawn from experimentation using a number of PRNG (Pseudo Random Number Generator)s show that the `xorshift` a good candidate. Further work is looking into simpler masking / shifting of an integer value, as statistically-sound randomness isn't mandatory.

4. EXPERIMENTS

1. The base experiment looks at the Avg.CPU (Average CPU Cycles) metric of creating tables based on the same input data, and performing the same set of queries, in the same order, amongst different table configurations and comparing their results. Input data is randomly generated, and HopH is used as a baseline.

2. A SHJ (Symmetric Hash Join) is constructed of two tables. A HotH configuration is compared with one backed by HopH, to measure the difference the adaptive approach we propose makes to realistic scenario. Experiments evaluating SHJ performance will use existing data-sets, over different relation sizes.

5. DISCUSSION

Implementations of all methods described have been complete, with results expected soon. As the overarching theme of this thesis is NVM, work will be conducted to exploit its characteristics to aid HotH. An approach by [4] provides a way for leaf nodes being stored in a DRAM/NVM hybrid B+Tree, where the system prefers `Contains()` operations to `Get()`, as keys are stored in quick DRAM and values in NVM. Following a similar methodology for hybrid NVM / DRAM should provide a fast key lookup for `Contains()`, as we can now fit many keys within C, but also gives slower reordering due to the extra cost of moving around NVM vs. DRAM. Unlike the problem solved by [4], we must still ensure spatial and temporal locality. Intuitively, this could mean analog structures of M elements in DRAM and NVM with a translation function $t(i, s) \rightarrow j$ that takes the bucket and slot numbers i, s respectively, and maps it to a position (offset) j in the NVM table.

6. ACKNOWLEDGMENTS

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh.

7. REFERENCES

[1] S. Albers and M. Karpinski. Randomized splay trees: Theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.

[2] M. Herlihy. Hopscotch Hashing. pages 0–15.

[3] S. Idreos, M. Kersten, and S. Manegold. Database Cracking. *CIDR '07: 3rd Biennial Conference on Innovative Data Systems Research*, pages 68–78, 2007.

[4] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*, pages 371–386, 2016.

[5] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[6] D. D. Sleator and R. E. Tarjan. Self-adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.