Information Retrieval Journal (2022) 25:481–518
https://doi.org/10.1007/s10791-022-09416-9

Applying burst-tries for error-tolerant prefix search

Berg Ferreira[1] · Edleno Silva de Moura[1,2] · Altigran da Silva[1]

Received: 4 June 2021 / Accepted: 29 September 2022 / Published online: 18 October 2022

Abstract
In this work we address the problem of performing an error-tolerant prefix search on a set of string keys. While the ideas presented here could be adopted in other applications, our primary target application is error-tolerant query autocompletion. Tries and their variations have been adopted as the basic data structure to implement recently proposed error-tolerant prefix search methods. However, they must require a lot of extra memory to process queries. Burst tries are alternative compact tries proposed to reduce storage costs and maintain a close performance when compared to the use of tries. Here we discuss alternatives for adapting burst tries as error-tolerant prefix search data structures. We show how to adapt state-of-the-art trie-based methods for use with burst tries. We studied the trade-off between memory usage and time performance while varying the parameters to build the burst trie index. As an example, when indexing the JusBrasil dataset, one of the datasets adopted in the experiments, the use of burst tries reduces the memory required by a full trie to 26% and increases the time performance to 116%. The possibility of balancing memory usage and time performance constitutes an advantage of the burst trie when compared to the full trie when adopted as an index for the task of performing error-tolerant prefix search.

Keywords Query processing · Error-tolerant · Autocompletion · Trie

1 Introduction

In this work we address the problem of performing an error-tolerant prefix search on a given set of string keys. Let Σ be an alphabet. A string q is a sequence of Σ symbols. We use $|q|$ to denote the length of q, $q[i]$ to denote the i-th symbol of q, starting at 1, and $q[i..j]$ to denote a sub-string of q starting at position i and ending at position j. Let p and

✉ Berg Ferreira
berg@icomp.ufam.edu.br

Edleno Silva de Moura
edleno@icomp.ufam.edu.br

Altigran da Silva
alti@icomp.ufam.edu.br

1 Institute of Computing, Federal University of Amazonas, AM, Manaus, Brazil

2 Search Division, Jusbrasil, Manaus, AM, Brazil

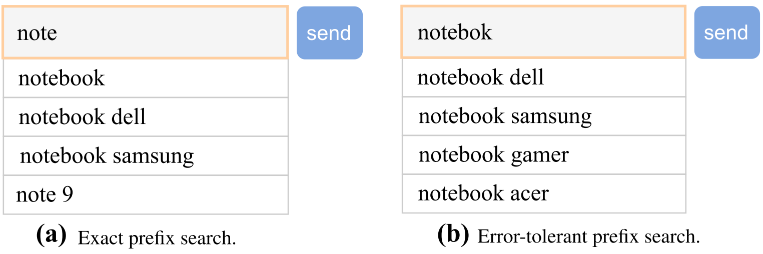


Fig. 1 Examples of query autocompletion using **a** exact prefix search and **b** error-tolerant prefix search

q be two distinct strings composed of the symbols in Σ. We say that p is a prefix of q if p=$q[1..|p|]$. When p is a prefix of q, we say that there is an exact prefix match between p and q.

When searching allowing errors, we need to define a metric to measure the *distance* between two compared strings p and q. Here we adopt the well-known edit distance, where the *number of errors*, or *distance*, is given by the minimum number of insertions, removals or substitutions of symbols needed to transform p into q. The number of errors to accept a match becomes a parameter. When the edit distance between p and q equals τ, we say that p matches q with τ errors. If p matches with τ errors to any prefix of a string q, we say that there is an *error-tolerant prefix match* with τ errors between p and q.

Given the above concepts, we can now explain the problem addressed here. Let p be a string and let Q=$\{q_1, \dots, q_n\}$ be a set of strings to search for. The *error-tolerant prefix search problem* addressed here is to find all strings $q_i \in Q$ such that there is an *error-tolerant prefix match* between p and q_i with a maximum number of errors τ.

There are a variety of practical applications where an error-tolerant prefix search can be useful, such as query autocompletion and spelling correction. Our study is motivated by the application of error-tolerant query autocompletion, which is an essential and ubiquitous feature in the interaction between the user and the input interface of modern search engines (Smith et al., 2017; Wang & Lin, 2020; Tahery & Farzi, 2020; Krishnan et al., 2020; Kang et al., 2021). It can be seen as a specialized instance of the more general query suggestion problem, where suggestions need to be selected instantly based on the prefix queries generated as the user types (Chen et al., 2020). Figure 1a shows an example where the user has typed the prefix query "note", and the system suggests possible queries that match it.

Error-tolerant query autocompletion can also be seen as a mechanism to help users spell difficult queries or correct typos when the user is writing a prefix query. An example of a search system that allows error-tolerant query autocompletion is shown in Fig. 1b, where the user receives suggestions "notebook dell", "notebook samsung", "notebook gamer", "notebook acer" and "notebook lenovo", all of which matching the misspelled typed prefix query "notebok".

When searching in a system that allows query autocompletion, users may submit prefix queries containing typos that can result in unsatisfactory or even empty query suggestion results in an exact search system. Because of this, recent works have proposed error-tolerant prefix search algorithms for such applications (Chaudhuri & Kaushik, 2009; Ji et al., 2009; Li et al., 2011; Xiao et al., 2013; Deng et al., 2016; Zhou et al., 2016; Qin et al., 2019). In this particular application, the searched string can be large. For instance, one of the datasets adopted in our experiments contains more than 23 million strings.

Solutions that adopt indexes based on tries (Fredkin, 1960) are among the most successful ones. Despite their popularity and effectiveness, tries may require more memory space than the searched string set itself. To reduce the storage costs while maintaining good performance, Heinz et al. (2002) proposed a data structure called burst trie. Here we propose and study the use of burst tries to implement error-tolerant prefix search. We show that such an approach results in a competitive alternative to perform error-tolerant prefix search on large sets of strings, since it yields a reduction in the memory usage for query processing when compared to using full tries, while achieving a similar query processing time performance. Furthermore, the approach can be easily adopted for a large set of trie-based error-tolerant prefix search methods.

We study three different heuristics to burst containers when creating burst tries. The first heuristic, which we call *Minimum Container Depth* (MCD), limits the minimum depth of containers in the burst trie, while the second heuristic limits the maximum number of elements in each container. The second heuristic was proposed by Heinz et al. (2002) and is referred to here as *Maximum Container Keys* (MCK). We also study as a third heuristic the combination of MCD and MCK and present experiments showing that the studied alternatives produce a considerable reduction in memory usage for processing error-tolerant prefix search, while keeping the time performance close to that achieved by the full trie.

As a complementary study, we also investigate alternative ways to build tries used to perform error-tolerant prefix search, proposing a simple but effective way to organize trie nodes in memory when building the index. Existing algorithms usually work by inserting nodes into the tree one key at a time, a strategy we call *DFS* building. Here we experiment with another index building strategy, we call *BFS* building, where nodes are inserted level by level, rather than key by key. It requires the trie keys to be known in advance, since it requires the insertion of nodes to occur one level at a time. Also, nodes need to be sorted by the keys before insertion. We argue and experimentally show that this strategy, when applicable, can considerably reduce query processing times. Such gain is achieved because the BFS index building strategy favors breadth-first search (BFS) in the trie nodes, which is adopted by many of the previously proposed trie-based error-tolerant prefix search algorithms (Chaudhuri & Kaushik, 2009; Ji et al., 2009; Li et al., 2011; Deng et al., 2016; Zhou et al., 2016). Importantly, this performance improvement is achieved without requiring any changes to the query processing algorithm. In our experiments, using BFS when processing prefix queries with $\tau = 3$ was more than twice as fast as using the DFS building strategy.

Our contributions can be summarized as follows:

- We discuss and evaluate the application of burst tries in error-tolerant prefix search tasks.
- We investigate the impact of building the trie using a BFS index building strategy as an alternative to the more intuitive DFS strategy for trie nodes allocation. Although it requires the keys to be sorted, we show that BFS, when applicable, can reduce query processing times.
- We present experiments to verify the impact of using our ideas in practical scenarios applied to two distinct datasets adopted in previous studies and to a real-case dataset extracted from an online search service.

The rest of this paper is organized as follows. Sect. 2 presents the problem we have addressed, reviews the related work, and presents some definitions necessary to understand our proposed strategies and methods. It focuses on trie-based error-tolerant prefix search algorithms in the literature, including a brief description of the state-of-the-art BEVA algorithm. Sect. 3 presents basic concepts about tries and search operations on them. Sect. 4 presents our discussion about how to use burst tries as indexes to perform error-tolerant prefix search. Sect. 5 presents a discussion about practical implementation issues, especially a discussion about BFS and DFS trie building strategies. Sect. 6 presents experiments performed with the alternative burst tries implementation studied here and a comparison of their performance with representative baselines data structures. Finally, in Sect. 7, we present our conclusions and possible directions for future research.

2 Background and related work

Krishnan et al. (2017) define a set of query autocomplete modes based on how the characters already typed by the user are matched with the dataset of query suggestions. Each mode may result in different completions being presented. The taxonomy includes mode 1, where a prefix match is performed between the characters already typed by the user and each complete query suggestion in the dataset. Mode 2, where a prefix match is performed word by word. As a result, the system has a set of matches for each word already typed by the user, and the final suggestions are selected based on these sets. Mode 2 may, for instance, show only results that match all words already typed or use a ranking function to select the best results. Mode 3 also parses the query into words, but it allows the matching between the words at any position, so does not restrict the results to prefix matches.

Matching allowing errors could be applied to extend all the 3 initial modes (Krishnan et al., 2017). The choice of a mode is a design decision, since each mode may bring positive aspects and also negative aspects to the solution. As an example, Krishnan et al. (2017) describe that mode 3 allows finding a match between the query "gam rone" and the suggestion "game of thrones". At a first glance it seems to be nice, but it might not be a good match and the decision depends on the user's interest. For instance, the Google[1] search engine gives "gamerone" or "gam ronex" as suggestions for the string "gam rone", and these might be better than "game of thrones". The discussion above shows that all modes might be useful and interesting. This evidences that query autocompletion systems might be, for instance, implemented as a combination of distinct modes.

The discussion about how to implement error-tolerant prefix search using compact trie representations presented here is useful for modes that perform prefix search, especially modes 1 and 2 when allowing errors. We stress that the error-tolerant prefix search is just a small part of the query autocompletion systems. This is especially true when processing the search using mode 1, where the prefix search is performed over a smaller set of strings, the vocabulary containing the distinct words found in the dataset of suggestions, and where each word of the vocabulary is associated with an inverted list. In mode 2 the processing of the inverted lists not only may take more space than the vocabulary, but is also more expensive, see for instance the work of Gog et al. (2020) as an example of query autocompletion system that adopts mode 2.

[1] visiting the site http://www.google.com, January 12th, 2022.

Besides the matching mode adopted, autocompletion systems usually do not show all the matches to their users, which raises the necessity of providing a ranking to select the top results. Ranking can be, for instance, computed based on features such as frequencies of suggestions in the documents indexed by the system, click counts in the suggestions, number of errors in match mode 1, number of errors in match mode 2, information about the user who is typing the query and so on. Furthermore, when computing the ranking and the top results, the methods could apply pruning strategies to accelerate the computation of results. We neither discuss ranking strategies nor pruning strategies here but leave them as a future work.

Query autocompletion has been frequently studied in literature. Grabski and Scheffer (2004) studied the query autocompletion problem and proposed a retrieval model to select sentences to be shown to users from the ones that might complete the prefix query already typed. Bast and Weber (2006) (see also Bast et al. (2008)) proposed the *Hyb* data structure, a method to perform autocompletion in mode 2, processing queries word by word. Bast et al. (2021) show how to achieve autocompletion for SPARQL queries on very large knowledge bases. They do not mention error-tolerant prefix search algorithms in their work, but it could be impacted if using the data structures studied here for fast error-tolerant prefix search.

Nandi and Jagadish (2007) also studied the query autocompletion problem at the level of a multi-word phrase called mode 1, instead of completing words. They introduced a data structure named *FussyTree* to select autocomplete phrases for a given prefix. They introduce the concept of a *significant phrase*, which is used to demarcate frequent phrase boundaries from the possible suggestions. They have not implemented an error-tolerant prefix search.

Besides efforts to improve the efficiency of query autocompletion methods, there has been also much attention in the literature to improve the quality of results. Smith et al. (2017) carried out a detailed user study that shows the value of query autocompletion in shorter sessions and higher retrieval performance. Tahery and Farzi (2020) investigated the impact of customizing features related to time, location, context, and demographic features in this application. Kang et al. (2021) studied the problem of generating suggestions for query autocompletion, proposing a framework employing an *n-gram* language model at a subword-level to generate suggestions for prefixes not seen in the past. Cai and de Rijke (2016) proposed a learning to rank-based approach where features derived from homologous queries and semantically related terms are adopted to improve ranking quality. Cai and de Rijke (2016) also presented a detailed survey about query autocompletion in information retrieval.

When looking to the literature about error-tolerant prefix search methods applied to query autocompletion, several approaches (Chaudhuri & Kaushik, 2009; Ji et al., 2009; Li et al., 2011; Deng et al., 2016; Zhou et al., 2016; Qin et al., 2019) adopt tries (Fredkin, 1960), or their variations, as the search indexing structure. Typically, these methods traverse the trie using breadth-first search (BFS) and produce a list of results for each character typed by a user when submitting a prefix query. These methods maintain a set of active nodes that are associated with trie nodes and obtained with a match that supports a given error limit τ. Several algorithms proposed in the literature use this approach and differ from each other by the strategy to maintain the set of active nodes.

Chaudhuri and Kaushik (2009) and Ji et al. (2009) proposed trie-based solutions that incrementally maintain a set of active nodes associated with the trie nodes. The methods process the matches using the trie as an automaton, activating or deactivating its nodes while processing the matches. For instance, when applying this method to query autocompletion, each character typed by the user might be processed as an input to update the list of active nodes. After updating the active nodes list for an already typed prefix, the result can be reported by taking all the leaf nodes that can be reached from the active nodes in the trie, and the list of active nodes can be used to update the results when a new symbol is added to the prefix query, as the user continues to type. While both methods use the same general strategy, Chaudhuri and Kaushik (2009) propose to partition all possible queries at a certain length into a limited number of equivalent classes (via reduction of the alphabet size) and previously compute the resulting active nodes for all these classes before. This strategy is a pre-computation step to quickly start the autocompletion and reduces the cost of maintaining the list of active nodes.

The number of active nodes can be extremely high when performing error-tolerant prefix search, which can slow down the search process. The subsequent research in the topic focused on reducing this number without impacting the final set of results. Li et al. (2011) proposed ICPAN, an alternative trie-based method to reduce the number of active nodes maintained by the method in Ji et al. (2009). This reduces memory consumption and query response time by only considering the subset of active nodes with the last characters being neither substituted nor deleted.

In another effort to reduce the costs for computing active nodes, Deng et al. (2016) proposed META, which features the ability to support top-k query matches. Deng et al. (2016) designed a compact tree index to maintain the active nodes to avoid the redundant computations that occur in previous methods.

Zhou et al. (2016) propose BEVA, another trie-based method that uses an even more efficient evaluation strategy for the active nodes, which speeds up query processing by entirely eliminating ancestor–descendant relationships among active nodes. The key idea is to store the edit vector values of each active node, which allows them to store a minimal set of active nodes required to perform the edit distance computation, the so-called *boundary active nodes*. BEVA is the algorithm adopted in our study about how to perform efficient error-tolerant prefix search on burst tries, and we thus better detail it in the next section.

Hu et al. (2018) proposed a trie-based method that allows combining location aware and error-tolerant query autocompletion. Wang and Lin (2020) extended the ICPAN (Li et al., 2011) method and propose a method called *AutoEL* to support error-tolerant location-aware query autocompletion. The error-tolerant feature is enabled by applying the edit distance to evaluate the textual similarity between a given query and the underlying data, while the location-aware feature is taken by choosing the k-nearest neighbors. Like ICPAN, AutoEL is a trie-based method and can take advantage of the ideas we propose in this paper.

Trie is a fast data structure and represents a good alternative for building error-tolerant query autocompletion systems, but is also space-intensive. To reduce the storage costs while keeping a good performance of tries, Heinz et al. (2002) proposed a data structure referred to as *burst trie*. Burst tries are collections of small data structures, called *containers*, that are accessed via a conventional trie, called *access trie*. Searching involves using the initial characters of a query string to identify a particular container, then using the remainder of the query string to find a record in the container. Heinz et al. (2002) have experimented alternative data structures to store information on each container and reported that using a binary search tree was a competitive alternative.

Several researches in literature have previously shown that taking care of cache hierarchy may largely improve the performance of algorithms that deal with tries and burst tries. Acharya et al. (1999) present cache-efficient algorithms for trie search. They use different data structures (partitioned-array, B-tree, hashtable, vectors) to represent different nodes in a trie. They also adapt to changes in the fanout at a node by dynamically switching the data structure used to represent it.

Inspired by the success of previous work that explored the cache hierarchy to improve the performance of tries, we here include in our contributions a discussion about how to build tries and burst tries in a cache-friendly approach designed specifically for the error-tolerant prefix search. We discuss the application of burst tries as a possible data structure for processing error-tolerant prefix search. Burst tries were originally developed to provide fast exact dictionary matches. We here discuss alternative burst heuristics and container storage data structures for applying burst tries as indexes for error-tolerant prefix search.

Part of our study was focused on finding efficient ways of implementing the tries and burst tries. Issues on the efficient implementation of tries have been studied in the literature since they were first proposed (Fredkin, 1960). Morrison (1968) proposed the *Practical algorithm to retrieve information coded in alphanumeric*, or *Patricia trie*. In summary, a Patricia trie is a trie where the symbols are represented in bits, becoming a binary tree, and where the nodes represent only the positions where the keys differ from each other. As a result, Patricia tries considerably reduce storage costs, at a price of increasing the computational cost for search in the data structure when compared to a conventional trie.

McCreight (1976) introduced the compact versions of a trie that we name here as *compact prefix trees (CPT)*, and that are also known as *prefix trees*, or *tries* or *compact suffix trees* (Clark, 1998). The compact prefix tree reduces the storage requirement of a regular trie by removing the degree one node. Nodes containing just one child have this child collapsed to them. Edge labels of a compact prefix tree represent a sequence of characters, while edge labels in the trie represent just one character. Notice this change increases the storage cost of each node, but on the other hand, it substantially reduces the number of nodes of the compact prefix tree compared to the trie. In this paper, we present experiments comparing implementations of error-tolerant prefix search, both with tries and compact prefix trees.

When a trie or any of its variants is used to index distinct suffixes of the indexed strings, it can be called a *suffix tree*. These structures usually index all the possible suffixes of each indexed string, becoming space expensive. Compact versions are even more important when creating suffix trees. Manber and Myers (1993) introduced a representation to represent a suffix tree that stores all the suffixes in an array, referred to as a *suffix array*. This data structure was also created in a parallel research (Gonnet et al., 1992). It is a sorted array of all the suffixes of a string. Abouelhoda et al. (2004) present a detailed discussion about how to use suffix arrays as a substitute to several applications of suffix trees.

Several works in literature have discussed how to use suffix arrays for performing error-tolerant string search. The algorithms usually break the search string into consecutive and non-overlapping sub-strings named as *n-grams*. Exact matches between the *n-grams* of the search string and the suffixes indexed are used to detect matches with errors between the whole searched string and text positions (Navarro et al., 2000, 2005).

Darragh et al. (1993) proposed the *Bonsai trie*, a trie representation where the nodes are maintained in a compact global structure, a hash table, that stores all the nodes of the trie. This allows a reduction in the space required to store each trie node. Darragh et al. (1993) discuss all iterations of implementing tries and compare to their implementation using a global hash. Here we adopt the idea of creating a global data structure to both reduce storage costs and accelerate the access to trie nodes.

Besides the compact representation, other efficient implementation of tries are discussed in several contexts of applications in the literature, including name lookup in

Table 1 Sample dataset

ID	String
1	autobus$
2	autonomy$
3	auto_off$
4	book$
5	cat_dog$
6	cattail$
7	cattle$
8	cat_food$

networks (Ghasemi et al., 2018; Xie et al., 2017), general database and dictionary search (Bender et al., 2002; Binna et al., 2018) and bioinformatics (Holley et al., 2016), among others. However, we have not found specific related work discussing efficient trie building for optimizing query autocompletion tasks. As we show here, we can considerably speed up the query autocompletion search when taking into account specific characteristics of such an application when building the trie.

Other recent work has also proposed compact and efficient trie variations, but none of them addressing error-tolerant prefix search. Belazzougui et al. (2010) and Jansson et al. (2015) presented compact trie variations to produce fast and compact data structures to allow fast exact prefix match in dynamic environments, with special attention to tries adopted in efficient implementation of online Lempel Ziv text factorization Ziv and Lempel (1977).

Kanda et al. (2020) use a technique called *path decomposition* to construct cache-friendly tries that are compact and fast. Path decomposition compresses the trie by modifying its structure by first choosing a root-to-leaf path in the original trie and then associating this path with a root of a new trie. They describe how to perform an exact string search in their structure, while we are interested in performing error-tolerant prefix search.

3 Tries as indexes for string search

Tries are search trees in which the keys are usually strings with symbols from a predefined alphabet Σ, where each character of the string is stored as a label on an edge. In a trie, each path from the root to a leaf represents a string. Consider a dataset of example containing strings {"autobus$", "autonomy$", "book$", "auto_off$", "cat_dog$", "cattail$", "cattle$", "cat_food$"}, with '$' used to indicate end of string and '_' representing a blank space, illustrated in Table 1.

An example of a trie containing these strings can be seen in Fig. 2. For convenience, we have numbered the nodes in the figure just for illustrative purposes and these numbers are not part of the structure. The trie starts with a root node, and since there is no edge on such an initial node, it represents an empty string. Each string inserted has a unique path representing it in the trie. For instance, the string "cattle" is represented by the path containing the nodes numbered as 3, 6, 9, 13, 21 and 28 in Fig. 2.

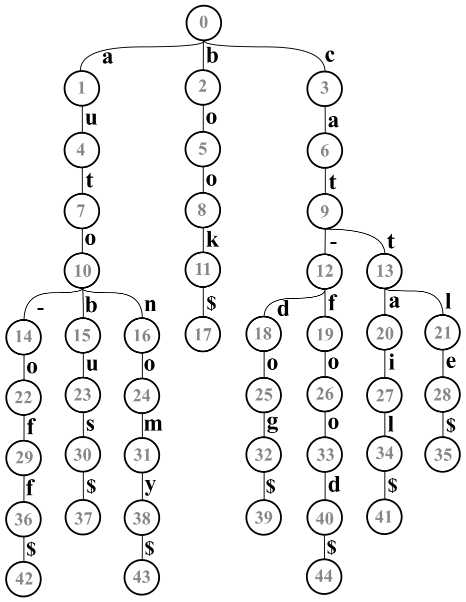


Fig. 2 A trie containing the 8 strings of our sample dataset

3.1 Exact prefix search in tries

When searching for an exact prefix match in a trie, we start from the root node and follow the path defined by the prefix searched. We say that a node which represents a match at each step of the prefix search is an *active node*. For instance, when searching for prefix “ca” using the trie presented in Fig. 2, we start at node 0, an active node that represents a match between the empty prefix search and the dataset. Then go to node 3 after processing ‘c’, and to node 6 after processing ‘a’, finishing the search at node 6. We say that node 6 is activated by the prefix query “ca”, and may keep this information when performing an incremental search, so that when the user adds a new letter to the prefix, the search can be incrementally continued from the active node 6. Notice that the search can be performed incrementally by storing previous active nodes of user searches, or started from the beginning if the previous active nodes list is not available.

Fetching is the task of processing the list of active nodes to get the list of string results of the search (Zhou et al., 2016). The fetching traverses the trie to find all the leaves that can be reached from the active nodes. Notice that fetching may be a costly operation in the search process when there are large numbers of matches or active nodes to the prefix query.

3.2 Error-tolerant prefix search in tries

To better understand recent error-tolerant prefix search methods, we first present a well-known method to compute the edit distance between two strings p and q (of lengths n and m, respectively). As explained in Zhou et al. (2016), this is a dynamic programming algorithm that fills in a matrix M of size $(n + 1) \cdot (m + 1)$. Each cell $M[i, j]$ stores the edit distance between the prefixes of lengths i and j of the two strings, respectively. The cell values can be computed in one pass in row-wise or column-wise order based on the following recurrence equation:

$$M[i,j] = min(M[i-1,j-1] + \delta(p[j], q[i]), M[i-1,j] + 1, M[i,j-1] + 1),$$

where $\delta(x,y) = 0$ if $x = y$, and 1 otherwise. The boundary conditions are $M[0,j] = j$ and $M[i,0] = i$. In the example below, to obtain the distance between the words "mid" and "main", just take the value of the position cell $M[n-1, m-1]$. The time complexity to calculate is $O(n \cdot m)$.

Ukkonen and Wood (1993) observed that the edit distance can be computed by representing only the elements in the k-diagonals of the matrix, where $k \in [-\tau, \tau]$ and τ is the maximum edit distance allowed. These values are stored in the so-called *edit distance vectors*. Given two strings of size m and n, the time complexity to compute the edit distance is $O(\tau \cdot min(n,m))$.

Recently proposed error-tolerant prefix search methods explore the general idea of computing the edit distance. However, as computing the edit distance between each pair of strings at a time would be too expensive to provide real-time search results, they usually adopt an index to simultaneously compute the edit distance between the prefix query typed and the whole set of strings in the dataset.

In our study, we adopt BEVA (Zhou et al., 2016), one of such methods, as the basic algorithm for performing error-tolerant prefix search on burst tries.

The algorithm starts with the root node as being an active node, and only this node is active up to when the prefix query p has more than τ characters already typed by the user ($|p| > \tau$). The method then computes and stores the new set of boundary active nodes after each character typed. The current list of boundary active nodes becomes inactive whenever a new character is added to the prefix query p. A scan in each of their children in the trie is performed to compute their respective edit vector values. Each child is then classified according to the value of the edit vectors found as follows:

- *terminal* - when the node is inactive and has no chance to activate other nodes.
- *inactive* - when the node does not represent a match, but its edit vector value indicates that one of its children has a chance of being active.
- *active* - when the node is inserted in the new list of boundary active nodes for the prefix query.

Nodes classified as inactive have their children recursively scanned, repeating the process until finding either active or terminal nodes in all paths derived from it. As matches can be found for paths of sizes from $|p| - \tau$ to $|p| + \tau$, the recursive process may continue up to $2\tau + 1$ levels in the trie. After finishing this computation, the updated list of active nodes can be used both to compute the answer to the current typed prefix and as the seed to compute the new list of active nodes when the user types a new character.

BEVA also features a way of quickly updating the edit vector values by using a data structure named *edit vector automaton* (EVA). EVA supports computing all the possible valid values of the edit vectors and all possible transitions between them for each possible given input scenario. As a result, it can be used to quickly update the edit vectors of active nodes when traversing the trie to compute error-tolerant query autocompletion results.

As the other algorithms that search on tries allowing errors, BEVA search performs a breadth-first search (BFS) traversal to find a list of nodes that represent matches between the prefix searched and the dataset, limiting the results to a given maximum number of errors

Table 2 Dynamic programming matrix when computing edit distance between "main" and "mid"

		0	1	2	3
		ϵ	**m**	**i**	**d**
0	ϵ	0	1	2	3
1	**m**	1	0	1	2
2	**a**	2	1	1	2
3	**i**	3	2	1	2
4	**n**	4	3	2	2

Table 3 Query processing in BEVA method to prefix query "cut" in our sample dataset

Prefix query	Active node set
∅	{0}
c	{0}
cu	{3, 4}
cut	{7, 9}

τ. The root node is the only node activated in BEVA for prefixes smaller than or equal to τ. When processing symbols at positions greater than τ, for each symbol processed, the algorithm takes the list of current active nodes, and checks for descendant nodes that match the prefix when adding this new symbol, creating a new list of active nodes with them. The list of current active nodes is then replaced by the new list found. After processing all the symbols from the prefix searched, the final list of active nodes is then used to fetch the strings from the dataset that match the query. Table 2

To better illustrate how BEVA traverses a trie, consider a search in our sample dataset using the trie presented in Fig. 2. Consider a search task allowing 1 error and the prefix query "cut". The set of active nodes achieved when applying BEVA is presented at Table 3. When starting the search, the root node becomes active for the first symbol 'c', since we allow 1 error in our example. When processing letter 'u', the algorithm takes the list of current active nodes, only node 0 of Fig. 2, and checks for descendant nodes that match the prefix after processing 'u'. Notice that after processing 'u' the root node will no longer be active. Nodes 3 and 4, which represent matches with keys starting with 'c' and "au" are activated, indicating that there is a match between all keys found in their respective subtrees and the prefix query "cu". When processing letter 't', the algorithm checks for descendants of nodes 3 and 4 to see what nodes will be activated, getting as a result only nodes 7 and 9, indicating a match between "cut" and all keys starting with "aut", represented by node 7, and "cat", represented by node 9. The step-by-step query processing was illustrated in Table 3.

Notice that BEVA updates the list of active nodes at each symbol processed from the prefix query, performing a BFS-style traversal on the trie to do so. Other researches in literature have proposed trie-based error-tolerant prefix search that traverses the trie in a BFS order, including Ji et al. (2009); Li et al. (2011); Deng et al. (2016); Hu et al. (2018); Wang and Lin (2020). The differences among these methods are in the number of active nodes at each step. BEVA is the one that activates fewer active nodes among them and Zhou et al. (2016) show that the set of active nodes maintained by BEVA at each step is minimal.

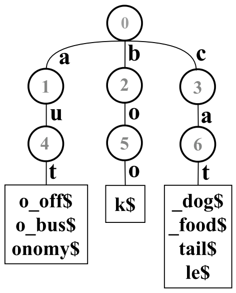


Fig. 3 Example of a burst trie with the minimum container depth (MCD) set to 3

4 Error-tolerant prefix search using burst tries

Here we discuss alternatives to implement the error-tolerant prefix search with burst tries. The idea is to view each burst trie container as a tree rooted by the node pointing to it in the access trie. This tree representation is virtual, without the need of specifically using it as the actual data structure to store the keys in the containers. With this representation, the search is performed by initially traversing the access trie, and continuing the processing in this virtual tree whenever it reaches a container. This simple representation has the advantage that tree-based search methods, such as BEVA, can be easily adapted to be used over burst tries, with the advantage of saving space when compared to a full trie representation. On the other hand, the representation presents redundant nodes when compared to a full trie, which can slow down the query processing. We discuss this trade-off in the experimental section and show that the proposed strategy leads to competitive methods, with marginal loss in time performance and a reduction in memory usage when compared to using full tries.

4.1 Burst heuristics studied

In our study, we investigate three burst heuristics used when creating the burst tries for error-tolerant prefix search:

minimum container depth (MCD): The first heuristic studied is to establish a minimum level for containers in burst tries. The idea is that by keeping longer paths in the access trie, we can speed up the query processing. The exception, of course, are the keys that contain fewer symbols than the MCD parameter, which are stored in containers at a depth determined by their sizes. Notice that when used alone, this heuristic actually produces burst tries with all containers set to the given minimum level. When combined with other heuristics, the complementary heuristic can, however, create containers at levels higher than the MCD. Figure 3 shows how the sample dataset presented in Table 1 is represented in a burst trie using the minimum container depth set to 3. Notice that in this case the number of elements in each container is not limited. This should be tuned to be large enough to allow the search to process a large portion of queries traversing the access trie nodes, speeding up the query processing. On the other hand, it must be small enough to allow a significant reduction in the total amount of memory used by the resulting burst trie. In the experimental section, we investigate this trade-off between memory usage and time performance for this parameter.

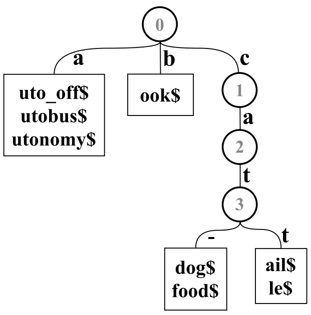


Fig. 4 Example of a burst trie limiting containers to 3 elements

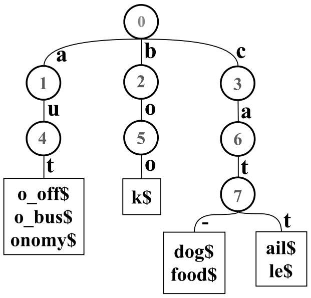


Fig. 5 Example of a burst trie with MCD set to 3 and MCK also set to 3

Regarding the query processing costs, notice that MCD does not limit the maximum number of strings in a container. Given this unbounded association, its worst case would be when all strings in the dataset are concentrated in a single container. Consider, for instance, a situation where all keys in the dataset have equal value in the first 30 symbols, a very unusual situation, and searched prefix also has length 30 and matches all keys. In this case, the number of elements in a container at any chosen MCD value less than 30 would be equal the number of keys in the dataset. Thus, the number of virtual nodes in the search would be $O(n)$, where n is the number of keys in the dataset. This provides a worst case scenario for the MCD. In practice, the keys inserted are expected to be different from each other, and the number of nodes, and virtual nodes, processed by BEVA when using a MCD tends to become close to the number processed when using the full trie.

Maximum container keys (MCK): The second heuristic studied is to limit the number of keys in each container, this heuristic was already proposed by Heinz et al. (2002). Here we study how the trie-based algorithms behave when varying the maximum number of keys allowed in each container (MCK). The lower the threshold value, the closer the burst trie is to a full trie, reducing the differences in query processing times between the full trie and the burst trie. On the other hand, a reduction in MCK also brings the burst trie's store costs closer to those required for a full trie. Figure 4 shows an example of a burst trie for the sample dataset when using MCK value set to 3. Notice that the access trie in this case contains 4 nodes.

One of the motivations for the adoption of the MCK heuristic is to reduce the computational cost in the worst case when compared to the use of MCD. MCK better controls the redundancy added when searching in the virtual trees of the containers. Given a value of MCK set to α, and assuming that BEVA activates $O(A)$ nodes when searching for a prefix in the full trie created for a dataset, it would activate at most $O(A \cdot \alpha)$ nodes in the burst trie

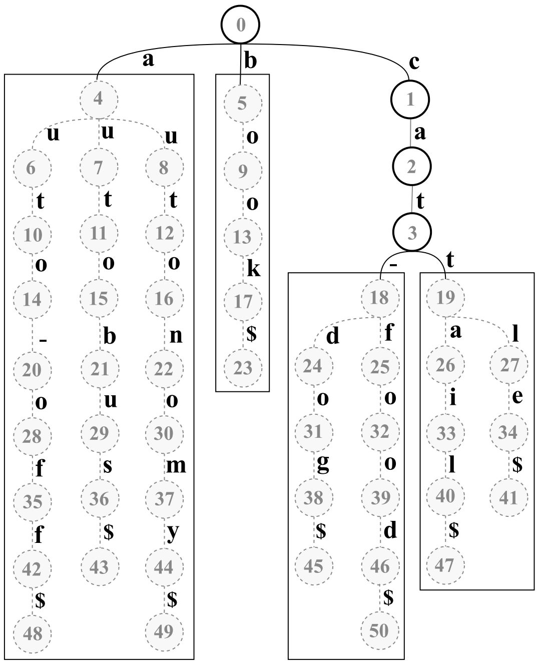


Fig. 6 Example of a burst trie limiting containers to 3 elements and representing the content in the containers as virtual trees

built with the MCK heuristic for the same dataset. As the parameter α can be considered as a constant, the asymptotic limit to the number of active nodes can be considered as $O(A)$, presenting the same computational complexity obtained when running BEVA over the full trie.

Combining MCD and MCK (MCD+MCK): The third combines the two heuristics (MCD and MCK) to produce a new burst criterion based on them. In this case, we limit the containers to only occur at a specified minimum depth, and we also limit the maximum number of elements in each container. An example using the two heuristics combined is shown in Fig. 5.

4.2 Viewing containers trees

The crucial observation that allows us to adapt burst tries to trie-based error-tolerant prefix search algorithms presented here is to see the content of each container C as a tree connected to the burst trie. This view has a root node that connects the container elements to the access trie, the edge between this root node and the access trie is labeled with the same symbol as the edge that connects the container to the access trie. Also, this view includes a path in the tree for each string of C. Given a string q from C, a node at level 1 of the tree is connected to the root of the container with an edge labeled with $q[1]$. Similarly, in this view a node at level 2 of the tree is connected to node of level 1 with an edge labeled as $q[2]$. Generalizing this idea, each node at level j of our tree view, $j > 1$, is connected to the node of level $j - 1$ with an edge labeled with $q[j]$.

We refer to this tree view as *virtual tree*, as we do not actually build or store this tree when building the burst trie. We only use this view when performing error-tolerant prefix search and its nodes are represented on demand when activated by the search algorithm.

Table 4 Query processing using the BEVA method with MCD-MCK to prefix query “cut” in a burst trie containing virtual nodes

Prefix query	Active node set
∅	{0}
c	{0}
cu	{6, 7, 8, 1}
cut	{10, 11, 12, 3}

For the same reason, we also name the nodes in the proposed view as *virtual nodes*. Notice that if multiple strings in the container have the same value of $q[1]$, starting with the first equal symbol, we view a virtual node for each of them, adding redundancy to our view. Also, any data structure can be used to store the elements of the container and, so the burst trie does not need to be modified to use BEVA and allow error-tolerant prefix search. In the following section we present a detailed discussion about how we implemented our tries and burst tries.

Figure 6 shows an example to illustrate the burst trie visualization presented in Fig. 4. In the example, only nodes from 0 to 3 are real nodes of the access trie. In the burst tries, the container root is inserted as special leaf node that connect the container to the access trie, and the root node of a container represents the entire container in the burst trie.

When looking to Fig. 6, node 4 represents the root node of the container. This container contains the suffixes {“uto_off$”, “utobus$”, “utonomy$”}. Node 6 is connected to the root by the edge labeled with ‘u’, the first position of string “uto_off$”, node 10 is connected to node 6 by an edge labeled with ‘t’. The same procedure is adopted for each of the remaining letters of “uto_off$” and also to view the remaining content of the container as a tree.

When processing queries with BEVA, the algorithm maintains and updates a list of so-called active nodes after processing each letter of the searched prefix. In BEVA this is implemented as a list of pointers to real trie nodes when processing the query. When using BEVA with a burst trie, they are pointers to the nodes of the access trie until we reach a root of a container, which is also an allocated node of the burst trie. When pointing to virtual nodes, we we point to the positions of the strings in the container (virtual nodes), instead of pointing to real nodes. So there is no extra space to store the virtual tree in the operation.

For example, if we processing a prefix “aut” using BEVA with exact match, we start with node 0 activated. That means that BEVA keeps a pointer to node 0 in the list of active nodes. After processing letter ‘a’, the leftmost container root node is activated and a pointer to node 4 is inserted in the list of active nodes. When processing letter ‘u’, the first letter of the container, we activate all virtual nodes connected to node 4 and this is done by requesting all the strings that start with ’u’ in the container. We insert pointers to the first position of “uto-off$”, “utobus$” and of “utonomy$” in the list of active nodes. These pointers are then used by BEVA to continue processing the query. The pointers can then be used to check whether or not these nodes activate their children, as we can get the next letter of each string just by adding 1 to each pointer. Thus, these pointers are used to traverse the virtual tree without allocating extra space to represent its nodes.

In summary, the only change is that instead of pointing to a real trie node (a memory address of a trie node), BEVA active nodes can now point to real nodes as well as virtual nodes, pointing to a position of an element (a string) in the container in this second case.

As another example to show how to process queries that allow errors, Table 4 presents the nodes activated in the search for the prefix “cut” allowing 1 error. At the beginning,

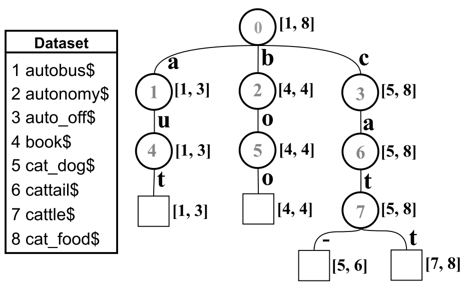


Fig. 7 Burst trie with MCD set to 3 and MCK also set to 3 with the range information to each node and container in the burst trie to our sample dataset

only node 0 is activated. After processing “c”, only node 0 is still activated. After processing “cu”, nodes 6, 7, 8 and 1 will be activated by a search using BEVA, and results will include all strings from the dataset in their subtrees. After adding the letter ‘t’ to form the prefix “cut”, nodes 10, 11, 12 and 3 are activated. This example is useful for illustrating how we adapt trie-based algorithms to perform search on burst tries. We can see that the search now activates more nodes as we add some redundancy to the tree by adding the virtual nodes. As we have show in the experiments, this redundancy does not have much effect on the query processing time, while the use of burst tries can significantly reduce the space needed to store the index. We reinforce with the given example that the virtual nodes are never allocated and no extra tree is built to represent the elements of the container when processing a prefix query with BEVA.

5 Efficiency issues

In this section we discuss possible alternatives to efficiently implement the error-tolerant prefix search methods using tries and their variations discussed in the previous sections. First, we consider a way to reduce the cost of fetching results when processing queries. The proposed optimization technique requires the dataset to be static, which means that no insertions or removals are allowed before a complete index rebuild. Second, we discuss alternative trie building solutions that are available when the dataset is static.

5.1 Reducing fetching costs

An alternative way to reduce the fetching cost is to assume that the string dataset has been sorted previously, which is acceptable for static datasets. In this case, the strings represented by each node can be stored in consecutive positions in the dataset, and we can keep the initial and final position of elements in the dataset, their occurrence interval, or just range or interval, associated with each node as information to avoid traversing the subtree when fetching the results. This idea has already been adopted by other works in literature when dealing with tries in scenarios where the dataset is static. See for instance Pibiri and Venturini (2017), and also see the work of Gog et al. (2020), which studies the use of this idea when implementing a query autocompletion system. Fig. 7 shows the trie containing range information for our sample dataset.

Storing the dataset in a sorted lexicographical order restricts the insertion and removal of trie keys and can be prohibitive for some applications. Here we assume that this is not a severe restriction to autocomplete systems, our target application, especially because the

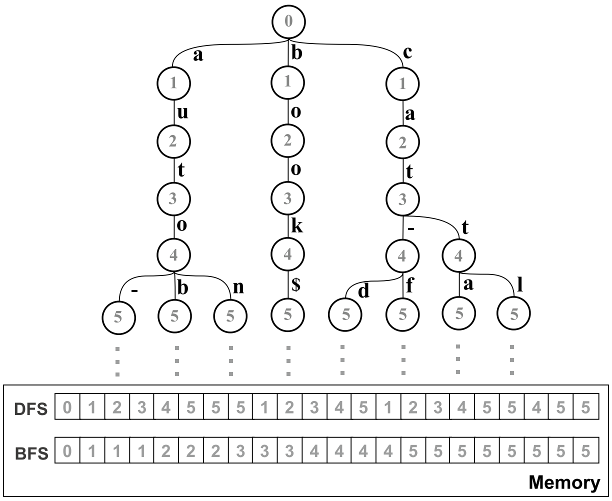

a
b
c
u
o
a
t
o
t
o
k
-
t
-
b
n
$
d
f
a
l
DFS 0 1 2 3 4 5 5 5 1 2 3 4 5 1 2 3 4 5 5 4 5 5
BFS 0 1 1 1 2 2 2 3 3 3 4 4 4 4 5 5 5 5 5 5 5 5
Memory

Fig. 8 Memory organization of nodes of distinct levels in a trie when building the index using the DFS and the BFS approaches. For illustrative purposes, nodes were labeled according to their levels in the trie

burst trie building is a quite fast process that can be periodically executed. For instance, the index building process takes just about one minute for the datasets we adopted in our experiments.

In addition to the advantage of reducing the fetching times, the use of range information can also be useful to make the burst trie representation even more compact, because instead of storing the strings in the container, we can represent them by storing only the range of keys in a container, and use this range information to get quick access to container elements directly in the dataset. For instance, when reaching the leftmost container in Fig. 7, we find a range $1 - 3$, which means that the first three strings of the dataset are stored in the container.

5.2 BFS index building

We can also create the trie nodes, or burst trie, in a cache-friendly arrangement by considering the dataset as static and pre-sorted in lexicographical order. In this section, we study alternative strategies for inserting keys into trees used as indexes in error-tolerant prefix search methods to achieve this goal.

Like other tree data structures, tries can be traversed using depth-first search (DFS) or breadth-first search (BFS). Error-tolerant prefix search algorithms that use tries, such as BEVA (Zhou et al., 2016) or ICPAN (Li et al., 2011), perform BFS as they need to find a new set of answers for each symbol typed by the user. Zhou et al. (2016) discussed an alternative implementation that uses DFS to traverse the trie and speed up situations where a user types a prefix query too fast, e.g., when the prefix query is pasted into the search box, but this is not the most common case. Furthermore, their experiments have showed that even in these specific cases, the DFS BEVA was only slightly faster. We assume that BFS is adopted as the default search strategy if we consider a user typing one character at a time.

On the other hand, given a certain dataset, the order in which keys are inserted into the trie determines the physical position of their nodes in memory, which in turn can impact the search performance. This impact is due to caching effects on the memory hierarchy. When building a trie, the most natural way to inserting keys is to create all the nodes necessary to represent a key once that key is inserted. We call this approach *DFS index building*, because the nodes are inserted in an order that resembles the DFS.

The DFS approach has an important side effect: nodes at the same depth are inserted noncontiguously, which can slow down BFS query processing due to the effects of caching on the memory hierarchy. This phenomenon is illustrated in Fig. 8, where we compare how the trie nodes are created by inserting keys using the DFS approach and using the BFS approach. In this figure, we can see that the most natural way of inserting keys into a trie, which resembles a DFS, tends to spread the nodes of equal depth in the trie across the memory used by the data structure. While this behavior is obvious, it is usually not considered a problem, since a search for an exact key in a trie is also performed in a DFS order.

However, the error-tolerant prefix search algorithms access the trie nodes one level at a time, which means that this access will not be contiguous unless the trie index building approach also creates the nodes in the same order. The relative distance in terms of memory allocation of nodes at the same depth may increase with the number of nodes inserted in the trie. As a result, nodes at a same depth may span different levels in the cache system, and the BFS used for query processing is likely to produce a high rate of cache misses. Algorithms based on the DFS approach are likely to create a data structure that is not cache-friendly for error-tolerant prefix search applications, that is, that does not take advantage of the cache system.

To address this issue, we present an alternative approach to building index trees for error-tolerant prefix search that inserts all keys in parallel. We call this approach *BFS index building*, whose goal is to reduce the relative distance of nodes at the same depth in terms of memory allocation. In BFS we start by inserting the first character of all keys in the dataset, then we insert the second character and so on. As a result, the nodes at each depth become contiguous in memory, creating a more cache-friendly data structure. At the end of the process, the position of the trie nodes in memory is sorted by their depths, as illustrated in the lowest vector in Fig. 8. We show in the experiments section that this simple procedure has a great impact on the time performance of the query autocompletion task.

6 Experiments

This section presents the experiments we carried out to evaluate the performance of query completion systems using BEVA as the error-tolerant prefix search algorithm and the burst tries with the three burst heuristics studied, as well as comparing them with the use of different trie data structures in the context of error-tolerant prefix search. We executed all the experiments on a machine with an Intel Xeon E5-4617 processor (2.90 GHz), 64 GB of RAM memory. The machine cache sizes are as follows: L1d of 32 KB, L1i of 32 KB, L2 of

Table 5 Examples of prefix queries and suggestions of JusBrasil and DBLP datasets

Dataset	Prefix queries	Suggestions
JusBrasil	1. indubio pro	1. in dubio pro reu
	2. viajem durante	2. viagem durante atestado medico
DBLP	1. infprmation reti	1. information retrieval model for crime investigation
	2. he design and sim	2. the design and simulation of beam pumping unit

Table 6 Statistics about query suggestions and prefix queries typed by users in the JusBrasil dataset

File	Size (bytes)	Items	Avg item len
Query suggestions	633,078,803	23,374,740	27.0
Prefix queries	12,189,441	648,264	18.8

256 KB, L3 of 15,360 KB. The operating system is Ubuntu 18.04.1 LTS. The algorithms were implemented in C++ and compiled using gcc 7.4.0.[2]

6.1 Experiments setup

6.1.1 Datasets

We present experiments to evaluate the performance of the studied data structures and algorithms using three distinct datasets. Most of the experiments are reported using a query autocompletion suggestion dataset extracted from JusBrasil.[3] We also report results using two synthetic datasets adopted in previous research articles, DBLP[4] and UMBC.[5] JusBrasil[6] is a Brazilian law-tech company that provides a vertical search service for their users. The query suggestions for JusBrasil contains 23, 374, 740 items and 648, 264 logs of prefix queries submitted to their autocompletion system. It contains the prefixes typed by their users before issuing the queries to the JusBrasil search engines. The query autocompletion system of JusBrasil receives a query whenever the user types a symbol in the prefix given in the search box. Table 6 presents details of the JusBrasil dataset.

Table 5 presents two examples of prefix queries and suggestions that belong to JusBrasil and DBLP datasets. The first example for the JusBrasil dataset is "indubio pro", a prefix for a query that writes the words "in" and "dubio", with the wrong spelling with a single error. This is quite a common error present in the query autocompletion log of JusBrasil. The prefix matches with one error with "in dubio pro reu", one of the alternative suggestions most clicked by the users, and that matches with the prefix with 1 error. In the second

[2] https://github.com/vdbergg/bursttries-for-autocompletion.

[3] Jusbrasil can make its dataset available for experiments. Send an email to datasets@jusbrasil.com.br for further instructions.

[4] https://dblp.uni-trier.de/faq/How+can+I+download+the+whole+dblp+dataset, dataset release dblp-2019-04-01.xml.

[5] https://ebiquity.umbc.edu/resource/html/id/351/UMBC-webbase-corpus

[6] http://www.jusbrasil.com.br.

Table 7 Hit rate (%) for relevant queries in the JusBrasil dataset as we vary the number of errors allowed in the search

Hit rate for relevant queries (%)			
$\tau = 0$	$\tau = 1$	$\tau = 2$	$\tau = 3$
72.05	87.83	94.33	95.88

example of prefix, the word "viajem durante" is wrongly spelled by the user, and we show the correct suggestion that would match the prefix with just 1 error.

The suggestions contain possible complete sentences available for the query autocompletion. The prefix queries contain only the prefixes typed by the users when interacting with the search box. For some queries, the user types only a small prefix of the query he/she intends to send, gets the suggestion from the JusBrasil query autocompletion system, and then clicks on a suggested option. For others, the query autocompletion suggestions are not selected by the user for any of the prefixes typed by him/her. Notice that the system changes the suggestion set for each new letter typed by the user in the search box. The user may also submit a query directly to the JusBrasil search engine without selecting any of the suggestions, as in other search systems available on the Web. The prefix queries described in Table 6 always contain the longest (that is the last) prefix typed by the user before selecting a suggestion or submitting a query directly to the search engine without selecting a prefix. We can see that the average prefix size is about 18.8 characters, while the average query suggestion size is 27.0. The suggestions are both extracted from query logs and from the law-tech dataset, including names of people and companies found in the dataset and also topics suggested by specialists in the area.

Table 7 shows the importance of performing an error-tolerant prefix search when performing query autocompletion in the JusBrasil dataset. To evaluate such importance, Table 7 presents the percentage of matches between query suggestions selected (clicked on) by users when typing a prefix query in the JusBrasil system when varying the number of errors allowed in this match. These experiments show the importance of allowing errors in the system. This is possible because JusBrasil already allows errors in its query autocompletion system.

As shown in Table 7, if the query suggestion does not allow errors, it would provide in its set of results only 72.05% of the suggestions clicked on by the users in JusBrasil. This match percentage increases with the number of errors allowed. The increase in the number of matches is high from 0 to 1 error, and from 1 to 2 errors. From 2 to 3 errors, the match percentage does not increase that much. The conclusion of these results is that the introduction of errors has a large impact on the capacity of the system to suggest correct queries. Of course, even with exact match, the autocompletion system needs to apply a ranking function to select the best results for the users. The ranking functions, the possible features adopted in the ranking, and the possible pruning strategy that may be adopted for selecting the best prefixes are not in the scope of this research. However, the methods described here will form the basis for implementing the query autocompletion systems.

Although there are some large query logs publicly available, it is hard to find good public datasets with real query logs for performing experiments with query autocompletion applications, with information, for instance, about the size of prefixes typed by a user before submitting a query, and with actual errors submitted by the users to the system that may be fixed by error-tolerant query autocompletion engines. The researches in literature that study prefix match in this scenario usually adopt public datasets that contain no logs of prefix queries. In such cases, they create a set of queries for the experiments. To avoid

Table 8 General statistics about the synthetic datasets adopted in the experiments

Dataset	Size (bytes)	Items	Avg item len (bytes)
DBLP	334,999,905	4,378,548	76.5
UMBC	17,427,838,773	38,449,902	453.2

presenting experiments with only the JusBrasil dataset, the only dataset containing real query logs that we had available, we have also included two datasets adopted in a previous work. For comparison purposes, we created queries using the same strategy described by this previous work.

The datasets chosen are DBLP and UMBC, and they were previously adopted in articles that studied error-tolerant prefix search methods. We classified them as synthetic, since these two datasets do not contain query logs and were neither extracted from a real case query autocompletion service. Table 5 presents examples of two prefix queries and suggestions that belong to DBLP. They are useful to show how we generate queries in the synthetic dataset. We may remove, add or substitute symbols at any position of the synthetic prefix queries. For instance, we have the prefix query "infprmation reti", which was generated from the query suggestion "information retrieval model for crime investigation". In this case, we can see that the character 'o', was substituted by character 'p' (in fact it could be any character) and the last character 'r', was deleted from "retri".

DBLP contains about 4.3 million computer science publication records. For the experiments, we adopted only the title of each publication. DBLP was adopted in the experiments presented by Chaudhuri and Kaushik (2009); Ji et al. (2009); Li et al. (2011); Xiao et al. (2013); Qin et al. (2019). UMBC[7]: The UMBC WebBase Corpus is a dataset containing a collection of English paragraphs with over three billion words processed from the February 2007 crawl of the Stanford WebBase project. UMBC was adopted in the experiments presented by Zhou et al. (2016); Qin et al. (2019). Table 8 presents detailed statistics about two datasets. In all cases, we have removed duplicated items from the datasets.

While these two synthetic datasets are not real case query autocompletion collections, we use them as complementary experiments since they were previously adopted in the literature. For each synthetic dataset, we created 1,000 queries using the same procedure adopted in a previous work that adopted them to perform experiments with query autocompletion methods. Using this approach we create an experimental environment that is close to the ones adopted in the previous work. We followed a procedure adopted by Chaudhuri and Kaushik (2009), which extracted 1,000 items from the dataset to be used as the base for the prefix queries and randomly introduced errors in these items. For each edit distance threshold tested, we generate a set of queries including the randomly generated errors. Experiments with distinct prefix sizes are performed by extracting the prefixes from these queries.

Queries from these datasets were not extracted from a query log and neither simulate any particular distribution of errors in a query log. However, while the methodology for creating the queries does not guarantee a reproduction of user behaviors when interacting with a real case autocompletion dataset, the inclusion of these two datasets is important because they have been used for experiments in previous articles.

[7] https://ebiquity.umbc.edu/resource/html/id/351/UMBC-webbase-corpus.

6.2 Indexing alternatives studied

We have completely implemented the BEVA (Zhou et al., 2016) method that is among the best algorithms proposed to be used with tries.[8] BEVA was adopted as the algorithm in the comparison among the studied data structures, including the variations of burst tries, the full trie and the CPT.

We have experimented with data structures studied here in two distinct scenarios. The first one considers that the dataset can be previously sorted and that no insertions or deletions of keys will be performed between index rebuilding tasks. We name this scenario as *static*, and use all the optimizations we studied for static index building on it, including the range representation and the BFS index building described in Sect. 5. We also consider that the dataset is lexicographically sorted, so we can use ranges by position of the items. The second scenario is to consider that insertions or deletions of items are allowed, and the dataset may be changed before a complete index rebuilding. This second scenario does not allow the use of optimizations described in Sect. 5, and so the fetching step was implemented with a traversal of the subtrees of the nodes where the matches occur to find all the suggestions that match the query. We name this second scenario as *dynamic*. It does not allow the range representation and the BFS index building described in Sect. 5. In the dynamic scenario we adopted linked lists to represent the elements in the container.

We experimented with three heuristics for the burst process, applying the minimum container depth (MCD); the maximum container keys (MCK); and the combination of them (MCD-MCK). Besides the burst trie methods, we also have experimented the use of BEVA with full tries and compact prefix tries (CPT). The CPT method was included in the experiments, as it is known for being an alternative compact trie representation. It is also adopted in previous articles that implement error-tolerant prefix search algorithms (Zhou et al., 2016).

Finally, we also have included experiments with suffix arrays. We stress that suffix arrays have not been adopted in recent previous work that studied error-tolerant prefix search in the context of query autocompletion applications. However, we have decided to include them in the experiments for comparison purposes, since they are a data structure applied to approximate string matching problems. We adopted another algorithm when using the suffix array data structure to perform error-tolerant prefix search, since we found no time-efficient alternative to adapt it to BEVA. We applied an *n-gram* approach, as described in Navarro et al. (2000, 2005).

In this approach, the suffix array indexes all positions of all suggestions in the dataset. Given a prefix query p, it is divided into $\tau + \alpha$ consecutive and non-overlapping sub-strings (the *n-grams*), τ being the number of errors allowed and α being a parameter to be calibrated according to the application. An exact match search for the occurrences of each n-gram of the prefix p is performed to find suggestions in the dataset that have potential matches with p. Suggestions that match with at least α n-grams of p are considered as potential matches. Let $pos_{(b,q)}$ be the position where a sub-string b starts within a string q, assuming that q contains b. Matches with an n-gram g that occur in p at position $pos_{(g,p)}$ are filtered according to the matching position of g at each suggestion q_i. A suggestion q_i is accepted as a potential match only if g occurs in q_i at position $pos_{(g,q_i)}$, such

[8] We have validated the performance and correctness of our BEVA source code by comparing it to the original binary code provided by Zhou et al. (2016). Results indicate our implementation is even faster than the binary code provided.

Table 9 Average prefix query processing and fetching times (ms) per query in the JusBrasil dataset when using BEVA, varying the full trie index version

Methods	Avg processing and fetching time per query (ms)					
	$\tau = 1$		$\tau = 2$		$\tau = 3$	
	Processing	Fetching	Processing	Fetching	Processing	Fetching
dynamic	0.16 $\pm$ 0.001	0.095 $\pm$ 0.0043	2.24 $\pm$ 0.0122	0.184 $\pm$ 0.0062	19.48 $\pm$ 0.0934	0.316 $\pm$ 0.0083
range+DFS	0.14 $\pm$ 0.0008	0.003 $\pm$ 0.0001	2.13 $\pm$ 0.0116	0.008 $\pm$ 0.0002	19.01 $\pm$ 0.0914	0.015 $\pm$ 0.0003
range+BFS	0.11 $\pm$ 0.0006	0.004 $\pm$ 0.0001	1.35 $\pm$ 0.0067	0.008 $\pm$ 0.0002	9.15 $\pm$ 0.0476	0.015 $\pm$ 0.0003

Table 10 Average fetching times (ms) per item retrieved in the JusBrasil dataset when using BEVA, varying the full trie index version

Methods	Avg fetching time per item (x10^{-3}ms)		
	$\tau = 1$	$\tau = 2$	$\tau = 3$
dynamic	0.416 $\pm$ 0.005	0.401 $\pm$ 0.006	0.475 $\pm$ 0.008
range+DFS	0.226 $\pm$ 0.004	0.206 $\pm$ 0.005	0.216 $\pm$ 0.006
range+BFS	0.233 $\pm$ 0.005	0.192 $\pm$ 0.004	0.158 $\pm$ 0.004

that $pos_{(g,p)} - \tau \leq pos_{(g,q_i)} \leq pos_{(g,p)} + \tau$. We have experimented with values of α in preliminary experiments and have chosen α as 1, which means we minimize the number of n-grams and maximize their size. The algorithm finishes with a sequential prefix match between p and each potential match to confirm or not the match.

6.2.1 Evaluation trie building optimizations

We performed experiments to compare the impact of the optimizations we proposed for the static scenario, comparing the time performance of the trie building with and without the optimizations proposed. We report in this section experiments with full trie and using only the JusBrasil dataset, since conclusions were similar when comparing versions of CPT and burst tries with and without the optimizations proposed.

Table 9 shows the results of the experiments. We report the processing time and the fetching time separately. The processing time is the time taken to find the set of active nodes. The fetching time is the time taken to get the query suggestions from the set of active nodes. In the remaining experiments of this article we will present the time for processing queries without separating the fetching times.

The fetching times reported in these experiments consider that the algorithm fetches only up to 10,000 results. We separated the fetching time to better illustrate the advantage of using the range optimization. It considerably reduces the fetching times, especially for queries allowing more errors. For instance, when $\tau = 3$, the fetching time for the

Table 11 Average cache misses per prefix query in the JusBrasil dataset for τ=3

Methods	Memory cache misses	
	D1	DL
range+BFS	2,757,363,310	2,294,981,511
range+DFS	3,657,308,594	3,211,465,633

range+DFS was only 0.015 milliseconds, while the fetching time in the dynamic version was 0.31 milliseconds, more than 20 times slower than range+DFS.

Notice that the trie building strategy considerably affects the performance of the prefix match. In case of range optimization, the gain is restricted to a reduction in the fetching times. The gain when adding BFS optimization is a natural consequence of better using the memory hierarchy.

Both range+DFS and range+BFS were developed for the static scenario. BFS organizes the nodes of the same depth contiguously, and in the same order in which BEVA traverses them. The gain of adding this optimization increases with the edit distance threshold τ, since the number of nodes to be traversed in each depth level of the trees also increases with τ, providing an advantage to BFS trie building. Achieving better performance for higher values of τ is important because these are the most expensive queries for autocompletion.

We emphasize that the algorithm for processing queries is exactly the same when using all the experimented versions.

The fetching time values presented in Table 9 shows the average fetching time per query. The reader may also be interested in knowing the fetching performance of the methods by item fetched, this number is presented in Table 10, where we can see the relative performance when comparing is almost the same as the ones achieved when reporting the average fetching time per query. The relationship does not change much because the number of fetched item does not change when switching from one method to another.

To better understand the reasons for the difference in performance achieved by the BFS and DFS index building strategies, we investigated the hypothesis of better using the cache memory system. The results confirmed our hypothesis. To illustrate this issue, in Table 11 we present the number of cache misses when processing queries in the JusBrasil dataset for $\tau = 3$. Data were obtained with the CacheGrind program.[9] It simulates a machine with independent first-level instruction and data caches and a unified second-level cache. As the misses in instruction cache were virtually the same when comparing range+BFS and range+DFS strategies, we present only the data cache results (D1 and DL). The CacheGrind simulated cache adopted has D1 cache size of 32 KB, block size of 64 bytes, and is 8-way associative and DL cache size is 15,360 KB, block size 64 bytes and is 30-way associative.

When comparing the results using tries, we can see a reduction of 24.5% at D1 and 28.5% at DL in the average number of cache misses, being a considerable decrease in the number of cache misses. While the gain achieved by range+BFS over range+DFS strategy depends on the machine's hardware cache strategy and configuration, still the experiments are useful to illustrate the potential benefits of using the BFS strategy for building the tries and the burst tries.

[9] https://valgrind.org/docs/manual/cg-manual.html.

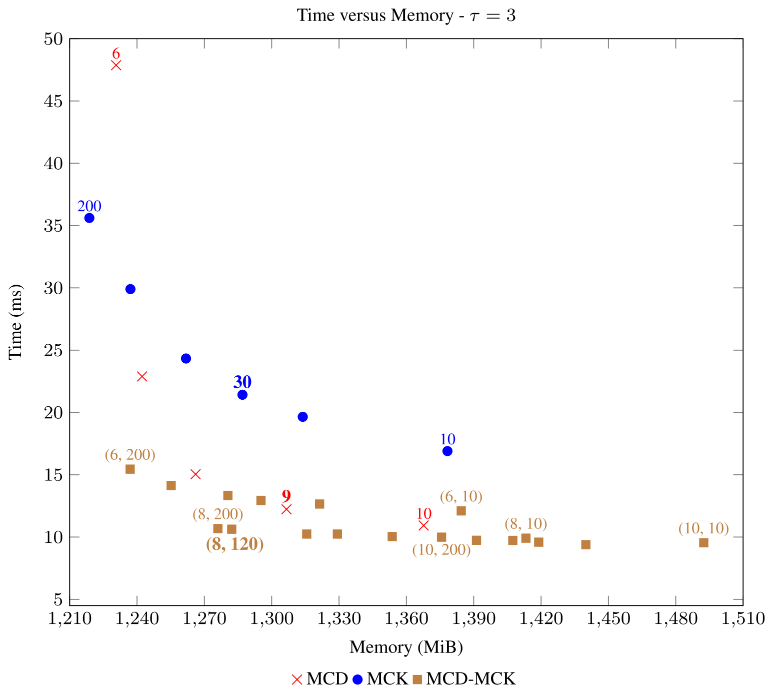


Fig. 9 Trade-off between time performance (ms) and memory usage (MiB) when processing queries with BEVA in JusBrasil dataset with data structures MCD with values varying from 6 to 10, MCK with values varying from 10 to 200 and MCD-MCK with MCD values 6, 8 and 10 and MCK varying also from 10 to 200

Table 12 Selected parameters for BEVA using MCD, MCK and MCD-MCK heuristics in JusBrasil dataset

Method	Parameters
MCD	9
MCK	30
MCD-MCK (MCD,MCK)	(8, 120)

We have performed similar experiments to assert the impact of the optimizations in CPT performance as well as in the burst trie performance. Conclusions were similar to the ones achieved with the full trie, with the range+BFS version being the faster one. We have decided to not report these comparison of results, since conclusions were similar to ones achieved for the full trie. Given the gain in the performance yielded by the range+BFS optimizations, we adopt this index building strategy in the remaining experiments for full tries, burst tries and CPTs.

6.2.2 Burst trie parameters selection

We start by discussing the parameter selection for burst tries and name the variations as MCD, MCK and MCD-MCK. We adopted a procedure for choosing the parameters using a separate set of 200 queries to study the effects of parameter variation in the performance of

Table 13 Processing time (ms) and memory usage (MiB) when using BEVA combined with MCD, MCK and MCD-MCK to process prefix queries in the JusBrasil dataset

Methods	MEM (MiB)	Time (ms)		
		$\tau = 1$	$\tau = 2$	$\tau = 3$
MCD	1,302.8	0.76 ± 0.004	2.88 ± 0.008	12.7 ± 0.029
MCK	1,282.1	0.2 ± 0.001	2.89 ± 0.015	21.84 ± 0.101
MCD-MCK	1,278.3	0.16 ± 0.001	1.65 ± 0.01	10.86 ± 0.066
dynamic-MCD	5,666.1	3.51 ± 0.119	23.88 ± 0.251	138.85 ± 0.837
dynamic-MCK	3,979.9	2.7 ± 0.013	29.43 ± 0.102	112.24 ± 0.392
dynamic-MCD-MCK	4,174.8	1.37 ± 0.009	21.99 ± 0.092	118.4 ± 0.419

Table 14 Processing time (ms) and memory usage (MiB) when using BEVA combined with MCD-MCK (set to 8 and 120, respectively), full trie, CPT and suffix array in the JusBrasil dataset

Methods	MEM (MiB)	Time (ms)		
		$\tau = 1$	$\tau = 2$	$\tau = 3$
MCD-MCK	1,278.3	0.16 ± 0.001	1.65 ± 0.01	10.86 ± 0.066
full trie	4,912.7	0.13 ± 0.001	1.53 ± 0.013	9.34 ± 0.026
CPT	1,820.1	0.23 ± 0.001	2.63 ± 0.008	18.82 ± 0.050
suffix array	7,470.0	38.09 ± 5.587	338.21 ± 21.103	1590.41 ± 58.597

the trie building strategies and analyze the relationship between time and memory usage. The parameter selections are presented just for the JusBrasil dataset. We have also performed the same parameter selection procedure for the other two collections and conclusions about the relative performance of the methods are essentially the same. We decided to only show results in JusBrasil to avoid too much redundant information, and also because it is the only real case query autocompletion dataset we adopted in this study.

We studied the MCD values varying from 6 to 10. For MCD values smaller than 6 the time achieved was too high and for values higher than 10, the memory usage was also too high. Then we have decided to not plot them. We also studied the MCK parameter, varying it from 10 to 200. For MCD-MCK we experimented with the same ranges of values used for both MCD and MCK.

Results are presented in Fig. 9. We report the variation in time for processing queries and memory usage as we vary the parameters.

Based on the results, we selected the parameters for each heuristic and dataset to be adopted in the remaining experiments. In all cases, the parameters were chosen by taking a value that provides a good trade-off between memory requirement and time performance, taking the points closer to the origins in both axes. We notice that other criteria or methodologies could be used to select the parameters. For instance, a test in a production system could lead designers to reach the maximum performance of the methods. Furthermore, the results indicate that such selection would not represent a challenge in practical applications. The parameters selected in our experiments to be adopted in the JusBrasil dataset are presented in Table 12 and marked in bold in Fig. 9.

Table 13 presents the time performance and memory consumption of each variation of burst trie experimented using the parameters selected when running prefix queries for JusBrasil. In this table we report both the dynamic scenario, with the burst trie indexes being built without optimizations and the static scenario, when we adopt both optimization strategies. The strategy MCD-MCK achieved a better combination of time performance and memory usage for JusBrasil in both scenarios. This table is also useful to reinforce the differences in time performance between the methods with or without the optimizations proposed here. We also compared the heuristics when using DBLP and UMBC datasets, and conclusions about the best option were the same. We decided to not show these comparisons to avoid presenting redundant information. Given the results, MCD-MCK is the option chosen for comparing our burst trie implementations in the remaining experiments.

6.3 Comparing burst trie with other trie representations

In this section we compare the studied data structures, including the burst trie with MCD-MCK heuristics, full trie, the compressed prefix trie (CPT) and the suffix array, when processing queries with the JusBrasil dataset. Burst trie using MCD-MCK heuristic was adopted with the best parameters found in the previous section. Table 14 shows the query processing time achieved by each compared data structure when processing queries in the JusBrasil dataset, varying the number of errors from 1 to 3. Each prefix query adopted is submitted exactly as typed by JusBrasil users. Values reported represent the cumulative time of each character in the prefix query to obtain the final results. The times are reported with a confidence interval using 99% of confidence. We present for these experiments the time and memory required when using the alternative data structures combined with the BEVA algorithm, since it minimizes the number of active states when processing queries.

We noted that MCD-MCK greatly reduced the memory requirement when compared to full trie and CPT, being the method that required less memory usage. MCD-MCK used only 26.0% of the memory requirement of full trie, while it was 16.27% slower. In addition, the query processing times in MCD-MCK are twice as fast as the times in the CPT method when considering the most expensive query option of allowing 3 errors. Compared to CPT, MCD-MCK was not only faster, reducing the time by about 42.29%, but also reducing the memory requirement to only 70.23%.

This positive result occurs because the BEVA algorithm traverses quite a minimized set of nodes that, after being visited, do not require to be maintained, reducing the overhead of creating redundant nodes when processing our virtual tree representation of containers. The most important conclusion is that MCD-MCK, a variation of burst trie with range optimization and BFS index building, largely reduced the memory usage while keeping the prefix processing times similar to the ones achieved when using the full trie. Considering the memory required by MCD-MCK, it becomes an extremely attractive alternative for query autocompletion, since it uses far less memory and reaches performance close to the full trie when BEVA is implemented with it.

When comparing MCD-MCK to the use of the compact prefix tree (CPTs), it was faster for all the experimented scenarios. The time performance difference between CPT and MCD-MCK was significant, as was the memory requirement. Another important aspect of MCD-MCK is that the confidence intervals achieved are not as high compared to those achieved when using the full trie. This result is important, since it shows there

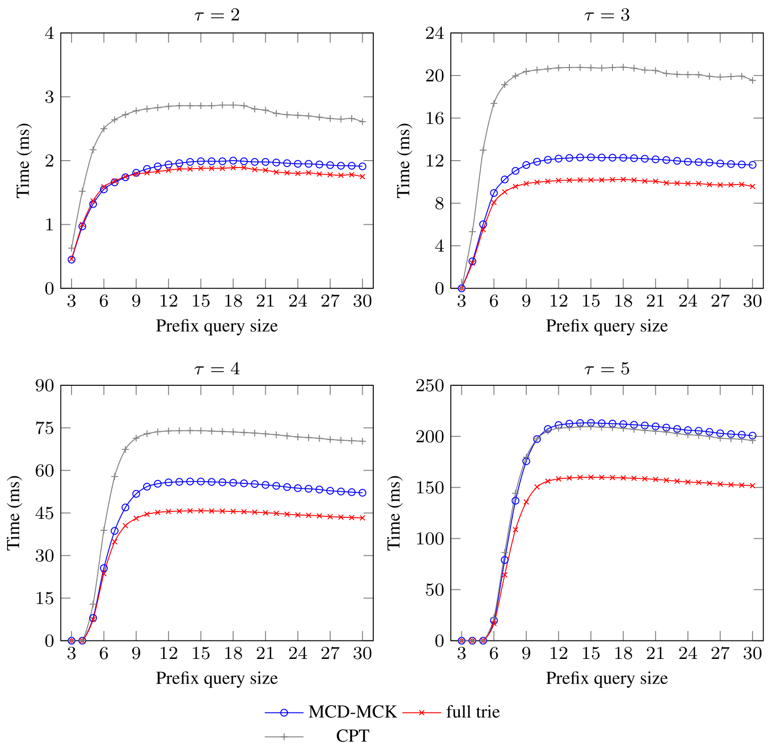


Fig. 10 Time performance (ms) of BEVA using MCD-MCK (set to 8 and 120, respectively), and full trie when processing prefix queries and varying the prefix query size and the number of errors allowed in JusBrasil dataset

is not much variation among prefix query processing times when comparing the use of the full trie, indicating that time is expected to be quite stable among distinct queries. This conclusion is also important to further validate the parameter selection procedure adopted here. As we selected the parameters using a separate set of queries, such stability in results contributes to the success of the procedure adopted.

When comparing the performance of the suffix array, which uses an *n-gram* approach for searching, we can see that the time for processing smaller prefixes is prohibitive if considering the limit of 100 ms for query autocompletion. The suffix array becomes more competitive for larger prefixes, however with a performance worse than the other data structures experimented. The suffix array also required far more memory than the burst trie with MCD-MCK heuristic and CPT. The *n-gram* approach required that the suffix array pointed to each suffix of each query suggestion present in the dataset, making the suffix array demand a large amount of space in memory.

6.3.1 Performance when varying prefix sizes and number of errors

We investigated how the time for processing queries increases when using MCD-MCK and the full trie as we increase the query prefix sizes and also as we increase the edit distance thresholds (τ). Fig. 10 presents the results of experiments, varying the prefix

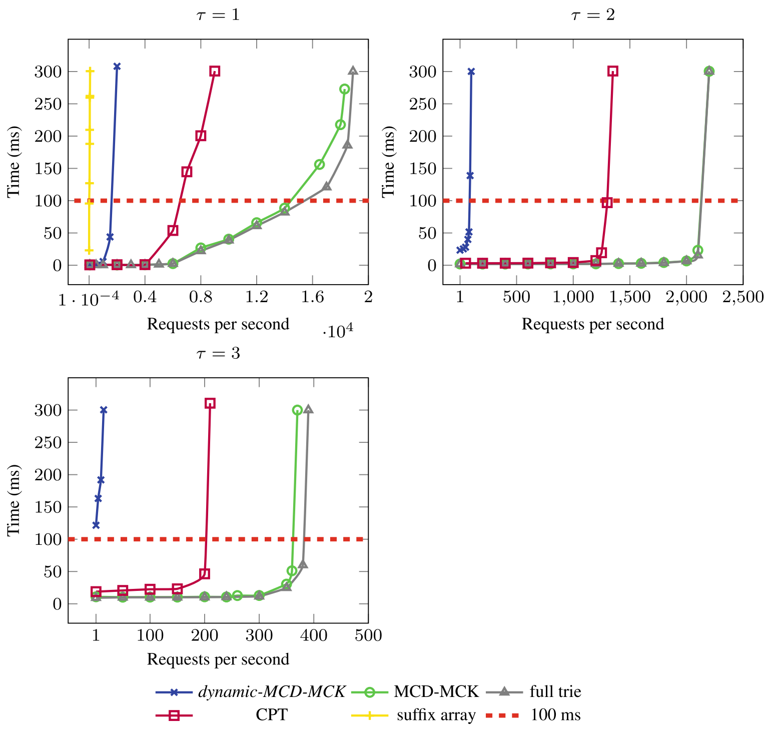
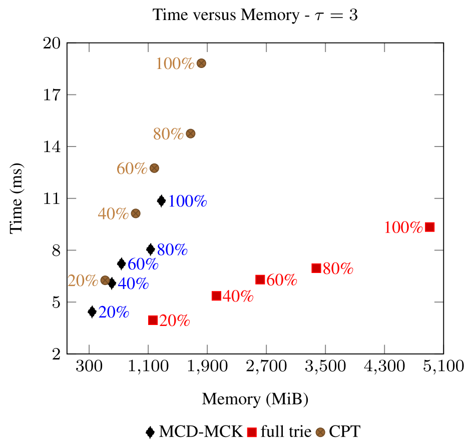


Fig. 11 Processing time (ms) as the number of requests per second increases for τ varying from 1 to 3 in the JusBrasil dataset

size from 3 to 30 and for edit distance thresholds (τ) 2, 3, 4 and 5, comparing the performance of MCD-MCK with the full trie in the BEVA method when running the experiment over the JusBrasil dataset. The suffix array was not included given its large difference in time performance, which would make it difficult to see the comparison among the other methods. We can see that the method performance is still comparable to the performance of the full trie, and that the times for processing queries do not vary much when processing a larger prefix. This result is important because larger prefix queries require more access to the virtual nodes, which could have a negative impact in the time performance when compared to the full trie. The good performance of the methods even for larger prefixes occurs because we expect fewer addition of redundant nodes as the query processing method reaches deeper nodes of our burst trie implementations. Most of the redundancy added by the methods is in the first levels of the trees containing the virtual nodes. This also explains the plateau that we see in Fig. 10.

As was expected, the time performance of the compact trie representations was a little worse when compared to the full trie. However, the differences are not so high, and do not change much for long prefixes, being almost stable as the prefixes increase from size 9 to 30. This is an important finding, especially for the larger prefixes, where MCD-MCK accesses virtual nodes more frequently. For smaller prefixes, the differences are even smaller, since there is not much access to virtual nodes in MCD-MCK. We can see that these findings also hold when varying the edit distance threshold. We summarize

Fig. 12 Time performance (ms) and memory usage (MiB) when processing queries with BEVA and indexing increasing percentages of the JusBrasil dataset (20%, 40%, 60%, 80%, 100%) with data structures MCD-MCK (set to 8 and 120, respectively), full trie and CPT



the results of the experiments concluding that the MCD-MCK method achieves time performances close to the use of full tries, while using far less memory in the dataset tested.

Finally, when experimenting with an increasing number of errors we can see that the performance of the MCD-MCK is degraded for higher error levels. For instance, MCD-MCK gets almost the same performance as CPT when processing queries with 5 errors. Hopefully, queries with higher error levels may not make sense for query autocompletion tasks, since they may bring an increasing number of matches to suggestions that may not be related to the user's intention.

We have also experimented with these variations in prefix query size and edit distance for the other datasets, but omitted them, since the conclusions are the same, with the time performance of MCD-MCK and full trie being close in all experimented parameters and collections.

6.3.2 How do methods affect scalability ?

A good question is whether the usage of burst tries static affects the throughput of systems to deal with high query workloads or not. We performed experiments with the methods submitting queries using Vegeta,[10] which is a versatile HTTP load-testing tool built out of a need to drill HTTP services with a constant request rate, to compare throughput achieved when using each of the data structures compared in this article.

Figure 11 presents the results achieved by BEVA with experimented data structures when processing queries with τ values 1, 2 and 3. Times reported here include the whole server response times, including communication and other related times necessary to produce the answers. The server was implemented to compute the full set of results, but to return only up to 100 results to avoid an excessive increase in communication time. Query autocompletion services usually send just the top results to the users for each query prefix, this restriction makes the experiment closer to real scenarios.

[10] https://github.com/tsenart/vegeta.

Table 15 Processing time (ms) and memory usage (MiB) to mode word by word when using BEVA combined with MCD-MCK (set to 8 and 120, respectively), full trie, CPT and suffix array when indexing the JusBrasil dataset

Methods	MEM (MiB)	Time (ms)		
		$\tau = 1$	$\tau = 2$	$\tau = 3$
MCD-MCK	54.4	0.08 ± 0.000	0.91 ± 0.004	4.85 ± 0.029
full trie	196.4	0.09 ± 0.002	0.95 ± 0.016	5.06 ± 0.101
CPT	53.8	0.13 ± 0.002	1.24 ± 0.022	7.05 ± 0.137
suffix array	186.2	1.9 ± 0.107	17.08 ± 0.483	46.74 ± 0.971

An important reference in this experiment is to check when the server response time reaches the limit of 100 milliseconds (Miller, 1968) considered acceptable for the autocompletion services. In this experiment, we have also included the best burst trie implemented by us for the scenario where the range and BFS optimizations are not allowed, we present it as dynamic-MCD-MCK. As shown in Fig. 11, dynamic-MCD-MCK and CPT were the first methods to exceed the 100-millisecond limit in the three values of τ experimented. MCD-MCK and the full trie, on the other hand, supported similar workloads in the three scenarios. For instance, when $\tau = 3$ the full trie and MCD-MCK were able to process more than 350 requests per second with responses below 100 milliseconds, while CPT was able to only process less than 200 requests per second. As expected, the suffix array resulted in the worse performance among the experimented data structures, and its results for $\tau = 2$ and $\tau = 3$ were not plotted because they were far above the limit of 100 milliseconds even for smaller workloads.

6.3.3 Performance when increasing the size of the dataset

We have also experimented with the variation of time and memory for MCD-MCK, CPT and full trie data structures as we increase the percentage of the JusBrasil dataset indexed from 20% to 100%. The suffix array was not included given its large difference in time performance, which would make it difficult to see the comparison among the other methods. Fig. 12 shows how the methods behave. While MCD-MCK presents a performance close to the full trie, we realized that the ratio between the time for processing queries using full trie and time for processing queries using MCD-MCK has slightly increased as we increase the amount of data indexed. For instance, when indexing only 20% of the dataset, MCD-MCK is 12.4% slower than the full trie. This difference increased when indexing larger portions of the dataset, becoming 16.27% when indexing 100% of the dataset. The increase in differences is however worse when considering CPT, which is 40% slower than the full trie when indexing 20% of the dataset, and 101% slower when indexing the full JusBrasil dataset. This increase in the differences between the methods should be considered and carefully studied when indexing larger query suggestion datasets. We can see in the figure that MCD-MCK presents a significant reduction in memory requirements when compared to the full trie, while keeping close time performance.

Table 16 Selected parameters for BEVA using MCD-MCK in DBLP and UMBC datasets

Method	Datasets	
	DBLP	UMBC
MCD-MCK (MCD,MCK)	(10, 200)	(8, 200)

6.4 Comparing trie indexes in mode word by word

The experiments so far considered a scenario where the system performs matches between the whole prefix typed by the user and the complete string of each query suggestion, called mode 1 in the taxonomy presented by Krishnan et al. (2017), but here we consider it allowing errors. Another possible usage for the data structures compared here is a scenario where the match is performed word by word, which is mode 2 of the taxonomy, but here also including the possibility of allowing errors.

In this new scenario, we store the vocabulary of the query suggestions containing all distinct words in it, and create a list of suggestions associated with each word. Such a list points to all query suggestions where the word occurs. The prefix typed by the user is parsed and divided into words. The first step is to compute the match between the words found in the prefix typed by the user and the words found in the vocabulary. After matching the words of the prefix typed to the words in the vocabulary, the system may perform list operations, such as intersection or union of suggestions, to find the final set of suggestions to be presented to the user. Here we present only the performance of the first step, since we are using data structures to perform prefix match operations.

Table 15 presents the results achieved by the compared data structures when performing word prefix match in the JusBrasil dataset. Memory usage reported here includes only the space adopted to store the vocabulary and the data structure adopted on each experiment, not including the size to store the suggestion dataset nor the space required to represent the inverted lists associated with each vocabulary entry. The time performance also reported only includes the time for performing error-tolerant prefix search on the vocabulary. Each word parsed from the prefix is submitted as a prefix search for the system and we get as a result the list of words from the dataset to match the words found in the prefix queries. Time performance of MCD-MCK was quite close to the times achieved when using full tries, and the space required was again several times smaller. Full trie required 196.42 MiB, while MCD-MCK required only 54.43 MiB. The space required for CPT in this scenario was slightly smaller than MCD-MCK, while its time performance was slightly worse. We may say both CPT and MCD-MCK provide a good trade-off between memory usage and time performance when implementing word prefix search for the JusBrasil dataset. (Table 16).

The time performance of the suffix array was also fairly worse than the other data structures in this scenario, but the times obtained for all error levels are quite small when compared to the limit of 100 milliseconds usually adopted for query autocompletion systems.

6.5 Experiments with DBLP and UMBC datasets

Tables 17 and 18 present the results achieved when processing the queries with the synthetic datasets DBLP and UMBC when varying the number of errors and the prefix sizes. Table 16 presents the parameters chosen for the MCD-MCK on these two datasets. The

Table 17 Time performance (ms) and memory usage (MiB) when processing queries with BEVA and indexing DBLP dataset with data structures MCD-MCK (set to 10 and 200, respectively), full trie, CPT and suffix array

Methods	MEM (MiB)	Time (ms)					
		$\tau = 1$		$\tau = 2$		$\tau = 3$	
		9	17	9	17	9	17
MCD-MCK	505.6	0.12 ± 0.002	0.16 ± 0.003	1.11 ± 0.015	1.18 ± 0.016	5.67 ± 0.072	5.86 ± 0.077
full trie	4,309.7	0.13 ± 0.002	0.14 ± 0.004	1.12 ± 0.018	1.15 ± 0.018	5.56 ± 0.084	5.65 ± 0.088
CPT	555.1	0.21 ± 0.004	0.23 ± 0.004	1.91 ± 0.036	1.94 ± 0.037	10.71 ± 0.132	10.82 ± 0.136
suffix array	4,871.3	10.08 ± 1.285	2.20 ± 0.238	48.74 ± 3.292	12.7 ± 1.273	55.69 ± 4.001	231.42 ± 11.813

Table 18 Time performance (ms) and memory usage (MiB) when processing queries with BEVA and indexing the UMBC dataset with data structures MCD-MCK (set to 8 and 200, respectively), full trie, CPT and suffix array

Methods	MEM (MiB)	Time (ms)					
		$\tau = 1$		$\tau = 2$		$\tau = 3$	
		9	17	9	17	9	17
MCD-MCK	18,194	0.10 ± 0.001	0.10 ± 0.001	1.37 ± 0.018	1.40 ± 0.019	11.51 ± 0.163	11.68 ± 0.170
full trie	–	–	–	–	–	–	–
CPT	19,909	0.79 ± 0.023	0.83 ± 0.024	11.00 ± 0.264	11.13 ± 0.268	55.21 ± 1.205	55.90 ± 1.251
suffix array	–	–	–	–	–	–	–

parameter selection followed the same procedure we described to select the parameters for the method when processing the JusBrasil dataset.

We can see from the results presented both for DBLP and UMBC that the relative performance of the methods is compatible with the conclusions for experiments with the JusBrasil dataset. Again MCD-MCK introduces a useful alternative balance of time performance and required memory space. We noticed that we were not able to run full trie for processing the UMBC dataset, since it required more memory than we had available in our server (more than 64 GiB), while the MCD-MCK was able to process queries by using just about 18 GiB (18, 194 MiB). Furthermore, MCD-MCK was more than 7 times faster than CPT when processing queries of UMBC, being also faster than CPT when processing queries of DBLP. Finally, regarding the performance of the suffix array, we can see in these experiments that it might become more competitive in time performance only for $\tau = 1$ and for larger prefixes. The algorithm adopted for processing queries with the suffix array becomes more efficient as we increase the size of the searched prefix. However, this is a property that is not convenient for query autocompletion systems, where the systems usually start to search for suggestions just after a few words, for instance 3 or 4, are typed by the user.

7 Conclusion

We have proposed and studied alternative ways of using burst tries for implementing error-tolerant prefix search using the trie-based algorithm BEVA. The alternatives proposed are able to reduce memory consumption, while keeping a performance close to the ones achieved when using the full trie. When processing the JusBrasil dataset, the burst trie with MCD-MCK heuristics, was able to process queries with performances only 16% slower than the full trie index when implemented with BEVA, while yielding a significant reduction in memory usage, reducing it to almost one-fourth of the space required by the system using the full trie. MCD-MCK was also faster than CPTs in most of the experiments, even when using parameters that resulted in less memory usage than CPTs. A general conclusion is that the idea of using virtual nodes in tries can produce a good balance between efficiency and memory consumption, being a competitive alternative for implementing error-tolerant prefix search algorithms.

We also studied the impact of two optimizations when implementing the tries, namely, the range optimization and the BFS optimization. BFS optimization produces cache-friendly structures for processing query autocompletion, since they organize the index in an order that matches the one adopted by trie-based search algorithms, such as BEVA. Compared to the current DFS index building, we verified that BFS yielded a significant gain in time performance when processing queries for all scenarios and collections tested, being an alternative to be considered when designing autocompletion systems.

For future work, we plan to study alternative ways of using CPT trees combined with burst tries, considering the use of the CPT as the access trie in burst trie implementations. This direction brings several challenges, including alternative ways of combining the burst heuristics to the CPT. Further, the access trie is already a compact data structure when compared to the size of the dataset, and CPT has been shown to be slower than full trie, which may limit the potential benefits of this idea. A second future work will be to study the impact of the proposed ideas when implementing ranking and pruning functions for query autocompletion methods. The combination of pruning strategies with the burst trie implementations proposed here may result in time performances even closer to the ones of full tries, since pruning strategies may significantly reduce the access to nodes in deeper levels of the trie, making the performance of the full tries and burst tries closer.

Acknowledgements This work was partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) under funding grants No. 8882.347588/2019-01, 88887.130299/2017-01, and financial code 001; FAPEAM-POSGRAD 2020; CNPq grants No. 309765/2018-8 and 307248/2019-4; and by project MMBIAS (FAPESP MCTIC / CGI, 2020/05173-4).

References

Abouelhoda, M. I., Kurtz, S., & Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms, 2*(1), 53–86.

Acharya, A., Zhu, H., & Shen, K. (1999). Adaptive algorithms for cache-efficient trie search. *Workshop on Algorithm Engineering and Experimentation* (pp. 300–315). Heidelberg: Springer. Berlin.

Bast, Hannah, Kalmbach, Johannes, Klumpp, Theresa, Kramer, Florian, & Schnelle, Niklas. (2021). Efficient sparql autocompletion via sparql. arXiv preprint arXiv:2104.14595.

Bast, H., & Weber, I. (2006). Type less, find more: Fast autocompletion search with a succinct index. In *Proceedings of the Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, page 364-371. ISBN 1595933697. https://doi.org/10.1145/1148170.1148234.

Bast, H., Mortensen, C. W., & Weber, I. (2008). Output-sensitive autocompletion search. *Information Retrieval, 11*(4), 269–286. https://doi.org/10.1007/s10791-008-9048-x.

Belazzougui, D., Boldi, P., & Vigna, S. (2010). Dynamic z-fast tries. In *International Symposium on String Processing and Information Retrieval*, pp. 159–172. Springer.

Bender, M. A., Demaine, E. D., & Farach C. M. (2002). Efficient tree layout in a multilevel memory hierarchy. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pp. 165-173. Springer-Verlag. ISBN 3540441808. URL http://arxiv.org/abs/cs/0211010.

Binna, R., Zangerle, E., Pichl, M., Specht, Günther, & Leis, Viktor. (2018). Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the International ACM SIGMOD Conference on Management of Data*, pp. 521–534. https://doi.org/10.1145/3183713.3196896.

Cai, Fei, & Rijke, Maarten de. (2016). Learning from homologous queries and semantically related terms for query auto completion. *Information Processing & Management*, 52 (4): 628–643. ISSN 0306-4573. https://doi.org/10.1016/j.ipm.2015.12.008.

Chaudhuri, S., & Kaushik, R. (2009). Extending autocompletion to tolerate errors. In *ACM SIGMOD*, pp. 707–718. Association for Computing Machinery, Inc., June. https://doi.org/10.1145/1559845.1559919.

Chen, W., Cai, F., Chen, H., & de Rijke, M. (2020). Hierarchical neural query suggestion with an attention mechanism. *Information Processing & Management*, pp. 57 (6): 102040. ISSN 0306-4573. https://www.sciencedirect.com/science/article/pii/S0306457318308732.

Clark, D. R. (1998). *Compact Pat Trees*. PhD thesis. http://hdl.handle.net/10012/64.

Darragh, J. J., Cleary, J. G., & Witten, I. H. (1993). Bonsai: a compact representation of trees. *Software Practice and Experience, 23*(3), 277–291.

Deng, D., Li, G. W., He, J., H.V., & Feng, J. (June 2016). Meta: An efficient matching-based method for error-tolerant autocompletion. *Proc. VLDB Endow.*, pp. 9 (10): 828-839. ISSN 2150-8097. https://doi.org/10.14778/2977797.2977808.

Fredkin, E. (September 1960). Trie memory. *Commun. ACM*, 3 (9): 490-499. ISSN 0001-0782. https://doi.org/10.1145/367390.367400.

Ghasemi, C., Yousefi, H., Shin, K. G., & Zhang, B. (2018). A fast and memory-efficient trie structure for name-based packet forwarding. In *Proceedings of the International Conference on Network Protocols*, pp. 302–312. https://doi.org/10.1109/ICNP.2018.00046.

Gog, S., Pibiri, G. E., & Venturini, R. (2020). Efficient and effective query auto-completion. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2271–2280.

Gonnet, G. H., Baeza-Yates, R. A., & Snider, T. (1992). New indices for text: Pat trees and pat arrays. *Information Retrieval Data Structures Algorithms, 66,* 82.

Grabski, K., & Scheffer, T. (2004). Sentence completion. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, page 433-439. Association for Computing Machinery. ISBN 1581138814. https://doi.org/10.1145/1008992.1009066.

Heinz, S., Zobel, J., & Williams, H. E. (April 2002). Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, pp. 20 (2): 192-223. ISSN 1046-8188.

Holley, G., Wittler, R., & Stoye, J. (2016). Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology, 11*(1), 1–9. https://doi.org/10.1186/s13015-016-0066-8.

Sheng, H., Xiao, C., & Ishikawa, Y. (2018). An efficient algorithm for location-aware query autocompletion. *IEICE Transactions on Information and Systems, 101*(1), 181–192. https://doi.org/10.1145/3397271.3401432.

Jansson, J., Sadakane, K., & Sung, W.-K. (2015). Linked dynamic tries with applications to lz-compression in sublinear time and space. *Algorithmica, 71*(4), 969–988.

Ji, S., Li, G., Li, C., & Feng, J. (2009). Efficient interactive fuzzy keyword search. WWW'09, pp. 371–380. *Association for Computing Machinery*. https://doi.org/10.1145/1526709.1526760.

Kanda, S., Köppl, D., Tabei, Y., Morita, K., & Fuketa, M. (2020). Dynamic path-decomposed tries. *Journal of Experimental Algorithmics (JEA), 25,* 1–28.

Kang, Y. M., Liu W., & Zhou, Y. (2021). Queryblazer: Efficient query autocompletion framework. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, WSDM '21, pp. 1020-1028. Association for Computing Machinery. https://doi.org/10.1145/3397271.3401432.

Krishnan, U., Moffat, A., & Zobel, J. (2017). A taxonomy of query auto completion modes. In *Proceedings of the 22nd Australasian Document Computing Symposium*, ADCS 2017, New York, NY, USA. Association for Computing Machinery. ISBN 9781450363914. https://doi.org/10.1145/3166072.3166081.

Krishnan, U., Moffat, A., Zobel, J., & Billerbeck, B. (2020). Generation of synthetic query auto completion logs. In *European Conference on Information Retrieval*, pp. 621–635. Springer. URL https://link.springer.com/chapter/10.1007%2F978-3-030-45439-5_41.

Li, G., Ji, S., Li, C., & Feng, J. (2011). Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20: pp. 617–640, 08. https://doi.org/10.1007/s00778-011-0218-x.

Manber, U., & Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *Siam Journal on Computing, 22*(5), 935–948.

McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of the ACM, 23*(2), 262–272. https://doi.org/10.1145/321941.321946.

Miller, R. B. (1968). Response time in man-computer conversational transactions. *In Proceedings of the Fall Joint Computer Conference, part, I,* 267–277.

Morrison, D. R. (1968). Patricia-practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM), 15*(4), 514–534.

Nandi, A., & Jagadish, H.V. (2007). Effective phrase prediction. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB '07, pp. 219-230. ISBN 9781595936493. URL http://www.vldb.org/conf/2007/papers/research/p219-nandi.pdf.

Navarro, G., Sutinen, E., Tanninen, J., & Tarhio, J. (2000). Indexing text with approximate q-grams. In *Annual Symposium on Combinatorial Pattern Matching*, pp. 350–363. Springer.

Navarro, G., Sutinen, E., & Tarhio, J. (2005). Indexing text with approximate q-grams. *Journal of Discrete Algorithms, 3*(2–4), 157–175.

Pibiri, G. E., & Venturini, R. (2017). Efficient data structures for massive n-gram datasets. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 615–624.

Qin, J., Xiao, C., Hu, S., Zhang, J., Wang, W., Ishikawa, Y., Tsuda, K., & Sadakane, K. (2019). Efficient query autocompletion with edit distance-based error tolerance. *VLDB Journal*, pp. 1–25. https://doi.org/10.14778/2536336.2536339.

Smith, C. L., Gwizdka, J., & Feild, H. (2017). The use of query auto-completion over the course of search sessions with multifaceted information needs. *Information Processing & Management*, 53 (5): pp. 1139–1155. ISSN 0306-4573. https://doi.org/10.1016/j.ipm.2017.05.001.

Tahery, S., & Farzi, S. (2020). Customized query auto-completion and suggestion - a review. *Information Systems*, 87: 101415. URL https://www.sciencedirect.com/science/article/pii/S0306437919303072.

Ukkonen, E., & Wood, D. (November 1993). Approximate string matching with suffix automata. *Algorithmica*, 10 (5): pp. 353-364. ISSN 0178-4617. https://doi.org/10.1007/BF01769703.

Wang, J., & Lin, C. (2020). Fast error-tolerant location-aware query autocompletion. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1998–2001. IEEE. https://doi.org/10.1109/ICDE48307.2020.00223.

Chuan X., Jianbin Q., Wei W., Yoshiharu I., Koji T., and Kunihiko S.2013 Efficient error-tolerant query autocompletion. *Proc. VLDB Endow.*, 6 (6): pp. 373-384, ISSN 2150-8097. https://doi.org/10.14778/2536336.2536339.

Xie, G., Jingxiu, S., Wang, X., He, T., Zhang, G., Uhlig, S., & Salamatian, K. (2017). Index-trie: efficient archival and retrieval of network traffic. *Computer Networks, 124,* 140–156. https://doi.org/10.1016/j.comnet.2017.06.010.

Zhou, X., Qin, J., Xiao, C., Wang, W., Lin, X., & Ishikawa, Y. 2016. Beva: An efficient query processing algorithm for error-tolerant autocompletion. *ACM Trans. Database Syst.*, 41 (1). ISSN 0362-5915. https://doi.org/10.1145/2877201.

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory, 23*(3), 337–343.