Rescore in a Flash: Compact, Cache Efficient Hashing Data Structures for N-gram Language Models

Grant P. Strimel, Ariya Rastrow, Gautam Tiwari, Adrien Piérard, Jon Webb

Amazon.com

{gsstrime, arastrow, tgautam, pierarda, webbajon}@amazon.com

Abstract

We introduce DashHashLM, an efficient data structure that stores an n-gram language model compactly while making minimal trade-offs on runtime lookup latency. The data structure implements a finite state transducer with a lossless structural compression approach. The authors extend the optimization's first-pass decoding, further justifying the need for efficient search and direction assignments. Specifically, we show that with roughly a 10% increase in memory size, compared to a highly optimized, compressed baseline n-gram representation, our proposed data structure can achieve up to a 6x query speedup.

Index Terms: n-gram language models, compact data structures, LM compression

1. Introduction

N-gram language models (LMs) are an essential component of most real-time speech recognition and understanding systems. It is widely accepted that larger n-gram models can achieve better accuracy, resulting in the desire to deploy these models at runtime. However, these LMs are required to be fast for real-time applications and memory efficient to reduce hardware costs. For hybrid Hidden Markov Model (HMM)-based speech recognition systems coupled with faster, smaller acoustic models supported by neural accelerator hardware, n-gram LMs can be a major bottleneck on overall latency, wherein hundreds of thousands of lookups are performed during decoding of a typical spoken utterance. Furthermore, as inference engines move to edge devices where resources are at a premium, keeping the footprint small is even more critical.

Neural LMs [1] are shown to outperform conventional n-gram models. Yet, due to the excessive computation burden, the common approach is to exploit neural LMs in a second-pass resoring framework [2]. Alternatively, there are methods to convert neural LMs into classic n-gram representations [3, 4] which can then be directly incorporated into speech recognition’s first-pass decoding, further justifying the need for effective n-gram LM compression methods.

A variety of techniques for language model compression have been proposed, each making specific trade-offs to balance speed, size and accuracy. The compression schemes proposed by [5, 6] are based on (reverse) trie structures. The representation assigns an implicitly-coded context at each node and uses sorted word labels and logical pointers for search and direction to child nodes during a query. The authors extend the optimization by applying variable-length compression on weights and block compression on key/value arrays. Aside from requiring a sequence of potentially costly binary searches in a block compressed array to navigate the trie, the technique overall yields a significantly smaller LM than naive implementations. To reduce the query time for tries, [6] recommends adjustments to the approach for faster evaluation on “rolling queries”, while the KenLM library [7] provides an optimized implementation that produces strong benchmark results on translation tasks.

A related approach described in [8–10] also exploits tree-like structures in a similar manner to the trie method. In these papers, however, the authors work to compress finite state transducers (FSTs) by decomposing the graph into a tree of contexts (with backoff weights) and a labeled arc array. The tree topology is compressed with the LOUDS data structure [11] which provides a method for indexing into a subsection of the arcs array where a binary search across labels occurs. Like other methods mentioned previously, [8] also suggests compressing the labels and values using variable-length and block coding schemes. While very compact, these data structures can be hindered by query costs. For example, moving between two full n-gram contexts requires one to navigate the LOUDS tree from leaf to root and back to the new leaf with each link in the traversal requiring multiple memory accesses. Nonetheless, the LOUDS approach is comparable to our algorithm presented here since both implement the FST interface in a succinct manner.

An alternative strategy to reduce the LM footprint is to permit lossy compression [12, 13]. The techniques of [13] particularly deliver impressive compression results. The ideas employed in these works are essentially variations of Bloomier filters [14] with the authors’ best variation furthering compression by successive perfect hashing. The approaches are lossy in the sense that there is no definitive way to recover n-grams present in the models once constructed. The algorithms provide a correct value for queries which are represented in the model but emit false-positive errors on queries not in the model. One can directly control the trade-off between LM size and accuracy by scaling the false-positive error rate. Like [13] we also exploit perfect hashing but apply it in a lossless fashion.

Overall, we view our work as complementary to this literature with ours having a particular focus on limiting expected memory accesses during queries while minimizing the overall memory footprint consumed. The understanding that cache misses are latency expensive presents a steep challenge in designing data structures that are both small and fast; however, the cache-conscious mentality gives the opportunity to design algorithms which greatly exploit spacial locality. Informed by these constraints, we present the DashHashLM, a new n-gram LM data structure design. The algorithm is named for its speed and its reliance on hashing methods. Our contributions include the introduction of several optimizations to the LM compression literature whose applicability is not limited to the DashHashLM.
including an ultra-fast perfect hash, search-type pivoting coupled with high-load hashing, and specialized offset quantization.

2. DashHashLM Data Structure

Generally, $n$-gram LMs estimate the probability of a particular word $w_i$ following a partial history of preceding words (a.k.a. context) $w_{i-n+1}, \ldots, w_{i-1}$:

$$P(w_i|w_{i-n+1}, \ldots, w_{i-1})$$

For notational convenience we will denote this expression generically as $v = P(w_i|w)$ where $w$ is the context tuple, $w_i$ is the queried future word and $v$ is the corresponding probability value. A common way to represent an $n$-gram LM is a weighted finite state transducer. Each state within the $n$-gram FST represents a partial history $w$. Each arc emitting from a state has a label and weight representing a future word $w_i$ and its corresponding value $v = P(w_i|w)$. Traversing an arc leads one to a new state which adds the arc label $w_i$ to the partial history (except for backoff arcs which reduce the order of the context and encode a backoff penalty value).

There are three primary ingredients to our DashHashLM data structure. First, we observe that since each state represents a unique history we can use this as a key to construct a minimal perfect hash function that assigns state ids $[0, n_s - 1]$ where $n_s$ is the number of states in the FST. Hence, we need only store the hash function and can compute state ids lazily on-the-fly rather than storing next state labels on each arc explicitly.

Second, DashHashLM stores its future word-value maps in a compact fashion. As others have noted prior [6, 7], a trade-off is typically made between space and lookup time when implementing maps. The two common map implementations are sorted arrays and hash maps. Sorted arrays are maximally space-efficient but require binary search which can be slow for large maps. Meanwhile, hash tables allow fast, constant time lookup but at the price of memory overhead, usually an additional 30-50% for the most common approaches. We strike a balance between storage and speed by employing a hybrid data structure utilizing both. We apply binary search for states with less than $C$ (e.g. 32 or 64) outgoing arcs, while using a high-load hash table for all other high-degree states. DashHashLM stores each map continguously into a single array and pivots its search algorithm based on the size of the subarray associated with the current state.

The last optimization concerns minimizing the size consumed by an offset index array. The offset index array acts as an ordered array of logical pointers that designate the boundaries of the arcs map for a state. Instead of storing this array naively, DashHashLM saves space by a novel technique for quantizing these offsets. We detail this approach further in Section 5.

Figure 1 shows the DashHashLM query procedure. Given a history $w$ and future $w_i$, the algorithm operates as follows:

1. Use a minimal perfect hash function $h^*$ to lookup $w$’s state id $h^*(w)$.
2. In an offset index array $I$, lookup $l[h^*(w)]$ and $l[h^*(w) + 1]$, the subarray boundaries for outgoing arcs from $w$.
3. If $l[h^*(w) + 1] - l[h^*(w)] \leq C$, perform binary search within packed arc array $A$ on entries $l[h^*(w)]$ through $l[h^*(w) + 1]$ to lookup the weight for $w_i$. Otherwise, conduct a hash lookup on the subarray. If $w_i$ is not present, return the backoff value for the state $B[h^*(w)]$.
4. (Optional) Use $w$ and $w_i$ (or backoff) to determine the next state history $w'$, Return to 1.

The remaining of the paper is organized as follows. Sections 3, 4 and 5 detail each of the data structure’s major components along with the design choices and alternatives considered. Section 6 does a comparative analysis of DashHashLM against other compressed structures on a speech recognition task.

3. Perfect Hashing

In perfect hashing, a function maps a set $S$ of $n_s$ keys into $m$ buckets with no collisions. A Minimal Perfect Hash Function (MPHF) hashes $S$ into $n_s$ buckets with no collisions. DashHashLM constructs a MPHF on the set of partial histories of the LM to assign each one a unique state id. There is a rich literature on construction methods for MPHFs [15–22] where the goal of most schemes is to build the smallest possible data structure that performs minimal perfect hashing while retaining $O(1)$ evaluation time. The DashHashLM data structure uses the Fox-Chen-Heath (FCH) [23] construction because it has an especially fast evaluation time [19] (our implementation requires a single memory access). We briefly outline the construction below.

The FCH algorithm requires a family of seeded hash functions. In practice, one can use the 32-bit variants of the Fowler-Noll-Vo Hash [24], MurmurHash [25], SpookyHash [26] or CityHash [27]. We seed three different functions, $h_1$, $h_2$ and $h_3$ for construction.
1. Hash each key with \( h_1 \) and find a threshold \( T_1 \) which produces a 60-40 split into two sets, \( S_1 = \{ \text{w} \in S | h_1(\text{w}) \leq T_1 \} \) and \( S_2 = \{ \text{w} \in S | h_1(\text{w}) > T_1 \} \) where \( |S_1| = 0.6n_s \) and \( |S_2| = 0.4n_s \).

2. Allocate \( m = c n_s / \log n_s \) total buckets for key assignment. \( \text{w} \in S_1 \) gets assigned bucket to \( h_2(\text{w}) \mod T_2 \), while \( \text{w} \in S_2 \) is assigned to bucket \( T_2 + (h_2(\text{w}) \mod (m - T_2)) \) where \( T_2 = 0.3m \).

3. Sort the buckets by size.

4. Working on each bucket from highest cardinality to lowest, use a third hash function \( h_3 \) to assign the bucket a “bit pattern” by hashing each key of the bucket into \( n_s \) slots on range \([0, n_s - 1]\). If the bit pattern fits (i.e. there are no collisions with previously assigned slots), mark the corresponding slots as now occupied. Otherwise, shift the pattern one slot (for all keys in the bucket) to the right and check again.

5. Once a shift amount that fits is found, record the bucket’s shift value, then move to the next bucket and run the search in step (4) for it.

The bucket shift values are what the MPH stores. Since the maximum shift amount is \( n_s \), only \( \log n_s \) bits need to be stored for each bucket yielding a structure of size approximately \( n_s \). bits. To compute the perfect hash of a key \( \text{w} \) at runtime, the scheme performs just one memory access to determine the shift value of \( \text{w} \)’s bucket (the bucket being computed from just \( h_1(\text{w}) \) and \( h_2(\text{w}) \)), and returns \( h_3(\text{w}) + \text{shift mod } n_s \) as its unique id. The seemingly arbitrary scalars (e.g. 0.6, 0.3) for construction are chosen as an inexpensive way to achieve a favorably-skewed bucket size distribution. Selecting the right constant \( c \) (within range \([2, 3]\)) gives a high likelihood for success on construction; otherwise, one should reseed the hash functions and try again.

For most applications, the FCH scheme has fallen out of fashion because modern techniques produce a smaller footprint. For example, the Hash, Displace and Compress method [20] achieves nearly 2 bits per key while most recently, RecSplit [22] achieves 1.56 bits per key (close to the \( \log e \approx 1.44 \) lower bound). Both techniques require \( \sim n_s \) memory accesses for evaluation but are practically fast enough for most applications. More critically, however, FCH can be grindingly slow to construct on large key sets, exhibiting super-linear asymptotic behavior (or has a prohibitively large constant factor) with little multi-threading opportunity, unlike RecSplit which has high parallelizability. In order to construct an FCH hash function over 50+ million partial histories within a reasonable time period, we engineered several improvements for our construction implementation. For example, we employ a technique we call “look-ahead galloping” where at each location of a bit pattern the implementation looks ahead many shifts (say 64 or 128) at a time then aggregates the results across before searching further. This is far more cache efficient for the search component, significantly reducing construction time. Additionally, when the search gets to a point where only singleton buckets remain, we simply create a queue of the final free slots to assign to each bucket, instead of using a linear search.

4. High-Load Hash Tables

The DashHashLM pivots between two storage and search techniques based on the outdegree of the state. For states with fewer than \( C \) arcs, the word-value entries are packed in key sorted order and binary search is used for future word lookup. The sorted array implementation keeps space overhead to a minimum and retains fast lookup because of the locality during search over small sets. However, for high-degree states, DashHashLM’s design employs a recent variant on Cuckoo hashing [28], known as Horton hashing [29], to achieve minimal space overhead.

Like Cuckoo, Horton hashing operates with multiple hash functions, a primary hash function \( h_1 \) and secondary hash function \( h_2 \). The hash table is divided into buckets of capacity \( L \) word-value entries. As with classic Cuckoo hashing, entries are stored either at their bucket hashed to with \( h_1 \) or with \( h_2 \). Buckets come in two varieties, Type A and Type B. Type A buckets are those that have not overflowed, i.e. \( \leq L \) attempted placements. Type B buckets, on the other hand, have overflowed and as a result Horton hashing stores only \( L - 1 \) entries in them while converting the last slot space to a remapping filter \( F \). The remapping filter is an array of seed values (2 or 3 bits each) for hashing with \( h_2 \) to a new bucket for those overflowing keys. At the same time, the filter can aid in determining whether or not a key is present in the table.

To search for a key \( w_1 \), the bucket \( h_1(w_1) \) is first accessed. If \( w_1 \) is found in the bucket, the associated value is returned. Otherwise, if the bucket is a Type A bucket, then \( w_1 \) is not in the table. If the bucket is a Type B bucket, then the filter is used to set \( \text{seed} = F[h_{\text{tag}}(w)] \). If \( \text{seed} = 0 \), then \( w_1 \) is not in the table. Otherwise, the bucket \( h_2(w_1, \text{seed}) \) is searched. At this point, \( w_1 \) must be present in its secondary bucket or else it is not in the table. Figure 2 illustrates a Horton hashing lookup.

There are a few notable aspects of this hashing method. First, \( L \) is chosen to approximately fit a bucket in a single 1.1 cache line (DashHashLM uses \( L = 8 \)) and since a key must be its primary or secondary bucket there are at most two memory accesses during search. Second, the table is constructed so as many keys as possible are placed in their primary buckets. While we omit here the algorithmic details for packing a Horton hash table, for the static setting one can fill a table to achieve overhead of less than 5% (table is 95% full) with expected bucket lookups of 1.18 and 1.06 for positive (key is present) and negative (not present) queries, respectively, by leveraging the remapping filter [29]. Last, for the overhead ratio to be realized, the number of elements stored must reach a certain minimum threshold, which is why DashHashLM resorts to using it only for large tables.

Practical alternative hashing techniques like quadratic prob-
ing, Robin Hood hashing [30], and Hopscotch hashing [31] all achieve relatively fast lookup but typically do not reach loads above 90%. Sparse implementations, like that of Google SparseHash [32], use a dense array to store initial entries while using a backing bitmap to mark off occupied buckets. This implementation boasts an impressive 2 bit overhead per entry; however, it requires at least 2 memory accesses per lookup. Furthermore, our simple Horton implementation is 25% - 40% faster for positive lookups than that of sparse maps.

5. Quantized Offsets

The last optimization DashHashLM makes is to compress the offset index array \( I \). Especially when the average outdegree of a state is low, the overhead for naively storing the offsets array increases since \( \sim \log_2(n_a) \) bits are needed for each value in the array where \( n_a \) is the number of arcs in the FST. This reality can account for at least 25% of the total DashHashLM size. However, since \( I \) is a sequence of non-decreasing integers, the array can be compressed and retain constant time random access with Elias-Fano coding [33–35] or via methods used in other LM compression approaches [36]. However, though Elias-Fano coding efficiently compresses \( I \), it requires 3 memory accesses to retrieve one of \( I \)'s values. DashHashLM implements an Elias-Fano variant, but we also propose here an alternative method which is not as space-efficient but requires only a single memory access to retrieve a value.

We break the array of offsets into blocks. Each block contains a base value and then encodes the differences (deltas) between subsequent values. To look up a value at an index, one first accesses its block's base value and then cumulatively sums the deltas within the block up to the desired index. We use 8-bit bytes to code 28 deltas within a convenient block of size 29 entries. Four bytes are used for the base value and the next 28 bytes encode the delta values. If a delta value is less than 128 we use the value explicitly. Otherwise, we use an exception array \( E \) and lookup value \( E[\delta - 128] \) as the true delta value.

This gives us the ability to store a limited set of 128 exception values. To make this work for our FSTs, we strategically add null arcs to the FST. These only serve to make the outdegrees of each state a more convenient value. Formally, for those states with degree more than 128, we wish to round each of them up to one of 128 strategically chosen degree values. Now, the goal is to choose these 128 "round up" values so that the minimum number of null arcs are added to the FST. We use a dynamic programming solution to solve for these values. We define a subproblem as the optimal placement of \( k \) roundup points \( r_1, \ldots, r_k \) for outdegrees \( d_1, \ldots, d_n \) where \( d_1, \ldots, d_n \) is the sorted list of outdegrees greater than 128. It should be apparent that for each optimal roundup point \( r \) we have \( r \in \{ d_1, \ldots, d_n \} \). Given this, our memoized optimal subproblem structure can be written recursively as

\[
M[k, i] = \min_{\ell \leq n \leq 2^i \epsilon^i} \left\{ M[k - 1, \ell + 1] + \sum_{j=1}^{i}(d_j - d_j) \right\}
\]

The dynamic program executes in polynomial time and much of the computation can be executed in parallel with each entry of a row being computed independently. Notice this technique grows \( A \) in order to compress \( I \). For practical LMs with which we experimented (see Section 6), we observe the resulting null arcs add less than 0.8% overhead to \( A \) while reducing

\[ I \] by \( \sim 75\% \) since only 8.83 bits per offset are used on average. Furthermore, to retrieve a value in \( I \), it requires only one memory access since a block consumes 32 bytes, easily fitting within an L1 cache line, and with only 128 entries, \( E \) is small enough to be readily available in the rare instances it is needed.

6. Empirical Results

We study empirically the performance of the DashHashLM by measuring memory consumption and conducting latency tests for an automatic speech recognition task. We experiment on 1000 voice assistant-like utterances with more than 100 million LM rescoring queries using a Kaldi BigLM Faster Decoder-style decoder [38] on a machine with 2.4 GHz Intel Xeon E5-2676 v3 processors. Our \( n \)-gram LM structure was built from a 4-gram backoff language model of approximately 145 million unique \( n \)-grams (FST non-backoff arcs) and 30 million partial histories (FST states).

We experimented against LM data structures comparable to ours, namely the LOUDS implementation of [9] which also implements an FST and the lossy S-MPHR [13] which utilizes perfect hashing. For all implementations weights were quantized to 12 bits. The results are summarized in Table 1.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Size (MB)</th>
<th>Queries/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOUDS FST</td>
<td>599</td>
<td>866</td>
</tr>
<tr>
<td>S-MPHR(^a)</td>
<td>486</td>
<td>1105</td>
</tr>
<tr>
<td>DashHashLM(^5)</td>
<td>824</td>
<td>6217</td>
</tr>
<tr>
<td>DashHashLM (EF)</td>
<td>640</td>
<td>3724</td>
</tr>
<tr>
<td>DashHashLM (QO)</td>
<td>658</td>
<td>5323</td>
</tr>
</tbody>
</table>

Table 1: Speed and memory use of the rescoring task with 12-bit weight quantization.

For our setup, DashHashLM demonstrates a significant speed improvement over its competitors while retaining a comparable memory compression. We also note using our specialized quantized offset index representation adds a marginal increase in space over Elias-Fano coding, but for this task it achieves a compelling boost in query speed. An interesting observation is the lossy S-MPHR implementation was measured at a slower query time despite its relatively simple construction. This result is likely due to its use of a slower perfect hash scheme which not only needs to be computed for each query but forces jumps to random memory locations on each query within sequences of matching contexts. In contrast, DashHashLM requires a single perfect hash computation for a partial history and retains at least some degree of spatial locality for batch queries at the context.

7. Conclusion

We presented the DashHashLM data structure for representing \( n \)-gram language models. The data structure proves to be effective at compressing language models while achieving a high degree of lookup speed. To accomplish the footprint reduction, DashHashLM introduced several new techniques. In future efforts, it would be interesting to see if the ideas employed by DashHashLM can also be applied to the other existing LM compression approaches.

\(^a\) Uses a best-effort replica implementation with 8-bit fingerprints.
\(^5\) No offset compression.
\(^b\) Uses Elias-Fano offset compression.
\(^7\) Uses custom Quantized Offset approach.
8. References


