Breaking The Hash Table Speed Limit

Optimizing Hash Table Retrieval for Text Vocabularies

Adam Cooper Stephen Duncan Ryan Gregg
Department of Electrical Engineering and Computer Science, The University of Kansas
adam@intellidev.com, jrduncans@screamingspheres.com, ryan@intellidev.com
May 16, 2003

Introduction

Searching large collections of documents using ad hoc queries is becoming increasingly more common, the zenith examples being internet search engines such as Yahoo! and Google. But such techniques are being applied to more than collections of web pages; information retrieval is making inroads into private collections of business documents and even popular desktop applications. Whether a searching application runs on a corporate server or a personal handheld device, some form of an index is always at its core. And given that the index can be used to return acceptably relevant results, the critical factor in index retrieval is speed.

In the majority of search applications, an instant speed gain can be had by keeping the index in memory. And in many such applications the initial delay required to load the index into memory so that the disk never be touched again is appropriate. Zobel, Heinz, and Williams follow this in-memory approach, exploring the potential of several viable candidate data structures for an in-memory index [1]. Their results show the hash table as the clear speed winner. They go on to offer some empirically-backed suggestions for streamlining its implementation. This paper builds on their work, demonstrating that minor implementation changes can make a hash table even faster by virtually eliminating one of the most costly operations in retrieving an entry: the string comparison.

Basic Hashing Process

Hash table speed comes at a price: the assignment of non-unique values to multiple indices in the table (collisions). Typical hash functions are not unique, but *nearly* unique; there will be cases where two different strings generate the same hash code. Moreover, the output of the hash function typically has a range much larger than the allowed indices of the hash table, and a single modulo operation is used to restrict the output of the hash function to a range appropriate to the actual table size:

index = hash_function(key) **modulo** table_size

Both the non-unique quality of the hash function and the memory limitations of the hash table itself contribute to collisions, typically accounted for in one of two ways: by associating more than one entry to a given index, usually though a linked data structure, or by using a known function to compute a new index if one already contains an entry. Zobel, Heinz, and Williams do well to point out the advantages of using a linked data structure within a given index for in-memory vocabularies, their main point being the use of a technique called move-to-front hashing [1].

Using the principle of locality, they suggest that by moving an entry searched for to the front of the list of entries for a given index overall search speed will improve. Over time, the entries more frequently searched for move to the front of their list, and the less frequently searched for entries get pushed to the end. Their empirical results confirm the usefulness of the move-to-front technique, particularly in tables where memory is cramped and collisions are heavy. It is worth noting that our testing does not use move-to-front hashing. In the end what this means is that, if anything, we will be doing more comparisons to find a requested entry and not less. Since our optimization (to be outlined shortly) deals with optimizing these individual entry comparisons, not implementing move-to-front hashing will emphasize the difference in the standard string comparison and our technique. However, implementing move-to-front hashing would not invalidate our optimization. It would, instead, complement it. A fully optimized implementation would do well to adhere to the move-to-front method outlined by Zobel, Heinz, and Williams, and the technique outlined in this paper.

Bottlenecks and Optimizations

There are three core parts to retrieving an entry from a hash table given a key: computing the key's hash code to generate an index into the table, searching all entries corresponding to the calculated index to find the correct entry, and as part of the last step, the actual search method used in an individual test, which is typically a string comparison. Usually the complexity of computing the hash value is some form of O(C), where C is the number of characters in the input string. Finding the correct entry for a group of entries corresponding to a given index is also of linear complexity, O(E), where E is the number of entries corresponding to a given index in the table. And finally, the string comparison is O(K), where K is the number of sequential characters beginning from the first which the two strings in question have in common. A single hash table retrieval involves all three calculations. Improving any one of them should have an impact on retrieval time, proportionate to the time spent in each.

Requesting an entry from the table always requires that we calculate the hash value for that word. As such, optimizing the actual hash function is a must. Given a string with c characters, the typical hash function performs some routine mathematical operation on each individual character. The time taken to compute the hash value for a given string is, in general, $c \cdot o$, where o is the mathematical operation performed on an individual character. Though most retrieval systems set some limit on the length of a string, severe truncation in order to reduce c is not a viable option for optimizing the hash function. Reducing the magnitude of o, however, is crucial. The shorter the per-character mathematical operation takes, the faster we can generate both the hash code and the index into the table. Unfortunately, optimizing o almost inevitably decreases the uniqueness of the resultant hash value. A hash function must be chosen which is both fast and generates acceptable, nearly unique output. If we could further flank this optimization with changes that minimized the delay incurred from searching through multiple, non-unique entries in a given index, we can somewhat attain the best of both worlds: push the hash function to the limit, and come close to eliminating the consequences. In order to do so, we must consider the other two major components of hash table retrieval.

Given that the entry for a given string, if it exists in the vocabulary, can be found at a specific index, we still need to search all the entries of the index. Given an index with e entries (whether we use a chaining system, where multiple entries are literally linked to one index, or a probing system, where entries are technically stored at different indices, but are still associated with a starting index), we must linearly search each entry until we find a match. Thus, there are two immediate ways to optimize the number of comparisons: decrease e, or try to store the entry being searched for at the beginning of the list of entries so it is the only comparison made. Decreasing e can be done immediately by simply increasing the hash table size. On large commercial systems where memory is not a limitation, this is a viable option. On smaller user systems though, memory is often tight and there are significant bounds for the hash table size. In any case, given that our hash table is as large as is feasible, we can further optimize the time it takes to find the entry we are looking for by implementing Zobel, Heinz, and Williams' move-to-front technique discussed earlier. So, in the worst case, we will search through all e entries associated with an index and never find a match for the requested word; in the best case, the first entry of e will be the entry being searched for.

Even in an ideal hash table where every request yields the comparison to a single entry, we still must contend with the actual comparison method. Since a hash function is only nearly unique, and particularly since its output is bounded to produce an index into our hash table, even in the scenario where only one entry is associated with an index we have no way of knowing if it matches the request. The standard solution to this dilemma is to compare the entry word with the requested word and see if they are the same. While this sounds trivial, string comparison is not a cheap operation. Unlike the higher-level linear search involved in finding an entry, where we can stop as soon as a match is found, a string comparison can only stop when a match is *not* found (or when there are no more characters to compare). The closer a word in the vocabulary is to the requested word, the longer it takes for the string comparison to return false. Furthermore, in large collections where there may be a plurality of highly similar words to those searched on, the time spent on the string comparison can rapidly become significant. Zobel, Heinz, and Williams found the standard **`strcmp`** function under both Solaris and Linux to be "highly inefficient;" replacing it with their own custom routine "yielded overall speed improvements of 20% or more." Optimizing the string comparison routine is a step in the right direction, but what if we could do away with it entirely? A total elimination of the string comparison is impossible, but it can be effectively hushed into a corner.

Recall that a hash code is calculated for each word put into the dictionary, and that this value is trimmed via a modulo operation to generate an index into the hash table. For simplicity, we will call the value before the modulo operation the *hash code*, and call the generated index the *trimmed hash code*. Typically the hash code is just treated as an intermediate step to an index, and consequentially thrown away. This is unfortunate, because for the majority of words in a dictionary, their corresponding hash code can uniquely identify them in the collection. Furthermore, any retrieval must generate a hash code for the token request before it can calculate the index into the hash table. Unfortunately, since the hash codes created during indexing are never saved, the hash code calculated for the word being searched on is useless. But it does not have to be.

The basic solution is simple: rather than store only the word itself in the dictionary entry, we also store the hash code. Then, when retrieving from the dictionary, rather than comparing two strings, we retain the hash code of the word being searched on and simply do a single integer compare with the hash code of the entry. In practice there are a few rare cases where this will not work out, so the actual implementation is only slightly more involved.

Eliminating the String Compare

We simply cannot totally eliminate the need for the string comparison. Even with complicated hashing functions that produce highly unique results, there are still cases where two different input strings return the same hash code. However, note that our proposal is not to compare trimmed hash codes (hash table indices), but the full result of the hash function. What this means is that if there are two entries in the dictionary which share the same hash code it is entirely a result of the non-uniqueness of the hash function itself, not the size of the hash table. This is good news, because it means that there will only be a relative handful of entries which share the same hash code. But the fact remains: the possibility of associating two unique words in the dictionary with the same hash code exists. And for that, we need the string comparison.

But we also need one more piece of information: the knowledge that the hash code of a given entry in the dictionary is or is not unique to the vocabulary. Given this, we can know at retrieval time if a single comparison of two hash codes is sufficient or if the more costly string comparison must be done. Simple changes to the dictionary entry format and construction are all that is needed.

Changes in Indexing

For the dictionary entry, we must add two new fields: one for the hash code, and another which tells us if the hash code is associated with only this word in the dictionary (we will term this field isUnique). An illustration of the old and new record implementations follows.

String: word	*other data...*

Original Entry Format

String: word	Integer: hashCode	Boolean: isUnique	*other data...*

New Entry Format

If we use a 32-bit integer to store both the hashCode and isUnique fields, we add 8 bytes to each entry in the dictionary—an acceptable expansion in all but the most memory-constrained environments.

The changes to indexing are equally trivial. Storing the word is the same as before. Storing the hash code only involves keeping a copy of the pre-modulo hash result so we can add it to the entry along with the word itself. The only thing remaining is the isUnique field. At first glance it may seem we would need to do a costly second pass of the dictionary to fill in the isUnique fields. Instead, we can take advantage of the other side of the modulo coin: for any given entry with a hash code *h*, the only other entries which could potentially share *h* are those whose trimmed hash code is also the same. That is, only entries assigned to the same index can possibly share the same hash code. We can easily determine isUnique during indexing simply by doing a check on all other entries currently assigned to the same index. If there does not exist within the index another entry with the new entry's hash code, or if this is the first entry to be added to the index, we set isUnique to true. If there exists within the index another entry whose hash code is the same as the entry we are adding, we set isUnique to false for both the new and already stored entry[1].

It is worth pointing out that this optimization favors, like the move-to-front method of Zobel, Heinz, and Williams, a linked approach to collision resolution. If the hash table implementation uses a probing resolution technique, it is possible to check many more entries in the table than is necessary to determine if the new entry has a unique hash code. In the worst case the hash table is full, we have no way of knowing which entries belong to the index we are trying to add to, and, via the order instructed by the probing function, we must search the entire table! The advantage of the linked approach is that we absolutely define which trimmed hash code (index) a given entry is associated with. And we have a definite terminating point when scanning this list to determine if a new entry's hash code is indeed unique. The bottom line is this: using a linked list to resolve collisions optimizes the calculation of isUnique during indexing, and allows for move-to-front hashing during retrieval: a win-win situation.

Changes in Retrieval

Once the new indexing code is in place, and the new dictionary is ready to be accessed in memory, the changes to the retrieval process are simple.

Given a word *W* to look up, we begin by computing its hash code, and corresponding trimmed hash code (the index into the hash table). Beginning with the first entry *E* in the index, we check the hash code of *W* against the hash code of *E*. If equivalent, then we need know only one more thing—is *E*'s hash code unique to the dictionary? If so, then we have found *W* in the dictionary. But if we fail the hash code test, then we know with certainty that this entry does not correspond to *W*, and so move on to the next available entry in the index. The rare case is when we pass the hash code test but fail the isUnique test. Then we must do a string compare of *W*

[1] Note that there is no need to check for other entries whose isUnique status needs to be cleared, since we never allow two entries to be termed unique and share the same hash code in the first place. The changing of the isUnique field of the already-stored entry is actually only significant the first time it becomes non-unique; any subsequent non-unique additions will simply make an already false isUnique value false once more for whichever existing entry was found.

against the word in *E*. If we pass the string comparison test, then we have found *W* in the dictionary; if not, we move on to the next available entry in the index.

In this system only a rare subset of requests typically needs to execute the string comparison. For all the rest, what used to be a costly string comparison to determine if *E* corresponded to *W* is now only two integer comparisons on success, and one integer comparison on failure.

Implementation

We used Java 1.4.1 to implement both an indexing and retrieval system that take advantage of the hash code optimization as discussed. Like Zobel, Heinz, and Williams, we used different hash functions for both string comparison and hash code comparison retrieval from the dictionary. The following section offers a brief discussion of the three hash functions used.

Hash Functions

Java String Hash Function

Java provides its own routine for getting a hash code from a String object, which made a good candidate for testing. The Java hash is essentially the same as the radix hash function minus the modulo operation (see listing 2).

```
int h = hash;

if (h == 0) {
     int off = offset;
     char val[] = value;
     int len = count;
     for (int i = 0; i < len; i++) {
            h = 31*h + val[off++];
     }
     hash = h;
}

return h;
```

Listing 1: Java's String hash function, © Copyright Sun Microsystems.

Radix and Bitwise Hash Functions

In addition, we ported Zobel, Heinz, and Williams's radix and bitwise hash functions to Java, and used them as further test cases. The radix hash is nearly identical in form to the Java String hash, except for the addition of a modulo operation. Since we do not have direct access to the character array inside the Java String object, some overhead is incurred by a necessary call to String.toCharArray().

```
   // Author J. Zobel, April 2001.
   // Permission to use this code is freely
granted, provided that this statement is
retained.

   int hval = 0;

   int length = keyValue.length();
   char value[] = keyValue.toCharArray();

   for(int i = 0; i < length; i++)
   {
     hval = (SEED*hval + value[i]) % LARGEPRIME;
   }
   return(hval);
```

Listing 2: Java port of J. Zobel's radix hash function.

```
   // Author J. Zobel, April 2001.
   // Permission to use this code is freely
granted, provided that this statement is
retained.

   int h = SEED;

   int length = keyValue.length();
   char value[] = keyValue.toCharArray();

   for(int i = 0; i < length; i++)
   {
       h ^= ((h << 5) + value[i] + (h >> 2));
   }

   return h&0x7fffffff;
```

Listing 3: Java port of J. Zobel's bitwise hash function.

Experiments

Test Data

Our test data was drawn from the TREC project, a subset of data 565 MB in size (WT01-WT06; approximately 21% of the entire WT01-WT28 collection). Our preprocessor collected 489,074 unique terms from the collection, with a sum term frequency of 13,760,483.

Test Environment

All tests were run on a 1000 MHz Intel Pentium III machine with 512 MB of RAM. Tests were executed using Sun's 1.4.1 Java Runtime Environment on Windows XP Professional.

System.currentTimeMillis(), the standard Java timing routine, returns values in intervals of 10 ms on Windows XP—definitely a sub-optimal resolution for profiling our performance. But all is not lost. Vlad Roubtsov has written a publicly available JNI timing class that provides native access to the Windows CPU timing API. Using his timing methods, we were able to realize truly sub-millisecond timing granularities.

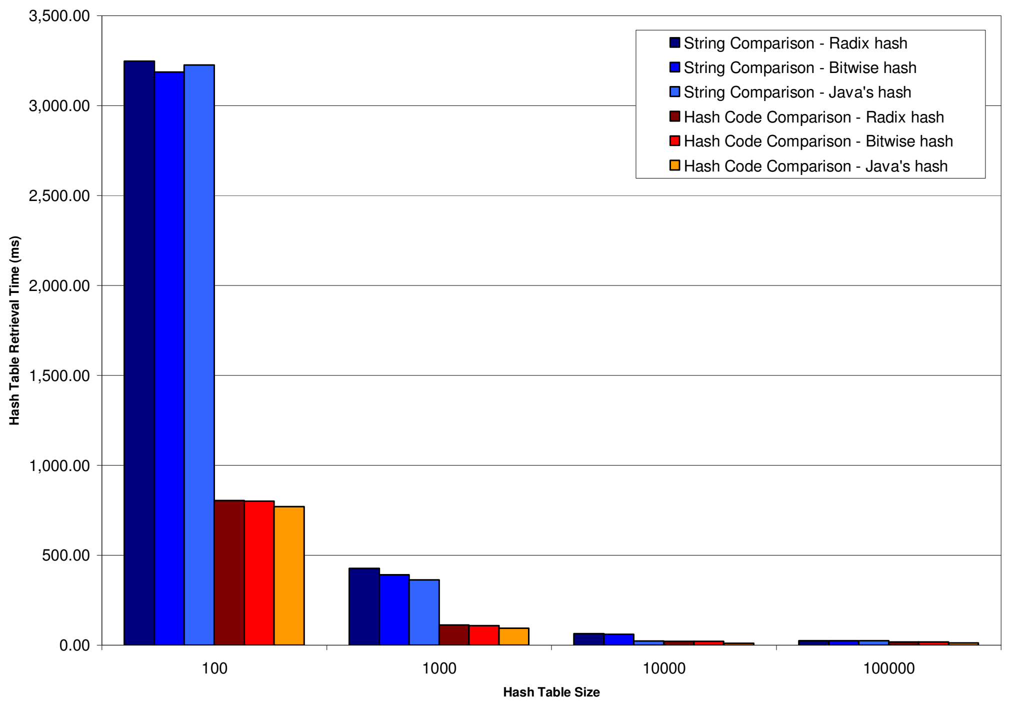


Figure 1: Elapsed times for string comparison versus hash code comparison for three hash functions.

Results

In our experiments we evaluated the typical string comparison implementation against our proposed hash code comparison for all three hash functions. Our measurements include hash table access time and the number of integer and string comparisons made. All timings are in milliseconds, and represent only the time taken to make retrievals from the hash table.

Elapsed Time

Table 1 shows the elapsed times in milliseconds for the standard string comparison implementation. The various hash table sizes—100, 1,000, 10,000, and 100,000—are ordered by column; the three different hash functions, by row. Table 2 uses the same format, and shows the elapsed times resulting from our optimization. Figure 1 is a visual representation of the data in Tables 1 and 2.

	100	**1,000**	**10,000**	**100,000**
Radix	3,248.92	426.668	62.846	25.691
Bitwise	3,187.96	389.713	60.971	25.63
Java	3,226.70	362.733	22.706	24.184

Table 1: Elapsed time (ms) using string comparison.

	100	**1,000**	**10,000**	**100,000**
Radix	803.795	112.013	21.325	15.915
Bitwise	800.011	109.828	21.145	15.781
Java	770.812	94.952	11.394	12.833

Table 2: Elapsed time (ms) using hash code comparison.

The difference is remarkable. However, it does diminish as the hash table size increases. This is expected. As we decrease the number of collisions, the comparisons required to find a correct entry also decrease. But even for the 100,000 size hash table, the hash code comparison did better than the old string compare.

String and Integer Comparisons

Further insight can be gained by observing the number of integer comparisons and string comparisons made. A string comparison involves a single execution of the string comparison routine on a string. An integer comparison is either the comparison of two hash codes, or the testing of the isUnique field.

Tables 3, 4, and 5 show the number of string comparisons required in the non-optimized implementation for each of the three hash functions.

	100	1,000	10,000	100,000
String	2,502,360	258,760	26,912	3,264

Table 3: Radix hash, non-optimized, string comparisons made during retrieval for various hash table sizes.

	100	1,000	10,000	100,000
String	2,431,920	251,992	26,408	3,600

Table 4: Bitwise hash, non-optimized, string comparisons made during retrieval for various hash table sizes.

	100	1,000	10,000	100,000
String	2,413,408	238,960	3,928	1,272

Table 5: Java hash, non-optimized, string comparisons made during retrieval for various hash table sizes.

Before comparing these numbers to the optimized counts, notice how much this tells us about the hash functions themselves. The number of string comparisons varies not only by hash table size, but by hash function. In every case, the Java hash function gets away with making the fewest number of calls to the string comparison routine. What this means is that the Java hash function is looking at fewer entries in the hash table before it finds the correct one. This is best illustrated by the 10,000 size hash table: the bitwise and radix hash functions make almost seven times as many comparisons before finding the correct entry! It is important to remember that the quality of a hash function is more than simply the per character speed it is able to operate at, but the time it saves later due to a good spread of the data. As the data shows, the difference is significant.

When we add the integer comparison optimizations, we see the same differences between the hash functions. Tables 6, 7, and 8 show the number of integer and string comparisons required in the optimized implementation for each of the three hash functions.

The string comparisons have been virtually eliminated. The bitwise hash made only 24 string comparisons; quite an improvement over the previous 3,600 to 2,431,920. The Java hash made 12 (compared to its previous 1,272 to 2,413,408), and the radix hash made none. The claim that the string comparison can be virtually eliminated is well defended.

	100	1000	10000	100000
String	0	0	0	0
Integer	2,646,736	263,184	27,872	4,224

Table 6: Radix hash, optimized, string and integer comparisons made during retrieval for various hash table sizes.

	100	1000	10000	100000
String	24	24	24	24
Integer	2,627,296	262,232	28,008	4,576

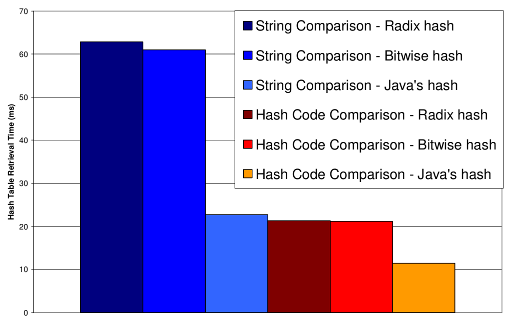
Table 7: Bitwise hash, optimized, string and integer comparisons made during retrieval for various hash table sizes.

	100	1000	10000	100000
String	16	16	16	16
Integer	2,617,808	241,536	4,888	2,232

Table 8: Java hash, optimized, string and integer comparisons made during retrieval for various hash table sizes.

Even more significant is that the proportionate performances of the hash functions in the non-optimized version propagate to the optimized version. But since the comparisons being made now average between only one or two integer comparisons, and very often only a single integer, the execution time drops for all cases.

A good example of this is, again, the 10,000 sized hash table. Consider figure 2, which graphs the elapsed times as before but only for the 10,000 size hash table (cf. Table 1 and 2):



Notice that, for the non-optimized implementation, it would appear that the Java hash function result is a fluke which would fit better in the optimized group. However, notice that the optimized Java hash function does proportionally better than its optimized contenders. This is directly related to the number of compares the Java function has to go through to get to the correct entry (its ability to spread data evenly in the hash table). For a hash table of size 10,000, the Java hash function did the best job of spreading and retrieving data. However, an optimized version of even the "worst" hash function for this table size—the radix hash—performed better than the non-optimized Java hash function. And more, when optimized, the Java function cut its time in half (cf. Tables 1 and 2). Using the hash code comparison nearly eliminates string comparison and significantly improves hash table retrieval time.

Conclusion

Significant hash table retrieval advantages can be had by making a few small changes to eliminate the need for the string comparison in all but the rarest cases.

Additions to the dictionary entry structure include the addition of two fields: the full hash code corresponding to this word in the dictionary, and a boolean indicating whether or not the full hash code uniquely identifies the term in the dictionary.

Additions to the indexing process include only the storing of the full hash code and the calculation and storage of the boolean unique identifier.

Changes to the retrieval process are simply a double integer test—one for the hash code, one for the boolean—before resorting to the string comparison routine. In the vast majority of cases, a single integer test is all that is needed to determine an entry is not what is being searched for, a double integer test is all that is needed to find the correct entry, and the string comparison will never execute.

This optimization can be complimented by the move-to-front technique outlined by Zobel, Heinz, and Williams.

Results are most dramatic for very dense hash tables with a high collision ratio. But whether the target platform is rich or poor in hardware resources, the integer comparison optimization outlined in this paper is useful. For portable devices where memory is cramped, hash tables are likely to be collision heavy, an area where this optimization shines. For heavy-traffic servers where memory is readily available, the fractional speed advantages offered by this optimization can quickly accrue to significance.

Future Work

A native implementation of the retrieval engine in C or C++ would hopefully further establish the results seen so far.

There is great room for more testing. More collections, both larger and more diverse than the sub-collection used, would provide better insight into the applicability of the suggested optimization. In addition to the optimization tested here, we would also like to implement move-to-front hashing and compare hash table performance with and without both of these optimizations.

Acknowledgements

We thank Dr. Susan Gauch and Devanand Ravindran. This research was supported by The University of Kansas.

References

[1] J. Zobel, S. Heinz, and H.E. Williams. In-memory Hash Tables for Accumulating Text Vocabularies. *Information Processing Letter*, 80(6): 271-277, 2001.