# Space-Efficient Data Structures for Top-$k$ Completion

Bo-June (Paul) Hsu
Microsoft Research
One Microsoft Way, Redmond, WA, 98052 USA
paulhsu@microsoft.com

Giuseppe Ottaviano
Dipartimento di Informatica
Università di Pisa
ottavian@di.unipi.it

## ABSTRACT

Virtually every modern search application, either desktop, web, or mobile, features some kind of query auto-completion. In its basic form, the problem consists in retrieving from a string set a small number of completions, i.e. strings beginning with a given prefix, that have the highest scores according to some static ranking. In this paper, we focus on the case where the string set is so large that compression is needed to fit the data structure in memory. This is a compelling case for web search engines and social networks, where it is necessary to index hundreds of millions of distinct queries to guarantee a reasonable coverage; and for mobile devices, where the amount of memory is limited.

We present three different trie-based data structures to address this problem, each one with different space/time/complexity trade-offs. Experiments on large-scale datasets show that it is possible to compress the string sets, including the scores, down to spaces competitive with the `gzip`'ed data, while supporting efficient retrieval of completions at about a microsecond per completion.

## Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; E.1 [**Data Structures**]: Trees

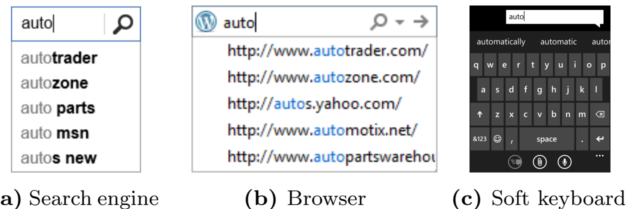
## Keywords

Top-k completion, Scored string sets, Tries, Compression

## 1. INTRODUCTION

Auto-completion is an important feature for modern search engines, social networking sites, mobile text entry, and many web and database applications [35, 23, 16]. Specifically, as the user enters a phrase one character at a time, the system presents the top-$k$ completion suggestions to speed up text entry, correct spelling mistakes, and help users formulate their intent. As shown in Figure 1, a search engine may suggest query completions of search prefixes, a browser may complete partial URLs, and a soft keyboard may predict word completions. Typically, the completion suggestions are drawn from a set of strings, each associated with a score. We call such a set a *scored string set*.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.
*WWW 2013*, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.




**(a)** Search engine **(b)** Browser **(c)** Soft keyboard

**Figure 1:** Usage scenarios of top-$k$ completion.

Definition 1.1 (Scored string set). *A* scored string *set $\mathcal{S}$, $|\mathcal{S}| = n$, is a set of $n$ pairs $(s, r)$ where $s \in \Sigma^*$ is a string drawn from an alphabet $\Sigma$ and $r$ is an integer* score.

Given a prefix string, the goal is to return the $k$ strings matching the prefix with the highest scores. Formally, we define the problem of top-$k$ completion as follows.

Problem 1.2 (Top-$k$ completion). *Given a string $p \in \Sigma^*$ and an integer $k$, a* top-$k$ completion *query in the scored string set $\mathcal{S}$ returns the $k$ highest scored pairs in $\mathcal{S}_p = \{(s, r) \in \mathcal{S} \mid p \text{ is a prefix of } s\}$ (or the whole set if $|\mathcal{S}_p| < k$).*

To be effective, an auto-completion system needs to be responsive, since users expect instantaneous suggestions as they type. As each keystroke triggers a request, the system needs to scale to handle the high volume. To host a large number of unique suggestions, the data should be compressed to avoid the latency costs associated with external memory access or distributed data structures. If the data needs to be hosted on mobile clients, the compression should further scale across dataset sizes.

A simple solution is to store all the strings in a trie or compacted trie [21], and associate each leaf node with its corresponding score. Although such a data structure is compact and allows us to quickly enumerate all the strings matching a given prefix, we need to explicitly sort the matches by their scores in order to return the top-$k$ completions. For large string sets where short prefixes may potentially match millions of strings, this approach is prohibitive in terms of speed. Although we can precompute and store the top-$k$ completions for the short prefixes [24], this requires a priori knowledge of $k$ and the space scales poorly with $k$.

Many of the top-$k$ completion application scenarios exhibit special properties which we can take advantage of to improve the space and time efficiency of the system. First, the scores associated with the strings often exhibit a skewed power law distribution, as demonstrated by the histogram of search

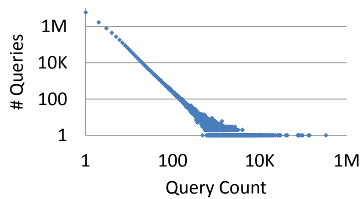


**Figure 2:** Score distribution in the AOL dataset.

counts associated with the AOL queries [1] in Figure 2. Most of the queries have low counts as scores that require only a few bits to encode. Second, the distribution of the target strings that users enter one character at a time often approximates the distribution of the scores, after ignoring the prefixes not matching any of the strings in the set. Specifically, in practical usages of top-$k$ completion systems, prefixes of entries with higher scores tend to be queried more than those associated with lower scored entries. In fact, a common folklore optimization in practical trie implementations is to sort the children of each node by decreasing score. Third, as a large number of strings share common prefixes, they are highly compressible.

In this work, we present three data structures that exploit the aforementioned properties to support efficient top-$k$ completion queries with different space/time/complexity trade-offs.

- Completion Trie: A compact data structure based on compressed compacted trie, where the children of each node are ordered by the highest score among their respective descendants. By storing the max score at each node, we can efficiently enumerate the completions of a string prefix in score order. This data structure uses standard compression techniques, such as variable-length encoding, to reduce space occupancy.

- RMQ Trie: A generic scheme that can be applied to any data structure that bijectively maps a set of strings to consecutive integers in lexicographic order, by using a Range Minimum Query (RMQ) data structure [13] on the sequence of scores to retrieve the top-$k$ completions. In our experiments, we apply the scheme to the lexicographic path-decomposed trie of [17].

- Score-Decomposed Trie: A compressed data structure derived from the path-decomposed trie of [17], where we use a path decomposition based on the maximum descendant score. This path decomposition enables efficient top-$k$ completion queries.

Large scale evaluations on search queries, web URLs, and English words demonstrate the effectiveness of the proposed approaches. For example, on the AOL query log with 10M unique queries [1], the Completion Trie achieves a size of 120 bits/query (including the scores) while requiring an average of only 3.7$\mu$s to compute the top-10 completion on a simulated workload. In comparison, the Score-Decomposed Trie increases the completion time to 8.0$\mu$s, but further reduces the size to 62 bits/query. In fact, this is less than 30% of the uncompressed data size and within 11% of the `gzip`'ed size. The RMQ Trie obtains a similar space occupancy at 65 bits/query, but is significantly slower at 33.9$\mu$s.

## 2. RELATED WORK

There is a vast literature on ranked retrieval, both in the classical and succinct settings. We report here the results closest to our work.

Using classical data structures, various studies have examined the task of word/phrase completion [7, 26, 24, 25, 30, 36], though most do not consider datasets of more than a million strings or explore efficient algorithms on compressed data structures. In [24], Li et al. precompute and materialize the top-$k$ completions of each possible word prefix and store them with each internal node of a trie. This requires a predetermined $k$ and is space inefficient. Church et al. employ a kd-tree style suffix array that alternates the sorting order of nodes between lexicographic and score order at each level [7]. However, the lookup time is in the worst case $O(\sqrt{n})$ and has empirical performance in milliseconds. Recently, Matani [26] describes an index similar in principle to the proposed RMQ Trie structure in Section 5, but uses a suboptimal data structure to perform RMQ. Although the system achieves sub-millisecond performance, both this and the previous work require storing the original string set in addition to the index.

From a theoretical point of view, Bialynicka-Birula and Grossi [4] introduce the notion of *rank-sensitive* data structures, and present a generic framework to support ranked retrieval in range-reporting data structures, such as suffix trees and tries. However, the space overhead is superlinear, which makes it impractical for our purposes.

As the strings are often highly compressible, we would like data structures that approach the theoretic lower bound in terms of space. Succinct data structures use space that is the information-theoretically optimal number of bits required to encode the input plus second-order terms, while supporting operations in time equal or close to that of the best known classical data structures [20, 28, 3, 33]. Recent advances have yielded many implementations of string dictionaries based on succinct data structure primitives [17, 6], without scores.

Hon et al. [19] use a combination of compressed suffix arrays [18, 12] and RMQ data structures to answer *top-$k$ document retrieval* queries, which ask for the $k$ highest-scored documents that contain the queried pattern as a *substring*, in compressed space. While this is strictly more powerful than top-$k$ completion, as shown in [6], string dictionaries based on compressed suffix arrays are significantly slower than prefix-based data structures such as front-coding, which in turn is about as fast as compressed tries [17]. The RMQ Trie of Section 5 uses a similar approach as [19], but is based on a trie instead of a suffix array. As we will discuss in Section 8.4, speed is crucial when implementing more sophisticated algorithms, such as fuzzy completion, on top of the core top-$k$ completion data structures.

## 3. PRELIMINARIES

In this section, we briefly describe some of the data structures and primitives used in this paper. For additional details on the design and implementation of these primitives, please refer to the cited references.

**String Dictionaries.** A *string dictionary* is a data structure that maps a prefix-free set $\mathcal{S} \subset \Sigma^*$ of strings drawn from an alphabet $\Sigma$ bijectively into $[0, |\mathcal{S}|)$, where prefix-free means that no string in the set is a prefix of another string in the set; this can be guaranteed, for example, by appending a special terminating null character to every string. We call

Lookup the function that maps a string to its *index*, and the inverse function Access, i.e. Access(Lookup($s$)) $= s$ for all $s \in \mathcal{S}$. Lookup($s$) returns $\bot$ if $s$ is not in $\mathcal{S}$. A popular way of implementing a string dictionary is by using a trie data structure [14], possibly *compacted*, where each chain of edges without branches is collapsed into a single edge.

**Priority queues.** A *priority queue* $Q$ maintains a set under operations Push($Q, v$), which adds the element $v$ to $Q$; and Pop($Q$), which returns the minimum element in $Q$ according to a given total ordering on the values, and removes it from the set. To implement priority queues, we use a classical binary heap [21]. While alternative solutions, such as Fibonacci heaps and pairing heaps, have O(1) amortized insertion cost, they are often slower than binary heaps in practice.

**Bitvectors with Rank and Select.** Given a bitvector $X$ with $n$ bits, we can define the following operations: $\text{Rank}_b(i)$ returns the number of occurrences of bit value $b \in \{0, 1\}$ in $X$ in the range $[0, i)$. $\text{Select}_b(i)$ returns the position of the $i$-th occurrence of bit value $b$ in $X$. Note that $\text{Rank}_b(\text{Select}_b(i)) = i$. These operations can be supported in constant time by adding $o(n)$ bits of redundancy to the bitvector [8, 20]. In our implementations we use the `rank9` data structure [37] and a variation of the `darray` [31] when only Select is needed.

**Balanced parentheses (BP).** In a sequence of $n$ balanced parentheses, each open parenthesis `(` is paired with its *mate* close parenthesis `)`. Operations FindClose and FindOpen find the position of the mate of an open and close parenthesis, respectively. The sequence can be encoded as a bitvector, where `1` represents `(` and `0` represents `)`. The difference between the number of open and close parentheses in the range $[0, i)$ is called the *excess* at $i$. Note that $\text{Excess}(i) = 2\,\text{Rank}_{(}(i) - i$. It is possible to support the above operations in constant time with a data structure that takes $o(n)$ bits [20, 28, 2, 33]. In our implementation we use the Range-Min tree [17], a variation of the Range-Min-Max tree [2, 33].

**DFUDS representation.** The DFUDS (depth-first unary degree sequence) representation [3] maps a tree with $t$ nodes to a BP sequence of $2t$ bits; several traversal operations can be implemented with a combination of Rank, Select, FindClose, and FindOpen operations.

**Range Minimum Queries (RMQ).** Given an array $A$ of $n$ values, the operation RMQ($i, j$) returns the position of the minimum value of $A$ in the range $[i, j]$, according to a given total ordering of the values (in case of ties, the leftmost value is chosen). RMQ can be supported in constant time by pre-computing the *Cartesian tree* of $A$, which can be encoded using BP into $2n + o(n)$ bits [13]. In our implementation we use this data structure with a slight variation in the RMQ algorithm, described in more detail in Appendix A.

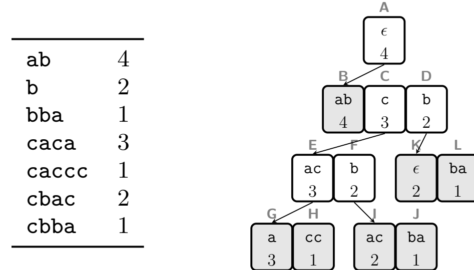
**Implementation details.** In implementing the succinct data structures described above, we are mostly concerned with the actual speed and space of the data structures we consider, rather than theoretical optimality. For this reason, although constant-time implementations of many succinct primitives are available, we often prefer logarithmic-time versions. As shown in several papers [31, 37, 2, 17], such implementations are actually *faster* and *smaller* than their constant-time counterparts. For this reason, when reporting time complexities, we will ignore the logarithmic factors introduced by succinct operations, treating them as constant-time; in this case we will use the $\tilde{O}$ notation to avoid ambiguity.

Our implementations of these structures are freely available as part of the *Succinct* C++ library [34].

# 4. COMPLETION TRIE

A trie, or prefix tree, is a tree data structure that encodes a set of strings, represented by concatenating the characters of the edges along the path from the root node to each corresponding leaf. We collapse common prefixes such that each string prefix corresponds to a unique path. Whereas each edge represents a single character in the simple trie, a compacted trie, also known as a Patricia trie or radix tree, allows a sequence of characters to be associated with each edge such that no node can have a single child (except for the root node in degenerate cases).

To encode the score associated with each string, we assign to each leaf node the score of the string it represents. To support efficient top-$k$ completion, we further assign to each intermediate node the maximum score among its descendant leaf nodes. Note that by construction, the score of each non-leaf node is simply the maximum score among its children, as exemplified in Figure 3. As each node score now represents the largest score among all strings starting with the prefix corresponding to the node, we can apply it as an exact heuristic function in a variation of the A* search algorithm [32] to find the best completion path from a node representing the prefix. Specifically, we first find the *locus node*, the highest node in the trie that matches or extends the prefix string, and insert it into a priority queue, if found. Iteratively, pop the node with the largest score. If it is a leaf node, add the string and score corresponding to the node to the list of completions. Otherwise, iteratively insert its children to the queue until $k$ completions have been found or the priority queue is empty.



| | |
|---|---|
| ab | 4 |
| b | 2 |
| bba | 1 |
| caca | 3 |
| caccc | 1 |
| cbac | 2 |
| cbba | 1 |



**Figure 3:** Compacted trie with max scores in each node.

The worst-case time complexity of this algorithm is $O(|\Sigma||p| + |\Sigma|kl \log |\Sigma|kl)$, where $\Sigma$ is the alphabet from which the strings are composed, $p$ is the input string prefix, $k$ is the number of requested completions, and $l$ is the average length of the completions returned excluding the common prefix $p$. Specifically, we need to examine up to $|p|$ nodes with up to $|\Sigma|$ children each to find the locus node. We may encounter and expand $kl$ nodes on the way to the leaf nodes corresponding to top-$k$ completions. As the algorithm inserts all children of each expanded node to the priority queue, we add up to $|\Sigma|kl$ nodes to the binary heap, contributing an additional $O(|\Sigma|kl \log |\Sigma|kl)$ term.

Instead of inserting all children of each expanded node to the priority queue, if we were to sort the children by order of decreasing score, we only need to add the first child and the next sibling, if any, of each expanded node. Conceptually, we can view this as adding a sorted iterator to the priority queue. Whenever we remove an iterator from the queue, we return the first element and insert the remainder of the iterator back into the queue. With this change, the time complexity

to find the top-$k$ completions reduces to $O(|\Sigma||p| + kl \log kl)$ as we insert a maximum of 2 nodes for each node expanded during the search algorithm. In practice, sorting the children by decreasing score also reduces the number of comparisons needed to find the locus node. A summary of the top-$k$ completion algorithm on the Completion Trie data structure is presented in Algorithm 1.

**Algorithm 1** Top-$k$ completion with Completion Trie.

```
    Input: Completion Trie 𝒯, prefix p, and k ≥ 0
    Output: List c of top-k completions of p
 1  Q ← Empty priority queue
 2  c ← Empty list
 3  n ← FindLocus(𝒯, p)
 4  if n is not null then
 5      Push(Q, (Score(n), n))
 6  while Q is not empty do
 7      r, n ← Pop(Q)
 8      if n is a leaf node then
 9          s ← String corresponding to n
10          Append (s, r) to result list c
11          if |c| = k then return c
12      else
13          fn, nn ← First child of n, next sibling of n
14          Push(Q, (Score(fn), fn))
15          if nn is not null then Push(Q, (Score(nn), nn))
16  return c
```

## 4.1 Compressed Encoding

In addition to improving the theoretical time complexity, improving the locality of memory access also plays a significant role in improving the practical running time, as accessing random data from RAM and hard drive can be 100 and 10M times slower than from the CPU cache, respectively, easily trumping any improvements in time complexity. For example, to improve memory locality when finding the locus node, we store each group of child nodes consecutively such that accessing the next sibling is less likely to incur a cache miss. However, instead of writing each group of sibling nodes in level order, we write the encodings of each group of trie node in depth-first search (DFS) order. As each internal node is assigned the maximum score of its children and the children are sorted by decreasing score, iteratively following the first child is guaranteed to reach a leaf node matching the score of an internal node. Thus, by writing the nodes in depth-first order, we typically incur only one cache miss per completion, resulting in significant speedup over other arrangements.

For each node, we encode the character sequence associated with its incoming edge, its score, whether it is the last sibling, and an offset pointer to its first child, if any. Note that if the node has a next sibling, it is simply the next node. Furthermore, we can use a special value of 0 as the first child offset for leaf nodes. Assuming 4-byte scores and pointers, a naïve encoding would require $(l + 1) + 4 + 1 + 4 = l + 10$ bytes, where $l$ is the length of the character sequence.

One way to reduce the size of each node is to apply a variable-byte encoding to scores and offsets. However, as each group of child nodes are sorted by decreasing order and we traverse the children sequentially, we can first perform delta encoding by storing only the score difference between the current node and its previous sibling. As the first child shares the same score as its parent and is always traversed from its parent node, we can simply store a differential score of 0. Similarly, we observe that the first child offset for siblings can only increase. Thus, we can apply the same delta encoding techniques to derive the first child offset of a node from its previous siblings. To find the first child offset for the first sibling, we can traverse all the remaining siblings and return the next node, as each set of sibling nodes are stored in depth-first order. However, as the number of siblings may be large, we simply store the difference in offset between the first child and the first sibling node. Note that we still encode leaf nodes with a first child offset of 0.

With delta encoding, we significantly reduce the values of the node scores and first child offsets. While many variable-byte encoding schemes exist, we choose to apply an approach where we encode the size of each value in a header block. As smaller values, including 0, are much more frequent than larger values due to the power law distribution of scores and the depth-first ordering of the nodes, we choose to allocate two bits in the header to represent values encoded with 0, 1, 2, or 4 bytes.[1] We further allocate another bit in the header to indicate if the node is the last sibling. Finally, if we limit the maximum number of characters that we can store with each node to 7 by adjusting how the trie is compacted, we can store the length of the character sequence in the remaining 3 bits of a 1 byte header. Figure 4 shows the binary Completion Trie encoding of the example from Figure 3.

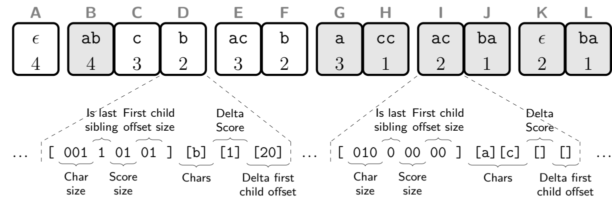



**Figure 4:** Binary Completion Trie encoding.

## 4.2 Implementation Details

As the trie nodes are stored in DFS order, it is possible to reconstruct the string corresponding to a completion leaf node by starting from the root node and iterative finding the child whose subtrie node offset range includes the target leaf node. However, this is an expensive $O(|\Sigma|d)$ operation, where $d$ is the depth of the leaf node. Instead, we can significantly reduce the cost of returning the top-$k$ completion strings through additional bookkeeping in the search algorithm. Specifically, we store the nodes to be inserted into the priority queue in an array, along with the index of its parent node in the array. By modifying the priority queue to access nodes through their corresponding array indices, we can retrieve the path from each completion node to the locus node by following the parent indices. Thus, we can efficiently construct the completion string in time $O(d)$ by concatenating the original prefix string with the character sequences encountered along the reverse path.

To further improve the running time of the algorithm, we employ a few bit manipulation techniques that take ad-

[1]If the dataset calls for scores or first child offsets that cannot be represented in 4 bytes, we can simply change 4 to the number of bytes required to represent the largest value.

vantage of our particular encoding scheme. With standard variable-byte encoding [38], we need to read multiple bytes to determine the size and decode the value. But by storing the size of the variable-byte value in a 2-bit code, we can determine the size $\ell$ by looking up the code $c$ in a small array: $\ell \leftarrow \text{sizeFromCode}[c]$. Furthermore, we can decode the value $v$ by reading a 64-bit integer from the starting position $p$ and applying a mask indexed by the size code $c$ to zero out the extra bytes: $v \leftarrow \text{ReadInt64}(p)$ & $\text{codeMask}[c]$.[2]

With a direct implementation, a significant amount of time is spent on matching strings in the prefix and constructing the completion string. In the compressed encoding of the Completion Trie, each trie node represents at most 7 characters. Thus, we can apply a similar masking technique to compare the first $\ell$ characters of two strings $p$ and $q$: *isMatch* $\leftarrow$ (ReadInt64($p$) & strMask[$\ell$]) = (ReadInt64($q$) & strMask[$\ell$]). When constructing the completion string, by over-allocating the string buffer that stores the completion string, we can copy 8 bytes from the node character sequence to the insertion position in one instruction and advance the insertion point by the desired length. By replacing several unpredictable conditional branching instructions with a few simple bit operations, these optimizations significantly improve the performance of the runtime algorithm.

## 5. RMQ TRIE

In this section, we describe a simple scheme to *augment* any sorted string dictionary data structure with an RMQ data structure, in order to support top-$k$ completion.

As shown in Figure 5, if the string set $\mathcal{S}$ is represented with a trie, the set $\mathcal{S}_p$ of strings prefixed by $p$ is a subtrie. Hence, if the scores are arranged in DFS order within an array $R$, the scores of $\mathcal{S}_p$ are those in an interval $R[a, b]$. This is true in general for any string dictionary data structure that maps the strings in $\mathcal{S}$ to $[0, |\mathcal{S}|)$ in lexicographic order. We call PrefixRange($p$) the operation that, given $p$, returns the pair $(a, b)$, or $\perp$ if no string matches the prefix.

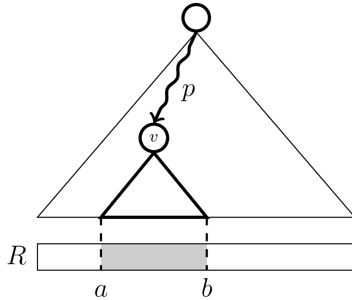



**Figure 5:** The scores of the strings prefixed by $p$ correspond to the interval $[a, b]$ in the scores vector $R$.

To enumerate the completions of $p$ in ranked order, we employ a standard recursive technique, used for example in [29, 19]. We build an RMQ data structure on top of $R$ using an inverted ordering, i.e. the *minimum* is the highest score. The index of the first completion is then $i = \text{RMQ}(a, b)$. Now the index of the second completion is the one with highest score among $\text{RMQ}(a, i-1)$ and $\text{RMQ}(i+1, b)$, which splits again either $[a, i-1]$ or $[i+1, b]$ into two subintervals. In general, the index of the next completion is the highest scored RMQ among all the intervals obtained with this recursive splitting. By maintaining the intervals in a priority queue ordered by score, it is hence possible to find the top-$k$ completion indices in $\tilde{O}(k \log k)$. We can then perform $k$ Access operations on the dictionary to retrieve the strings. The pseudo-code is shown in Algorithm 2.

**Algorithm 2** Top-$k$ completion with RMQ Trie.

```
   Input: Trie T, scores vector R, prefix p, and k ≥ 0
   Output: List c of top-k completions of p
 1 Q ← Empty priority queue
 2 c ← Empty list
 3 found, a, b ← PrefixRange(T, p)
 4 if found then
 5     i ← RMQ_R(a, b)
 6     Push(Q, (R[i], i, a, b))
 7     while Q is not empty do
 8         r, i, a, b ← Pop(Q)
 9         s ← Access_T(i)
10         Append (s, r) to result list c
11         if |c| = k then return c
12         if i > a then
13             j ← RMQ_R(a, i − 1)
14             Push(Q, (R[j], j, a, i − 1))
15         if i < b then
16             j ← RMQ_R(i + 1, b)
17             Push(Q, (R[j], j, i + 1, b))
18 return c
```

The space overhead of this data structure, beyond the space needed to store the trie and the scores, is just the space needed for the RMQ data structure, which is $2n + o(n)$ bits, where $n = |\mathcal{S}|$. If the trie can answer PrefixRange in time $T_P$ and Access in time $T_A$, the total time to retrieve the top-$k$ completions is $\tilde{O}(T_P + k(T_A + \log k))$.

The advantages of this scheme are its simplicity and modularity, since it is possible to re-use an existing dictionary data structure without any significant modification. In fact, in our experiments we use the lexicographic compressed trie of [17]. The only change we needed to make was to implement the operation PrefixRange. On the other hand, as we will see in Section 8, this comes at the cost of significantly worse performance than the two other data structures, which are specifically designed for the task of top-$k$ completion.

## 6. SCORE-DECOMPOSED TRIE

In this section, we introduce a compressed trie data structure specifically tailored to solve the top-$k$ completion problem. The structure is based on the succinct path-decomposed tries described in [17], but with a different path decomposition that takes into account the scores.

**Path decompositions.** Let $\mathcal{T}$ be the trie built on the strings of the scored string set $\mathcal{S}$. A *path decomposition* of $\mathcal{T}$ is a tree $\mathcal{T}^c$ whose nodes correspond to node-to-leaf paths in $\mathcal{T}$. The tree is built by first choosing a root-to-leaf path $\pi$ in $\mathcal{T}$ and associating it with the root node $u_\pi$ of $\mathcal{T}^c$; the children of $u_\pi$ are defined recursively as the path decompositions of

[2]This expression is specific to little-endian architectures. An additional shift operation is required for big-endian systems. Furthermore, we need to pad the end of the binary trie encoding with 7 buffer bytes to avoid reading past the end of the data. On processors without support for unaligned access, such as some ARM processors, ReadInt64 is less efficient.

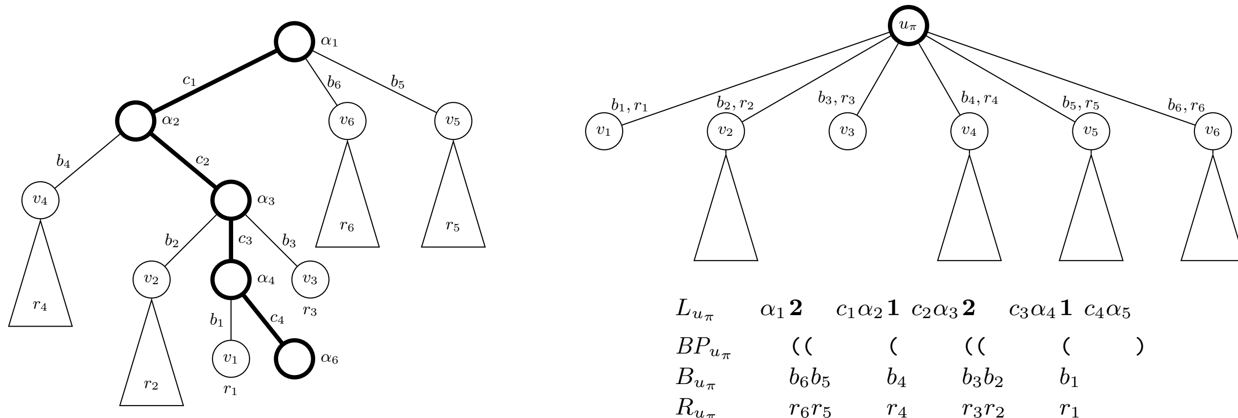



**Figure 6:** On the left, trie $\mathcal{T}$ with the decomposition path $\pi$ highlighted. On the right, root node $u_\pi$ in $\mathcal{T}^c$ and its encoding (spaces are for clarity only). In this example $v_6$ is arranged after $v_5$ because $r_5 > r_6$.

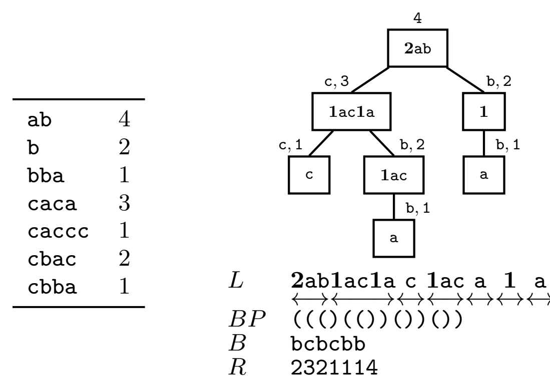
the subtries hanging off the path $\pi$, and their edges are labeled with the labels of the edges from the path $\pi$ to the subtries. See Figure 6 for an example.

Note that while each string $s$ in $\mathcal{S}$ corresponds to a root-to-leaf path in $\mathcal{T}$, in $\mathcal{T}^c$ it corresponds to a *root-to-node* path. Specifically, each leaf $\ell$ in $\mathcal{T}$ is chosen at some point in the construction as the decomposition path of a subtrie, which becomes a node $u$ in $\mathcal{T}^c$; the path from $u_\pi$ to $u$ in $\mathcal{T}^c$ corresponds to the root-to-leaf path of $\ell$ in $\mathcal{T}$. For the sake of simplicity we will say that $s$ corresponds to the node $u$.

***Max-score* path decomposition.** A path decomposition is completely defined by the strategy used to choose the path $\pi$ and order the subtries hanging off $\pi$ as children of the root $u_\pi$. Since each string corresponds to a leaf in $\mathcal{T}$, we can associate its score with the corresponding leaf. We define the *max-score path decomposition* as follows. We choose path $\pi$ as the one to the leaf with the highest score (ties are broken arbitrarily). The subtries are ordered bottom-to-top, while subtries at the same level are arranged in decreasing order of score (the score of a subtrie is defined as the highest score in the subtrie).

To enable scored queries, we need to augment the data structure to store the scores. Following the notation of Figure 6, let $u_\pi$ be the root node of $\mathcal{T}^c$ and $v_1, \ldots, v_d$ be the nodes hanging off the path $\pi$. We call $r_i$ the *highest score* in the subtrie rooted at $v_i$ (if $v_i$ is a leaf, $r_i$ is just its corresponding score). We add $r_i$ to the label of the edge leading to the corresponding child, such that the label becomes the pair $(b_i, r_i)$.

**Succinct tree representation.** To represent the Score-Decomposed Trie, we use the same encoding described in [17], which we briefly summarize here. For each node $u$ in $\mathcal{T}^c$ we build three sequences $L_u$, $BP_u$, and $B_u$. Figure 6 shows the encoding of the root node $u_\pi$; the other nodes are encoded recursively. $L_{u_\pi}$ contains the concatenation of the node and edge labels along the path $\pi$, interleaved with special characters $\mathbf{1}, \mathbf{2}, \ldots$ that indicate how many subtries branch off that point in the path $\pi$. We call the positions of these special characters *branching points*. $BP_{u_\pi}$ contains one open parenthesis for each child of $u_\pi$, followed by a single close parenthesis. $B_{u_\pi}$ contains the sequence of the characters $b_i$ branching off the path $\pi$ in reverse order. The sequences for each node are concatenated in DFS order into the three



| | |
|---|---|
| ab | 4 |
| b | 2 |
| bba | 1 |
| caca | 3 |
| caccc | 1 |
| cbac | 2 |
| cbba | 1 |



**Figure 7:** Score-Decomposed Trie example and its encoding.

sequences $L$, $BP$, and $B$. In particular, after prepending an open parenthesis, $BP$ is the DFUDS representation of the topology of the path-decomposed tree. Note that the branching characters $b_i$ are in one-to-one correspondence with the open parentheses in $BP$, which in turn correspond to the nodes of $\mathcal{T}^c$. In addition, we need to store the scores in the edges along with the branching characters. We follow the same strategy used for the branching characters: concatenate the $r_i$'s in reverse order into a sequence $R_{u_\pi}$, and then concatenate the sequences $R_u$ for each node $u$ into a sequence $R$ in DFS order. Finally, append the root score to $R$.

The advantage of storing $BP$, $B$, $L$, and $R$ separately is that they can be compressed independently with specialized data structures, provided that they support the operations needed by the traversal algorithms. Specifically, $BP$ and $B$ are stored explicitly as a balanced parentheses structure and character array, respectively. We compress the sequence of labels $L$ using a variant of RePair [22] that supports scanning each label in constant-time per character [17]. The sequence $R$ is compressed using the data structure described in Section 7.

**Top-$k$ completions enumeration.** The operations Lookup and Access [17] do not need any modification, as they do not depend on the particular path decomposition strategy used. We now describe how to support top-$k$ completion queries.

Because of the *max-score* decomposition strategy, the highest score in each subtrie is exactly the score of the decompo-

sition path for that subtrie. Hence if $r_i$ is the highest score of the subtrie rooted in $v_i$, and $u_i$ is the node in $\mathcal{T}^c$ corresponding to that subtrie, then $r_i$ is the score of the string corresponding to $u_i$. This implies that for each $(s, r)$ in $\mathcal{S}$, if $u$ is the node corresponding to $s$, then $r$ is stored in the incoming edge of $u$, except when $u$ is the root $u_\pi$, whose score is stored separately. Another immediate consequence of the decomposition is that the tree has the *heap property*: the score of each node is less or equal to the score of its parent.

We exploit this property to retrieve the top-$k$ completions. First, we follow the algorithm of the Lookup operation until the prefix $p$ is exhausted, leading to the *locus node* $u$, the highest node whose corresponding string contains $p$. This takes time $\widetilde{O}(|\Sigma||p|)$. By construction, this is also the highest scored completion of $p$, so we can immediately report it. To find the next completions, we note that the prefix $p$ ends at some position $i$ in the label $L_u$. Thus, all the other completions must be in the subtrees whose roots are the children of $u$ branching *after* position $i$. We call the set of such children the *seed set*, and add them into a priority queue.

To enumerate the completions in sorted order, we extract the highest scored node from the priority queue, report the string corresponding to it, and add all its children to the priority queue. For the algorithm to be correct, we need to prove that, at each point in the enumeration, the node corresponding to the next completion is in the priority queue. This follows from the fact that every node $u$ corresponding to a completion must be reached at some point, because it is a descendant of the seed set. Suppose that $u$ is reported after a lower-scored node $u'$. This means that $u$ was not in the priority queue when $u'$ was reported, implying that $u$ is a descendant of $u'$. But this would violate the heap property.

The previous algorithm still has a dependency on the number of children in each node, since all of them must be placed in the priority queue. With a slight modification in the algorithm, this dependency can be avoided. Note that in the construction, we sort the children branching off the same branching point in decreasing score order. Thus, we can delay the insertion of a node into the priority queue until after all other higher-scored nodes from the same branching point have already been expanded. For each node $u$, the number of branching points in $L_u$ is at most $|L_u|$. Hence, we add at most $|L_u| + 1$ nodes to the priority queue: 1 for each branching point and the next sibling, if any, of node $u$. Thus, the time to return $k$ completions is $\widetilde{O}(lk \log lk)$ where $l$ is the average length of the completions returned minus the prefix length $|p|$.

**Comparison with Completion Trie.** The algorithm described above is very similar to Algorithm 1 for the Completion Trie. In fact, the Score-Decomposed Trie can be seen as a path decomposition of the Completion Trie, and the previous algorithm as a simulation of Algorithm 1 on the transformed tree. However there are two significant differences. First, the scores in the Completion Trie along the max-score path are, by construction, all the same. Thus, they can be written just once. Hence, while the Completion Trie stores at least $2n - 1$ scores for $n$ strings, the Score-Decomposed Trie only stores $n$. Second, after the locus node is found, only $k - 1$ nodes need to be visited in order to return $k$ completions. In contrast, Completion Trie may require visiting up to $\Omega(kl)$ nodes. This property makes the Score-Decomposed Trie very suitable for succinct representations, whose traversal operations are significantly slower than pointer-based data structures.

# 7. SCORE COMPRESSION

For both data structures described in Section 5 and Section 6, it is necessary to store the array $R$ of scores, and perform random access quickly. Further, it is crucial to effectively compress the scores: if stored directly as 64 bit integers, they would take more than half of the overall space.

As noted in Section 1, many scoring functions (number of clicks/impressions, occurrence probability, ...) exhibit a power law distribution. Under this assumption, encoding the scores with $\gamma$-codes [11] (or in general $\zeta$-codes [5]) would give nearly optimal compression. However it would not be possible to support efficient random access to such arrays. Specifically, we experimented with a random-access version of $\gamma$-codes: concatenate the binary representations of the values of $R$ (without the leading 1) into a bitvector and use a second bitvector to delimit their endpoints, which can be retrieved using $\text{Select}_1$. While this obtained very good compression, it came at the cost of a significant slowdown in retrieval.

We use instead a data structure inspired by *Frame of Reference* compression [15], which we call *packed-blocks array*. The scores array of length $n$ is divided into blocks of length $l$; within each block $j$ the scores are encoded with $b_j$ bits each, where $b_j$ is the minimum number of bits sufficient to encode each value in the block. The block encodings are then concatenated in a bitvector $B$. To retrieve the endpoints of the blocks inside $B$ we employ a two-level directory structure: the blocks are grouped into *super-blocks* of size $L$, and the endpoint of each block is stored relative to the beginning of the superblock using $O(\log(Lw))$ bits, where $w$ is the size in bits of the largest representable value. The endpoint of each superblock is encoded using $O(\log(nw))$ bits. To retrieve a value, the endpoints of its block are retrieved using the directory structure; then $b_j$ is found by dividing the size of the block by $l$. The overall time complexity is constant. In our implementation, we use $l = 16$, $L = 512$, 16-bit integers for the block endpoints, and 64-bit integers for the super-block endpoints.

In our experiments, the slowdown caused by the packed-blocks array instead of a plain 64-bits array was basically negligible. On the other hand, as we show in Section 8 in more detail, we obtain very good compression on the scores, down to a few bits per integer. We attribute the good compression to the fact that each group of sibling scores are arranged in DFS order. As the decomposed trie exhibits the heap property, the score of each node upper bounds the scores of its descendants. This increases the likelihood that adjacent sibling groups have scores with the same order of magnitude. Hence, the waste induced by using the same number of bits for $l$ consecutive values is relatively small.

# 8. EXPERIMENTAL ANALYSIS

To evaluate the effectiveness of the proposed top-$k$ completion techniques, Completion Trie (CT), Score-Decomposed Trie (SDT), and baseline RMQ Trie (RT), we will compare their effectiveness on the following datasets from different application scenarios on an Intel i7-2640M 2.8GHz processor with 128/512/4096KB of L1/2/3 cache and 8GB of RAM, compiled with Visual C++ 2012 running on Windows 8.

- QueriesA: 10,154,742 unfiltered search queries and their associated counts from the AOL query log [1]. This dataset is representative of the style and frequency of

queries users may enter into the search box of a search engine or large website.

- QueriesB: More than 400M filtered search queries and their click counts from a commercial search engine for scalability evaluation.
- URLs: 18M URL domains and click counts derived from the query click log of a commercial search engine, representing the scenario of URL completion suggestions in web browser address bars. As users generally skip the initial URL protocol (eg. `http`) and often the `www` domain prefix, for each URL, we artificially inject additional entries with the same count to accommodate such behavior, for a total of 42M unique scored strings. Unlike queries, URLs share a small set of extension suffixes, which makes the data more compressible.
- Unigrams: The top 1M words and their probabilities from the Microsoft Web N-gram Service (bing-body:apr10) [27]. We quantize the probabilities to $\lfloor 1000 \ln(p) \rfloor$. This dataset is representative of the lexicons used by mobile devices with soft keyboards, which need a large lexicon for each language to support predictive text entry and spelling correction, but the tight memory resources require a space-efficient storage.

In each dataset we subtracted from the scores their minimum, so that the smallest score is 0, without affecting the ordering. The minimum is then added back at query time.

## 8.1 Space

We evaluate the compactness of the data structures by reporting in Table 1 the average number of bits per string (including score). For comparison, we also report the size of the original uncompressed text file (Raw) and the `gzip` compressed binary (GZip). Across the 4 datasets, the three presented techniques achieve an average compression ratio of between 29% and 51%, with SDT consistently having the smallest size. In fact, its size is only 3% larger than that achieved by `gzip` compression on average, and is actually 10% smaller on the Unigrams datatset.

| Dataset | Raw | GZip | CT | SDT | RT |
|---|---|---|---|---|---|
| QueriesA | 209.8 | 56.3 | 120.5 | 62.4 | 65.5 |
| QueriesB | 235.6 | 57.9 | 112.0 | 61.2 | 64.4 |
| URLs | 228.7 | 54.7 | 130.9 | 58.6 | 62.0 |
| Unigrams | 114.3 | 44.2 | 49.3 | 39.8 | 42.1 |

**Table 1:** Data structure sizes in bits per string.

To better understand how the space is used, we present in Figure 8 the storage breakdown of each of the techniques on QueriesA. For CT, 70% of the space is used to store the uncompressed character sequences. Compressing the node character sequences with RePair [22] can further reduce the size, but will incur some sacrifice in speed. With delta encoding, storing the scores, including the 2 bit header, takes only 4.0 and 9.6 bits per node and string, respectively. In comparison, standard variable-byte encoding with a single continuation bit [38] requires at least 8 bits per node. Similarly, we utilize an average of only 16.4 bits per string in the dataset to encode the tree structure. As reference, it would have required 24 bits just to encode the index of a string.

For SDT, nearly 90% of the space is dedicated to storing the compressed labels and branching characters. On average, each score takes 4.1 bits, less than half of CT; while maintaining the tree structure via *BP* requires only 2.7 bits per string. RT behaves similarly except each score takes 4.9 bits as the child nodes are sorted lexicographically rather than by score. In addition, it requires a Cartesian tree to perform Range Minimum Queries, which takes a further 2.7 bits per string.

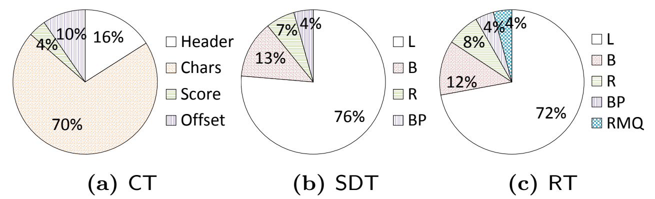



**Figure 8:** Data structure size breakdowns.

## 8.2 Time

To evaluate the runtime performance of the proposed data structures, we synthesize a sequence of completion requests to simulate an actual server workload. Specifically, we first sample 1M queries in random order from the dataset according to the normalized scores. Assuming that user queries arrive according to a Poisson process, we can model the inter-arrival time of each query using an exponential distribution. We can control the average queries per second (QPS) by adjusting the $\lambda$ parameter of the exponential distribution. For simplicity, we assume that each subsequent keystroke arrives 0.3 seconds apart, corresponding to an average typing speed of 40 word per minutes. Users will continue to enter additional keys until the target query appears as the top suggestion, or until the query has been fully entered. Note that with higher QPS, requests from different queries are more likely to overlap, leading to more cache misses.

In Table 2, we present the mean time to compute the top-10 completions, averaged over 10 runs. Overall, CT achieves the best performance, about twice as fast as SDT. While much of the differences can be attributed to SDT's use of succinct operations for trie traversal and RePair decoding of label sequence $L$, CT's better memory locality, where all node information are stored together, still plays an important part. For instance, we see that when the nodes are not arranged for locality, as is the case for RT, the performance is extremely poor. Similarly, as the requests corresponding to higher QPS exhibit less overlap in memory access, the performance degrades by an average of 10% for CT and 21% for SDT. As the prefixes used by the two workloads differ only in order, the performance gap is due entirely to the effect of CPU cache, where CT shines. To simulate a moderate workload, we use 1K QPS in the remaining analyses.

| | 1 QPS | | | 1K QPS | | |
|---|---|---|---|---|---|---|
| Dataset | CT | SDT | RT | CT | SDT | RT |
| QueriesA | 3.30 | 6.65 | 30.41 | 3.65 | 8.04 | 33.92 |
| QueriesB | 4.45 | 9.85 | 49.09 | 5.34 | 13.71 | 58.43 |
| URLs | 4.81 | 9.47 | 50.23 | 4.99 | 10.25 | 52.94 |
| Unigrams | 2.06 | 3.89 | 17.13 | 2.12 | 4.08 | 17.82 |

**Table 2:** Average time per top-10 completion query in $\mu$s.

To better understand the performance differences between the techniques, we break down the total time to compute the top-10 completions on QueriesA into the time spent finding the locus node and computing each successive completion. As shown in Figure 9, CT using pointer arithmetic is significantly faster than data structures using balanced parentheses for traversal, especially in finding the initial locus node. Theoretically, the cost of retrieving each additional completion increases logarithmically. But in practice, the incremental cost for both CT and SDT remains mostly constant (not shown), as it is dominated by memory access time, with decreasing probability of cache miss for each additional completion. In fact, for RT, it actually takes less time to compute each additional completion. Furthermore, although we are also returning the completion string, each completion in SDT is about twice as fast as a random Access operation. CT has an even larger ratio due to its less efficient Access operation. Thus, by integrating string construction into the completion algorithm, we sigificantly reduce the overall time required to enumerate the top-$k$ completions.

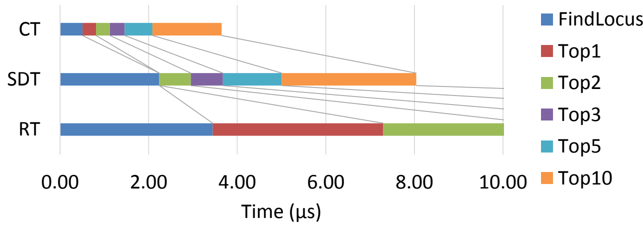



**Figure 9:** Completion time breakdowns.

In terms of build time, CT, SDT, and RT with unoptimized code currently take an average of 1.8, 7.8, and 7.7 $\mu s$ per string in QueriesA, respectively, with RePair compression taking up 73% of the time for the two succinct tries. All algorithms use memory linear to the size of the binary output.

## 8.3 Scalability

To assess the scalability of the data structures, we compare their performance on different size subsets of the QueriesB dataset. Specifically, to mimic practical scenarios where we have a limited memory budget and can only afford to serve the most popular queries, we will generate these subsets by taking the top-$N$ distinct queries in decreasing score order. Figure 10 plots the change in average bytes per query as we increase the number of queries. Overall, we see that lower count tail queries are longer and require more space across all techniques, likely due to the different characteristics exhibited by queries with only a few counts. While SDT requires more space than CT below 100 queries due to its large sublinear overhead, its size continues to fall with increasing number of queries and actually becomes *smaller* than GZip on a wide range of dataset sizes.

We present in Figure 11 the effect the number of queries has on the average time per completion for top-10 completion requests. We use the synthesized workload based on the full QueriesB dataset to best approximate real world usage scenarios where users enter prefixes without knowing what queries the system can complete. Thus, both the average number of completions and average completion length increase with the dataset size. As shown, the average time per completion for CT increases very slowly, due to increasing completion length and more cache misses. It is higher for smaller datasets as

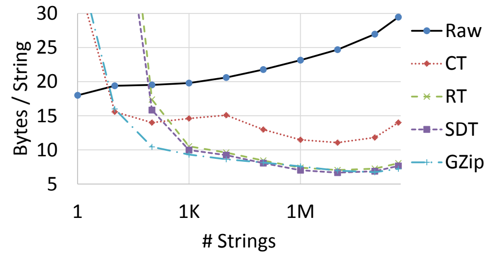



**Figure 10:** Data structure size vs. dataset size.

the we have fewer completions to distribute the cost of FindLocus over. As SDT accesses more lines of CPU cache per completion, it performs worse than CT, with increasing time ratio. RT further suffers from lack of memory locality among the top completions which magnifies the effect of cache miss.

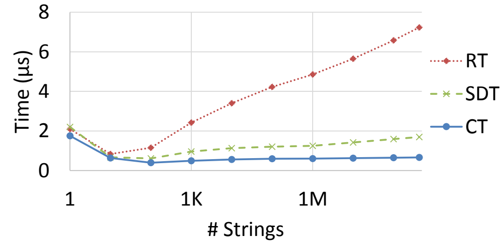



**Figure 11:** Average time per completion vs. dataset size.

## 8.4 Discussions

In practical scenarios, auto-completion needs to support not only exact prefix matches, but also inexact matches due to differences in casing, accents, or spelling. One way to support case and accent insensitive match is to normalize both the dataset strings and the input prefix into lowercase unaccented characters before computing the completions. However, this removes the original casing and accents from the completions, which may be important for certain languages and scenarios.

An alternative technique is to apply a fuzzy completion algorithm, such as the one described by Duan and Hsu [10]. In short, after adding the root node to a priority queue, iteratively add the children of the best path to the queue, applying a penalty as determined by a weighted transformation function if the character sequence of the child node does not match the input prefix. Once a path reaches a leaf node in the trie and has explained all characters in the input prefix, return the completion. This fuzzy completion algorithm only requires basic trie traversal operations and access to the best descendant score of each node, which are supported by all of the proposed trie data structures. As this algorithm essentially merges the top completions of various spell corrected prefixes, the ability to retrieve additional completions efficiently and on-demand is critical to meeting target performances on web-scale server loads.

Another common scenario is the need to associate additional data with each string entry. For example, to map the injected partial URLs from the URLs dataset to their canonical forms, as shown in Figure 1b, we can create an array that maps the index of each string in the dataset to the index of full URL, or a special value if no alteration mapping is required. These auxiliary arrays are often sparse and can be compressed efficiently using various succinct and compressed

data structures [31]. Although CT only maps each completion to a node offset, we can create a small bitvector with Rank and Select capabilities to convert between the offsets and indices.

Furthermore, some applications need to retrieve the top-$k$ completions according to a *dynamic* score that depends on the prefix and completion. As the static score is usually a prominent component of the dynamic score, an approximate solution can be obtained by taking the top-$k'$ completions with $k' > k$ according to the static score and re-ranking the completion list.

To truly scale to large datasets, we need to build the proposed trie structures efficiently. Although we have not discussed the build process in detail due to the lack of space, we have implemented efficient algorithms that scale linearly with the size of the dataset. For CT, we have further developed efficient techniques to merge tries with additive scores, enabling distributed trie building across machines.

# 9. CONCLUSION

In this paper, we have presented three data structures to address the problem of top-$k$ completion, each with different space/time/complexity trade-offs. Experiments on large-scale datasets showed that Completion Trie, based on classical data structures, requires roughly double the size of Score-Decomposed Trie, based on succinct primitives. However, it is about twice as fast. As it turns out, organizing the data in a locality-sensitive ordering is crucial to the performance gains of these two structures over the simpler RMQ Trie.

For scenarios where memory is scarce, Score-Decomposed Trie can achieve sizes that are competitive with `gzip`. When throughput dominates the cost, Completion Trie can reduce the time for each completion to under a microsecond. For most applications, the difference of a few microseconds between Completion Trie and Score-Decomposed Trie should be negligible. However, for algorithms that require numerous trie traversals, such as fuzzy completion where we consider a large number of locus nodes, the speedup from Completion Trie may become significant.

As handling big data becomes ever more important, succinct data structures have the potential to significantly reduce the storage requirement of such data while enabling efficient operations over it. Although their theoretical performance matches their classical counterparts, there is still a noticeable gap in practice. It is an interesting open question whether such gap can be closed, thus obtaining the best of both worlds.

# 10. ACKNOWLEDGEMENTS

The second author would like to thank Roberto Grossi for discussions and suggestions on an early draft of the paper, and to acknowledge the partial support of Midas EU Project (318786), InGeoCloudS EU Project (297300), MaRea project (POR-FSE-2012), and FIRB Linguistica 2006 (RBNE07C4R9_003).

# 11. REFERENCES

[1] AOL search data. `http://www.gregsadetsky.com/aol-data/`, 2006.

[2] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *ALENEX*, pages 84–97, 2010.

[3] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[4] I. Bialynicka-Birula and R. Grossi. Rank-sensitive data structures. In *SPIRE*, pages 79–90, 2005.

[5] P. Boldi and S. Vigna. Codes for the World Wide Web. *Internet Mathematics*, 2(4):407–429, 2005.

[6] N. R. Brisaboa, R. Cánovas, F. Claude, M. A. Martínez-Prieto, and G. Navarro. Compressed string dictionaries. In *SEA*, pages 136–147, 2011.

[7] K. W. Church, B. Thiesson, and R. Ragno. K-best suffix arrays. In *HLT-NAACL (Short Papers)*, pages 17–20, 2007.

[8] D. R. Clark. *Compact pat trees*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1998. UMI Order No. GAXNQ-21335.

[9] P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In *COCOON*, pages 396–407, 2012.

[10] H. Duan and B.-J. P. Hsu. Online spelling correction for query completion. In *WWW*, pages 117–126, 2011.

[11] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, Mar. 1975.

[12] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[13] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

[14] E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, September 1960.

[15] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, pages 370–379, 1998.

[16] J. Grieves. The secrets of the Windows Phone 8 keyboard. `http://blogs.windows.com/windows_phone/b/windowsphone/archive/2012/12/06/the-secrets-of-the-windows-phone-8-keyboard.aspx`, December 2012.

[17] R. Grossi and G. Ottaviano. Fast compressed tries through path decompositions. In *ALENEX*, pages 65–74, 2012.

[18] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[19] W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *FOCS*, pages 713–722, 2009.

[20] G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554, 1989.

[21] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, USA, 2nd edition, 1998.

[22] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference*, pages 296–305, 1999.

[23] F. Li. Simpler search. `http://blog.twitter.com/2012/07/simpler-search.html`, July 2012.

[24] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a TASTIER approach. In *SIGMOD*, pages 695–706, 2009.
[25] G. Li, J. Wang, C. Li, and J. Feng. Supporting efficient top-k queries in type-ahead search. In *SIGIR*, pages 355–364, 2012.
[26] D. Matani. An $O(k \log n)$ algorithm for prefix based ranked autocomplete. `http://www.dhruvbird.com/autocomplete.pdf`, 2011. Preprint.
[27] Microsoft Web N-gram Service. `http://web-ngram.research.microsoft.com/`.
[28] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, June 2001.
[29] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.
[30] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, pages 219–230, 2007.
[31] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*, 2007.
[32] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, N.J., 3 edition, 2003.
[33] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *SODA*, pages 134–149, 2010.
[34] *Succinct* library. `http://github.com/ot/succinct`.
[35] D. Sullivan. How Google Instant's autocomplete suggestions work. `http://searchengineland.com/how-google-instant-autocomplete-suggestions-work-62592`, April 2011.
[36] R. Vernica and C. Li. Efficient top-k algorithms for fuzzy search in string collections. In *KEYS*, 2009.
[37] S. Vigna. Broadword implementation of rank/select queries. In *WEA*, pages 154–168, 2008.
[38] H. E. Williams and J. Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999.

# APPENDIX

## A. DATA STRUCTURE FOR RMQ

To perform Range Minimum Queries on a vector of scores, we use the 2d-Min-Heap described by Fischer and Heun [13], which, as noted by Davoodi et al. [9], is an alternative representation of the Cartesian tree. As in [13], we build the DFUDS representation of the 2d-Min-Heap in the bitvector $U$. The RMQ of $i$ and $j$ can then be reduced to the RMQ on the excess sequence $E$ of $U$, denoted as $\pm 1\text{RMQ}_E$, with the following algorithm. Note that the indices are slightly different from [13] because all our primitives are 0-based.

Lemma A.1 ([13, Section 5.1]). *If $U$ is the DFUDS representation of a 2d-Min-Heap, then* $\text{RMQ}(i, j)$ *can be computed by the following algorithm.*

1 $x \leftarrow \text{Select}_{)}(U, i+1)$
2 $y \leftarrow \text{Select}_{)}(U, j+1)$
3 $w \leftarrow \pm 1\text{RMQ}_E(x, y)$
4 **if** $\text{Rank}_{)}(U, \text{FindOpen}(U, w-1)) = i+1$ **then**
5     **return** $i$
6 **else**
7     **return** $\text{Rank}_{)}(U, w-1)$

Our implementation differs from the one in [13] in two ways. First, instead of using an ad-hoc data structure for $\pm 1\text{RMQ}_E$, we re-use the same Range-Min tree [17] that is used for tree navigation. The Range-Min tree divides the excess sequence $E$ into blocks and builds a binary tree whose nodes correspond to intervals in $E$: the leaves correspond to individual blocks, while each internal node corresponds to the union of the intervals corresponding to its children. In each node, we store the excess minimum for the corresponding interval. It is straightforward to find the block with minimal excess between the block containing $x$ and the one containing $y$ in $O(\log n)$ time. Then, to find the position of minimal excess within a block, we perform a simple linear scan by dividing the block into 8-bit chunks and using a lookup table to find the minimum inside each chunk.

Second, line 4 in Lemma A.1 checks whether the node at position $x$ is the parent of $w-1$ (in which case the minimal value is at position $i$). We replace it with the following line.

4 **if** $\text{Excess}(U, \text{Select}_{)}(i)+1) \leq \text{Excess}(U, w)$ **then**

Since, in our implementation, Select is significantly faster than FindOpen, the whole algorithm speeds up by $10-20\%$ with this change. The following lemma proves that the two conditions are equivalent.

Lemma A.2. *In the algorithm described in Lemma A.1,* $\text{Excess}(U, \text{Select}_{)}(i)+1) \leq \text{Excess}(U, w)$ *if and only if the node at $w-1$ in $U$ is a child of the node at* $\text{Select}_{)}(i+1)$.

Proof. Let $t = \text{Select}_{)}(i)$ and $x = \text{Select}_{)}(i+1)$. If the node at $w-1$ is a child of the node at $x$, then its mate $z$ is between $t$ and $x$ as shown below:

$$\underset{t}{)}(\cdots\underset{z}{(}\cdots\underset{x}{()}\cdots\cdots\cdots\underset{w-1}{)}$$

If follows that $\text{Excess}(t+1) \leq \text{Excess}(z)$. Since $\text{Excess}(z) = \text{Excess}(w)$, we obtain $\text{Excess}(t+1) \leq \text{Excess}(w)$.

Conversely, suppose $\text{Excess}(U, t+1) \leq \text{Excess}(U, w)$. Since $w = \pm 1\text{RMQ}_E(x, y)$, it holds $\text{Excess}(U, w) < \text{Excess}(U, x)$. Hence there must be a position $z$ such that $t+1 \leq z < x$ and $\text{Excess}(U, z) = \text{Excess}(U, w)$. To prove that $z$ is the mate of $w-1$, it suffices to note that the excess between $z+1$ and $w-1$ is strictly greater than $\text{Excess}(w)$, again because $w$ is the leftmost excess minimum in the range $[x, y]$. □