Collision Resolution in Hash Tables for Vocabulary Accumulation During Parallel Indexing

Matt Crane
Department of Computer Science
University of Otago
Dunedin, New Zealand
mcrane@cs.otago.ac.nz

Andrew Trotman
eBay Inc.
atrotman@ebay.com

ABSTRACT

During indexing the vocabulary of a collection needs to be built. The structure used for this needs to account for the skew distribution of terms. Parallel indexing allows for a large reduction in number of times the global vocabulary needs to be examined, however, this also raises a new set of challenges. In this paper we examine the structures used to resolve collisions in a hash table during parallel indexing, and find that the best structure is different from those suggested previously.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing – Dictionaries

General Terms

Algorithms, Measurement, Performance

Keywords

Indexing, Parallel, Collision resolution

1. INTRODUCTION

During indexing of a collection, the vocabulary for that collection needs to built. As terms are encountered the vocabulary is consulted so that postings, and statistics, for that term can then be updated. Terms within a collection typically follow Zipf's law which states that the frequency of a term in a collection is proportional to the inverse of its rank in a frequency table [7]. That is, let α be a parameter that describes the decay rate of frequencies, and f_1 be the frequency of the most common term. The frequency of the k^{th} most common term, f_k, is then given by $f_k \approx {}^{f_1}/_{k^\alpha}$. Figure 1 shows this distribution of collection frequencies (CF) occurring in the Wall Street Journal (WSJ) collection.

When indexing documents in parallel, the vocabulary only needs to be consulted once per term, rather than per term

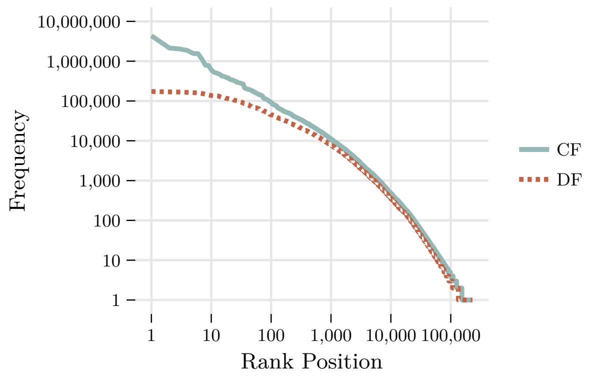


Figure 1: Frequency rank of a term plotted against its collection (CF) and document (DF) frequencies from the WSJ collection.

occurrence. However, as shown in Figure 1, this is still a skew distribution. Because there are multiple indexers that are accessing the vocabulary concurrently, the best structure may be different from a serial indexer.

2. RELATED WORK

Williams *et al.* [6] tested a range of structures that took the skew distribution into consideration. The structures they considered were splay trees, periodic splay trees, red-black trees, and randomised search trees. A splay tree is a binary search tree (BST) where nodes are rotated towards the root after access, a periodic splay tree does so periodically. Red-black trees are a self-balancing BST. Randomised search trees are a treap (discussed in Section 4.4) where the heap property is randomly assigned. They concluded that of these structures the periodic splay tree was the best performing, however, overall a hash table was the best.

Heinz & Zobel [2] also tested a range of tree structures, including those of Williams *et al.*, with the addition of a frequency-adaptive BST (fab tree), and a burst trie. A fab tree is a treap where the heap property is an access counter, which approaches the collection frequency of the term. A burst trie is a trie where suffixes are stored at a trie node, rather than individual characters. The trie nodes are burst apart when they contain too many suffixes. Heinz & Zobel also conclude that hash tables are the fastest overall structure.

This prior work shows that the hash table is the fastest structure overall. However, when multiple terms hash to the

Collection	Documents	Unique Terms	Total Terms
Wall Street Journal (WSJ)	173,252	229,498	83,270,622
.GOV	1,247,753	5,410,059	1,100,383,268
.GOV2	25,205,179	37,236,871	20,100,412,569
ClueWeb09 Category B (CW09B)	50,220,423	96,111,883	55,466,785,086

Table 1: Document, unique term, and total term counts for tested collections.

same slot a collision occurs and must be resolved. Due to the birthday paradox, even with very large numbers of slots and small numbers of terms, the chances of a collision can be high. For example only $109,124$ terms are required for a 75% chance of a collision across 2^{32} slots. Even if no collisions occur across n slots after hashing n terms, due to the pigeonhole principle, the next term *must* cause a collision.

The structure used for resolving collisions within the hash table has received little attention. Previous work has used linked lists to resolve collisions. These lists have utilised various heuristics, and these are discussed in Section 4.1. Our research question is then *"Which data structure for collision resolution in a hash table is best in a parallel indexer?"*

Experiments are performed using the ATIRE search engine and indexer [5], revision `0ab3cd0228d8`. Serialisation of the index, and accumulation of postings are disabled to allow for more experimental repetition. These account for $\approx 50\%$ of indexing time, and are constant regardless of vocabulary accumulation method, so can be disabled without affecting results. Table 1 shows the document, unique term, and total term counts for the collections used in experimentation. For example, CW09B contains 55B occurrences of 96M unique terms extracted from 50M documents.

3. PARALLEL INDEXING

The ATIRE indexer indexes documents in parallel and then merges these per document indexes into the final index. This means that the global vocabulary is only consulted relative to a terms document frequency, rather than collection frequency. This feature alone saves a substantial amount of lookups in the global vocabulary. For instance, *the* occurs 1.5B times across 42M documents in the CW09B collection, this feature then saves 1.46B lookups in the global vocabulary for *the* alone.

When a document indexer encounters a new term, the global vocabulary is consulted and a reference to the node in the global structure is stored. This saves work when the single document index gets merged into the global index. For nodes that have a reference, no lookup in the global vocabulary needs to be performed. For nodes that do not, the global vocabulary is consulted again and the node is updated if found, or inserted if not.

This feature saves a substantial amount of time, as the merge is single-threaded—to allow for sequential, contiguous, and unique document ids. The document indexers, however, are running concurrently. The presence of these multiple readers require that any manipulations to the resolution structure be performed using thread-safe operations. For chained structures this is achieved by using the atomic *compare-and-swap* operation on the pointers to other parts of the structure.

Even with this feature, the skew of document frequencies presented in Figure 1 must still be taken into account. For example, *the* appears in 42M documents in the CW09B collection. Every step down the resolution chain in which *the* appears would incur another 42M string comparisons per lookup.

4. RESOLUTION STRUCTURES

In this paper we examine a number of collision resolving structures: linked lists, BSTs, periodic self-balancing BSTs, and treaps. For the linked lists an *insert-at-back* heuristic is used as this was found to be the best performing heuristic [1, p. 121]. The selection of a BST that balances periodically follows the work of Williams *et al.* [6] who found periodic splaying out performed always splaying. The heap property for the treap is set to the document frequency of the term, this is equivalent to the fab trees of Heinz & Zobel [2].

4.1 Linked Lists

A number of heuristics for linked lists have been proposed to account for the term skew. The first of these is *insert-at-back* where new terms are inserted at the back. This heuristic makes the assumption that frequent terms will appear sooner in the collection than infrequent terms. Another commonly used heuristic is *move-to-front* where terms are moved to the beginning of the list on access. The ordering of terms in the list approaches ordering by CF.

Zobel *et al.* [8] showed that the *move-to-front* heuristic performed better than resizing the hash table. Büttcher *et al.* [1, p. 121] showed that the *move-to-front* and *insert-at-back* heuristics performed equally. As the *insert-at-back* heuristic is simpler to implement, this is selected for our experiments.

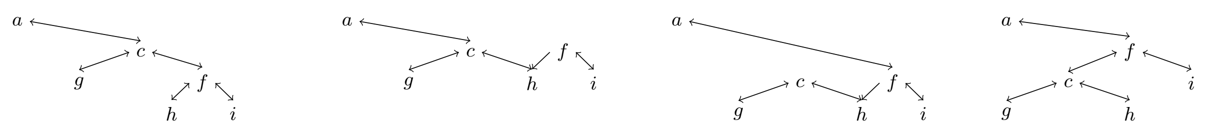
4.2 Binary Search Trees (BSTs)

A BST provides an ordering over terms that are inserted, typically a lexicographical ordering. The choice of a BST relies on two assumptions. The first of these, like the *insert-at-back* heuristic of linked lists, is that a more common term will appear before an infrequent term. The second assumption is that in running text the order of terms is sufficiently random to create a tree that is approximately balanced.

4.3 Periodic Self-Balancing BSTs

One disadvantage of BSTs is that performance is dependent on the ordering that terms are inserted. If the second assumption above does not hold, and terms inserted in sorted order then the BST performs as if it were a linked list. This affects both insertion and lookup, and causes what would otherwise be an $\mathcal{O}(\log n)$ operation to be $\mathcal{O}(n)$.

One technique to avoid this is to balance the BST. One solution that does so is the Day-Stout-Warren (DSW) algorithm [3, 4]. This is chosen over other methods as rebalancing only needs to be performed periodically. Rolfe shows that the time required to balance a given tree using this method is less than other methods [3]. In addition Williams *et al.* show that only periodically splaying trees improved run time when compared to always splaying [6].



(a) **Initial.** (b) **Step 1.** (c) **Step 2.** (d) **Final.**

Figure 2: Valid intermediary stages of treap rotation. The shown rotation occurs when f's document frequency is increased to be larger than c, resulting in an upward rotation.

The DSW algorithm degrades the BST to a linked list by repeated rotations. After this, every second node down a branch is rotated the opposite direction. This repeats until the BST is balanced and complete (filled from left to right). The balancing is performed in-place in $\mathcal{O}(n)$ time, and as it is only performed periodically, this cost is amortized across all insertions of new terms.

The rebalancing is triggered when a new node is created at a depth greater than d. In slots with more than 2^d unique terms this will trigger unnecessary rebalancing. The number of times rebalancing is expected depends on the chosen rebalancing depth. Even in a very dense hash table the expected depth of a fully balanced tree is minimal. A rebalancing depth that is set to the expected depth should be sufficient as this will only affect trees that are already degenerating.

4.4 Document Frequency Treap

A treap is a data structure that provides two orderings on the data. The first is the same as the BST described above. The second is the heap property, and for this, we select the document frequency. This is selected over the collection frequency as we are only considering the global structure and this is only consulted per document occurrence. As indexing proceeds, this approaches the final document frequency, putting the most frequent terms closer to the root.

Figure 2 shows an example of the stages of maintaining a treap. The shown manipulations occur after an update to node f has caused it to have a larger document frequency than c, but not larger than a.

5. EXPERIMENTS

Conventional advice for hashing suggests calculating a hash value, and then taking a prime number modulus of that value. However, the table sizes selected are powers of two following the advice of Bob Jenkins—“Table lengths should always be a power of two because that's faster than prime lengths and all acceptable hashes allow it.”[1]

The collections selected for experimentation (see Table 1) were chosen to provide a range of expected terms per slot (shown in Table 2) for a range of hash table sizes. On the WSJ collection for instance, there are an expected 3.5 terms per slot when 2^{16} table slots are available.

Figure 3 shows the effect of the size of the hash table (measured by the big length of the hash) and resolver on indexing time for the WSJ collection. As the number of collisions rises (by reducing the number of table slots) the performance of linked lists degrades substantially while the BSTs and treaps perform approximately the same regardless of table size, with treaps being consistently the worse of the two. Absent from these results are the periodic self-balancing BSTs (see Section 5.2).

[1]`http://burtleburtle.net/bob/hash/doobs.html`

Table	Collection			
Slots	WSJ	.GOV	.GOV2	CW09B
2^8	896.48	21,133.04	145,456.53	375,437.04
2^{16}	3.50	82.55	568.19	1,466.55
2^{24}	0.01	0.32	2.22	5.73

Table 2: Expected number of unique terms per hash table slot for various collections and table slots.

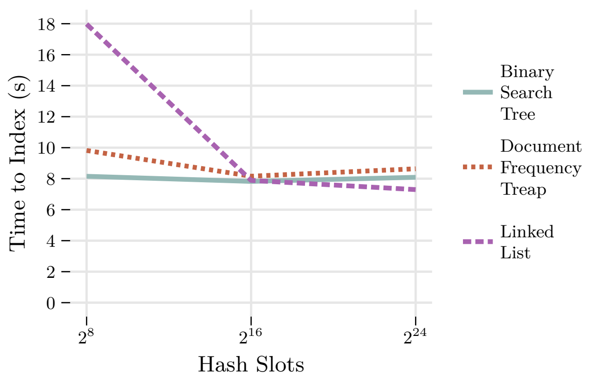


Figure 3: Effect of resolver and table slots on time to index the WSJ collection. Shown is the median of five runs.

The performance of resolvers is highly dependent on the number of terms in each slot. If only one term is in a slot then the advantages of a BST or treap over a linked list are lost as no resolution is performed. In fact, there might be a penalty, as additional memory is used for the tree-based resolvers that could cause more cache misses.

From Table 2 and Figure 3, 2^{16} slots for .GOV2 would introduce performance degradation comparable to 2^8 slots on the WSJ collection. Because of this, 2^{24} slots are used for remaining experimentation.

Figure 4 shows the effect that changing the resolution method has on the time taken to index using 2^{24} slots and across multiple collections. As discussed, the reason that the linked list performs on par with the BST resolver is because the expected number of collisions is low and thus the benefits of the BST are lost.

5.1 Document Frequency Treap

The treap consistently performs worse than a regular BST, despite the theoretical benefits and results of Heinz *et al.* [2]. The reasons for this are due to the nature of the parallel indexing carried out by ATIRE.

When treap maintenance is performed (exampled by Figure 2), any number of document indexers can be traversing the treap to find global references for a term. However, in the first (Figure 2b) and second (Figure 2c) steps of a ro-

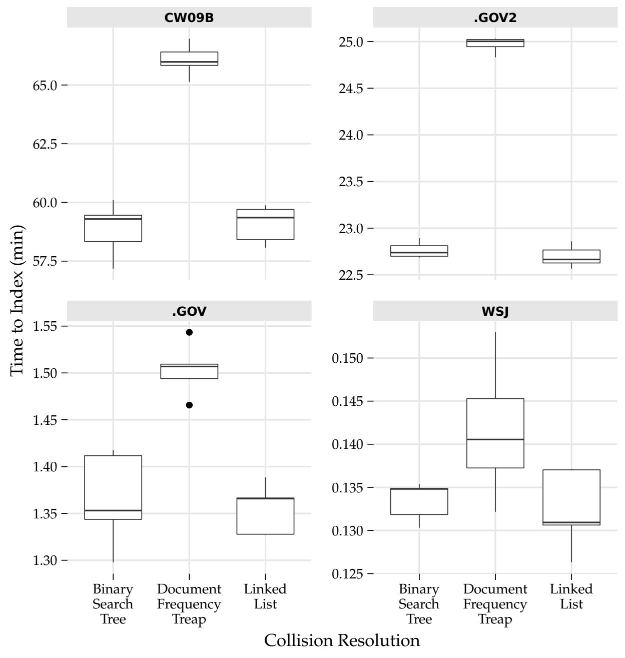


Figure 4: Effect of resolver on time to index a number of collections. Shown are the results of five runs and 2^{24} slots.

tation, while a term that is being looked for may occur in the global vocabulary, it may not be findable. This then requires additional lookups at merging time causing delays.

For example, suppose a document indexer has just found the term, f. It looks through the treap for the term and determines it needs to look to the right of node a, to node c (Figure 2a). After this happens, another document being merged in has increased f's document frequency causing it to rotate upwards in the treap. The first step of this (Figure 2b) sets c's right child to be f's left child, h. From this stage onwards, even though f has already been seen in the collection it will not be discoverable from c, the node that the document indexer is searching from. Depending on the size of the treap now rooted at c this could incur a substantial number of extra comparisons. However, the lookup performed when the document is merged into the index should resolve quicker, as f is now closer to the root.

While this reduction means the treap performs fewer comparisons, maintaining the heap property also incurs a substantial cost. Every time a term is encountered in a new document, the document frequency is updated and the heap property maintained. While a single such check is inexpensive, the collective number of these across an indexing run becomes a substantial factor.

It may be possible to amortize this cost by only periodically maintaining the treap, allowing for higher discoverability. However, this is left for future work.

5.2 Periodic Self-Balancing BSTs

When the rebalance depth is less than $\log_2$(expected terms), then rebalancing can occur when the tree is already balanced. As the rebalance depth, d, increases the number of rebalances decreases. When $d = 10$, the median (of five) time to index the WSJ collection is 56 seconds, compared with 8 when $d = \infty$, as a result of 155,000 rebalances.

For the tested depths, the index time was either statistically significantly slower ($d < 13$) or not different ($d \geq 13$) at the 0.01 level. The best value cannot be determined until after indexing has completed, as it is then that the expected number of terms per slot can be calculated.

In addition to the parameter sensitivity, this structure also has the same problem as the treap. During balancing the tree many rotations are performed and the document indexer lookups can fail. Even if the interleaving of rotations and lookup was such that all nodes were discoverable, the lookup can degrade to $\mathcal{O}(n)$ in the worst case.

It may be possible to adjust the rebalance depth during indexing if excessive rebalancing was performed. Alternatively, rebalances could be triggered when lookups perform large numbers of comparisons. Both are left for future work.

6. CONCLUSIONS

This paper presented an investigation into the structures used to resolve collisions in a hash table. Four structures were tested: linked lists with an *insert-at-back* heuristic, BST, periodically self-balancing BSTs, and treaps.

The periodic self-balancing BSTs were overly parameter sensitive, and the best parameter could not be determined prior to indexing. The performance of the remaining structures are all directly related to the expected terms per hash slot. Of these, the treap performed the worst, and reasons for this were presented. The linked list was shown to severely degrade as the density of the hash table went up, and so the BST is selected as the best performing collision resolving structure for parallel indexing.

References

[1] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[2] S. Heinz and J. Zobel. Performance of data structures for small sets of strings. In *Proceedings of the 25th Australasian Conference on Computer Science*, ACSC '02, pages 87–94, 2002.

[3] T. J. Rolfe. One-time binary search tree balancing: The Day/Stout/Warren (DSW) algorithm. *SIGCSE Bull.*, 34(4):85–88, Dec. 2002.

[4] Q. F. Stout and B. L. Warren. Tree rebalancing in optimal time and space. *Commun. ACM*, 29(9):902–908, Sept. 1986.

[5] A. Trotman, X.-F. Jia, and M. Crane. Towards an efficient and effective search engine. In *SIGIR 2012 Workshop on Open Source Information Retrieval*, pages 40–47, 2012.

[6] H. E. Williams, J. Zobel, and S. Heinz. Self-adjusting trees in practice for large text collections. *Software: Practice and Experience*, 31(10):925–939, 2001.

[7] G. K. Zipf. Human behavior and the principle of least effort. 1949.

[8] J. Zobel, S. Heinz, and H. E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271 – 277, 2001.